SpringMVC框架第一天

第一章：三层架构和MVC

1. 三层架构

1. 咱们开发服务器端程序，一般都基于两种形式，一种C/S架构程序，一种B/S架构程序
2. 使用Java语言基本上都是开发B/S架构的程序，B/S架构又分成了三层架构
3. 三层架构
    1. 表现层：WEB层，用来和客户端进行数据交互的。表现层一般会采用MVC的设计模型
    2. 业务层：处理公司具体的业务逻辑的
    3. 持久层：用来操作数据库的

2. MVC模型

1. MVC全名是Model View Controller 模型视图控制器，每个部分各司其职。
2. Model：数据模型，JavaBean的类，用来进行数据封装。
3. View：指JSP、HTML用来展示数据给用户
4. Controller：用来接收用户的请求，整个流程的控制器。用来进行数据校验等。

第二章：SpringMVC的入门案例

1. SpringMVC的概述（查看大纲文档）

1. SpringMVC的概述
    1. 是一种基于Java实现的MVC设计模型的请求驱动类型的轻量级WEB框架。
    2. Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。
    3. 使用 Spring 可插入的 MVC 架构，从而在使用Spring进行WEB开发时，可以选择使用Spring的SpringMVC框架或集成其他MVC开发框架，如Struts1(现在一般不用)，Struts2等。
2. SpringMVC在三层架构中的位置
    1. 表现层框架
3. SpringMVC的优势
4. SpringMVC和Struts2框架的对比

2. SpringMVC的入门程序

1. 创建WEB工程，引入开发的jar包
    1. 具体的坐标如下

```
    <!-- 版本锁定 -->
    <properties>
        <spring.version>5.0.2.RELEASE</spring.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

2. 配置核心的控制器（配置DispatcherServlet）
    1. 在web.xml配置文件中核心控制器DispatcherServlet

```
    <!-- SpringMVC的核心控制器 -->
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!-- 配置Servlet的初始化参数，读取springmvc的配置文件，创建spring容器 -->

        <init-param>
```

```
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!-- 配置servlet启动时加载对象 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

3. 编写springmvc.xml的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 配置spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.itheima"></context:component-scan>

    <!-- 配置视图解析器 -->
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

    <!-- 配置spring开启注解mvc的支持
    <mvc:annotation-driven></mvc:annotation-driven>-->
</beans>
```

4. 编写index.jsp和HelloController控制器类

    1. index.jsp

```
<body>

    <h3>入门案例</h3>

    <a href="${ pageContext.request.contextPath }/hello">入门案例</a>

</body>
```

2. HelloController

```
package cn.itcast.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * 控制器
 * @author rt
 */
@Controller
public class HelloController {

    /**
     * 接收请求
     * @return
     */
    @RequestMapping(path="/hello")
    public String sayHello() {
        System.out.println("Hello SpringMVC!!");
        return "success";
    }

}
```

5. 在WEB-INF目录下创建pages文件夹，编写success.jsp的成功页面

```
<body>

    <h3>入门成功！！</h3>

</body>
```

6. 启动Tomcat服务器，进行测试

3. 入门案例的执行过程分析

1. 入门案例的执行流程
    1. 当启动Tomcat服务器的时候，因为配置了load-on-startup标签，所以会创建DispatcherServlet对象，就会加载springmvc.xml配置文件
    2. 开启了注解扫描，那么HelloController对象就会被创建
    3. 从index.jsp发送请求，请求会先到达DispatcherServlet核心控制器，根据配置@RequestMapping注解找到执行的具体方法
    4. 根据执行方法的返回值，再根据配置的视图解析器，去指定的目录下查找指定名称的JSP文件
    5. Tomcat服务器渲染页面，做出响应
2. SpringMVC官方提供图形

3. 入门案例中的组件分析
    1. 前端控制器（DispatcherServlet）
    2. 处理器映射器（HandlerMapping）
    3. 处理器（Handler）
    4. 处理器适配器（HandlAdapter）
    5. 视图解析器（View Resolver）
    6. 视图（View）

4. RequestMapping注解

1. RequestMapping注解的作用是建立请求URL和处理方法之间的对应关系
2. RequestMapping注解可以作用在方法和类上
    1. 作用在类上：第一级的访问目录
    2. 作用在方法上：第二级的访问目录
    3. 细节：路径可以不编写 / 表示应用的根目录开始
    4. 细节：${ pageContext.request.contextPath }也可以省略不写，但是路径上不能写 /
3. RequestMapping的属性
    1. path　　指定请求路径的url
    2. value　　value属性和path属性是一样的
    3. mthod　 指定该方法的请求方式
    4. params　指定限制请求参数的条件
    5. headers　发送的请求中必须包含的请求头

第三章：请求参数的绑定

1. 请求参数的绑定说明
    1. 绑定机制
        1. 表单提交的数据都是k=v格式的　username=haha&password=123
        2. SpringMVC的参数绑定过程是把表单提交的请求参数，作为控制器中方法的参数进行绑定的
        3. 要求：提交表单的name和参数的名称是相同的
    2. 支持的数据类型
        1. 基本数据类型和字符串类型
        2. 实体类型（JavaBean）
        3. 集合数据类型（List、map集合等）
2. 基本数据类型和字符串类型
    1. 提交表单的name和参数的名称是相同的
    2. 区分大小写
3. 实体类型（JavaBean）
    1. 提交表单的name和JavaBean中的属性名称需要一致
    2. 如果一个JavaBean类中包含其他的引用类型，那么表单的name属性需要编写成：对象.属性 例如：address.name
4. 给集合属性数据封装

1. JSP页面编写方式：list[0].属性

5. 请求参数中文乱码的解决
    1. 在web.xml中配置Spring提供的过滤器类

```
    <!-- 配置过滤器，解决中文乱码的问题 -->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
        <!-- 指定字符集 -->
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

6. 自定义类型转换器
    1. 表单提交的任何数据类型全部都是字符串类型，但是后台定义Integer类型，数据也可以封装上，说明Spring框架内部会默认进行数据类型转换。
    2. 如果想自定义数据类型转换，可以实现Converter的接口
        1. 自定义类型转换器

```
package cn.itcast.utils;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.core.convert.converter.Converter;

/**
 * 把字符串转换成日期的转换器
 * @author rt
 */
public class StringToDateConverter implements Converter<String, Date>{

    /**
     * 进行类型转换的方法
     */
    public Date convert(String source) {
        // 判断
        if(source == null) {
            throw new RuntimeException("参数不能为空");

        }
```

```
        try {
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            // 解析字符串
            Date date = df.parse(source);
            return date;
        } catch (Exception e) {
            throw new RuntimeException("类型转换错误");
        }
    }

}
```

2. 注册自定义类型转换器，在springmvc.xml配置文件中编写配置

```
    <!-- 注册自定义类型转换器 -->
    <bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="cn.itcast.utils.StringToDateConverter"/>
            </set>
        </property>
    </bean>

    <!-- 开启Spring对MVC注解的支持 -->
    <mvc:annotation-driven conversion-service="conversionService"/>
```

7. 在控制器中使用原生的ServletAPI对象
    1. 只需要在控制器的方法参数定义HttpServletRequest和HttpServletResponse对象

第四章：常用的注解

1. RequestParam注解
    1. 作用：把请求中的指定名称的参数传递给控制器中的形参赋值
    2. 属性
        1. value：请求参数中的名称
        2. required：请求参数中是否必须提供此参数，默认值是true，必须提供
    3. 代码如下

```
/**
 * 接收请求
 * @return
 */
@RequestMapping(path="/hello")
public String sayHello(@RequestParam(value="username",required=false)String name) {
    System.out.println("aaaa");
    System.out.println(name);
    return "success";
}
```

2. RequestBody注解
    1. 作用：用于获取请求体的内容（注意：get方法不可以）
    2. 属性
        1. required：是否必须有请求体，默认值是true
    3. 代码如下

```
/**
 * 接收请求
 * @return
 */
@RequestMapping(path="/hello")
public String sayHello(@RequestBody String body) {
    System.out.println("aaaa");
    System.out.println(body);
    return "success";
}
```

3. PathVariable注解
    1. 作用：拥有绑定url中的占位符的。例如：url中有/delete/{id}，{id}就是占位符
    2. 属性
        1. value：指定url中的占位符名称
    3. Restful风格的URL
        1. 请求路径一样，可以根据不同的请求方式去执行后台的不同方法
        2. restful风格的URL优点
            1. 结构清晰
            2. 符合标准
            3. 易于理解
            4. 扩展方便
    4. 代码如下

```
    <a href="user/hello/1">入门案例</a>

    /**
     * 接收请求
     * @return
     */
    @RequestMapping(path="/hello/{id}")
    public String sayHello(@PathVariable(value="id") String id) {
        System.out.println(id);
        return "success";
    }
```

4. RequestHeader注解
    1. 作用：获取指定请求头的值
    2. 属性
        1. value：请求头的名称
    3. 代码如下

```
@RequestMapping(path="/hello")
    public String sayHello(@RequestHeader(value="Accept") String header) {
        System.out.println(header);
        return "success";
    }
```

5. CookieValue注解
    1. 作用：用于获取指定cookie的名称的值
    2. 属性
        1. value：cookie的名称
    3. 代码

```
    @RequestMapping(path="/hello")
    public String sayHello(@CookieValue(value="JSESSIONID") String cookieValue) {
        System.out.println(cookieValue);
        return "success";
    }
```

6. ModelAttribute注解
    1. 作用
        1. 出现在方法上：表示当前方法会在控制器方法执行前线执行。
        2. 出现在参数上：获取指定的数据给参数赋值。
    2. 应用场景
        1. 当提交表单数据不是完整的实体数据时，保证没有提交的字段使用数据库原来的数据。
    3. 具体的代码

1. 修饰的方法有返回值

```
/**
 * 作用在方法，先执行
 * @param name
 * @return
 */
@ModelAttribute
public User showUser(String name) {
    System.out.println("showUser执行了...");
    // 模拟从数据库中查询对象
    User user = new User();
    user.setName("哈哈");
    user.setPassword("123");
    user.setMoney(100d);
    return user;
}

/**
 * 修改用户的方法
 * @param cookieValue
 * @return
 */
@RequestMapping(path="/updateUser")
public String updateUser(User user) {
    System.out.println(user);
    return "success";
}
```

2. 修饰的方法没有返回值

```
/**
 * 作用在方法，先执行
 * @param name
 * @return
 */
@ModelAttribute
public void showUser(String name,Map<String, User> map) {
    System.out.println("showUser执行了...");
    // 模拟从数据库中查询对象
    User user = new User();
    user.setName("哈哈");
    user.setPassword("123");
    user.setMoney(100d);
    map.put("abc", user);
}

/**
 * 修改用户的方法
 * @param cookieValue
 * @return
 */
```

```
    @RequestMapping(path="/updateUser")
    public String updateUser(@ModelAttribute(value="abc") User user) {
        System.out.println(user);
        return "success";
    }
```

4. SessionAttributes注解
    1. 作用：用于多次执行控制器方法间的参数共享
    2. 属性
        1. value：指定存入属性的名称
    3. 代码如下

```
@Controller
@RequestMapping(path="/user")
@SessionAttributes(value= {"username","password","age"},types=
{String.class,Integer.class})        // 把数据存入到session域对象中
public class HelloController {

    /**
     * 向session中存入值
     * @return
     */
    @RequestMapping(path="/save")
    public String save(Model model) {
        System.out.println("向session域中保存数据");
        model.addAttribute("username", "root");
        model.addAttribute("password", "123");
        model.addAttribute("age", 20);
        return "success";
    }

    /**
     * 从session中获取值
     * @return
     */
    @RequestMapping(path="/find")
    public String find(ModelMap modelMap) {
        String username = (String) modelMap.get("username");
        String password = (String) modelMap.get("password");
        Integer age = (Integer) modelMap.get("age");
        System.out.println(username + " : "+password +" : "+age);
        return "success";
    }

    /**
     * 清除值
     * @return
     */
    @RequestMapping(path="/delete")

    public String delete(SessionStatus status) {
```

```
        status.setComplete();
        return "success";
    }

}
```

课程总结

1. SpringMVC的概述
2. 入门
    1. 创建工程，导入坐标
    2. 在web.xml中配置前端控制器（启动服务器，加载springmvc.xml配置文件）
    3. 编写springmvc.xml配置文件
    4. 编写index.jsp的页面，发送请求
    5. 编写Controller类，编写方法（@RequestMapping(path="/hello")），处理请求
    6. 编写配置文件（开启注解扫描），配置视图解析器
    7. 执行的流程
    8. @RequestMapping注解
        1. path
        2. value
        3. method
        4. ....
3. 参数绑定
    1. 参数绑定必须会
    2. 解决中文乱码，配置过滤器
    3. 自定义数据类型转换器

SpringMVC框架第二天

第一章：响应数据和结果视图

1. 返回值分类

1. 返回字符串
    1. Controller方法返回字符串可以指定逻辑视图的名称，根据视图解析器为物理视图的地址。

```
    @RequestMapping(value="/hello")
    public String sayHello() {
        System.out.println("Hello SpringMVC!!");
        // 跳转到XX页面
        return "success";
    }
```

    2. 具体的应用场景

```
@Controller
@RequestMapping("/user")
public class UserController {

    /**
     * 请求参数的绑定
     */
    @RequestMapping(value="/initUpdate")
    public String initUpdate(Model model) {
        // 模拟从数据库中查询的数据
        User user = new User();
        user.setUsername("张三");
        user.setPassword("123");
        user.setMoney(100d);
        user.setBirthday(new Date());
        model.addAttribute("user", user);
        return "update";
    }

}
```

```
    <h3>修改用户</h3>
    ${ requestScope }
    <form action="user/update" method="post">
        姓名：<input type="text" name="username" value="${ user.username }"><br>
        密码：<input type="text" name="password" value="${ user.password }"><br>
        金额：<input type="text" name="money" value="${ user.money }"><br>
        <input type="submit" value="提交">

    </form>
```

2. 返回值是void

    1. 如果控制器的方法返回值编写成void，执行程序报404的异常，默认查找JSP页面没有找到。

        1. 默认会跳转到@RequestMapping(value="/initUpdate") initUpdate的页面。

    2. 可以使用请求转发或者重定向跳转到指定的页面

```
@RequestMapping(value="/initAdd")
    public void initAdd(HttpServletRequest request,HttpServletResponse response) throws
Exception {
        System.out.println("请求转发或者重定向");
        // 请求转发
        // request.getRequestDispatcher("/WEB-INF/pages/add.jsp").forward(request,
response);
        // 重定向
        // response.sendRedirect(request.getContextPath()+"/add2.jsp");

        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/html;charset=UTF-8");

        // 直接响应数据
        response.getWriter().print("你好");
        return;
    }
```

3. 返回值是ModelAndView对象

    1. ModelAndView对象是Spring提供的一个对象，可以用来调整具体的JSP视图

    2. 具体的代码如下

```
    /**
     * 返回ModelAndView对象
     * 可以传入视图的名称（即跳转的页面），还可以传入对象。
     * @return
     * @throws Exception
     */
    @RequestMapping(value="/findAll")
    public ModelAndView findAll() throws Exception {
        ModelAndView mv = new ModelAndView();
        // 跳转到list.jsp的页面
        mv.setViewName("list");

        // 模拟从数据库中查询所有的用户信息
        List<User> users = new ArrayList<>();
        User user1 = new User();
        user1.setUsername("张三");
        user1.setPassword("123");

        User user2 = new User();
        user2.setUsername("赵四");
        user2.setPassword("456");
```

```
        users.add(user1);
        users.add(user2);
        // 添加对象
        mv.addObject("users", users);

        return mv;
    }

    <%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

    <h3>查询所有的数据</h3>
    <c:forEach items="${ users }" var="user">
        ${ user.username }
    </c:forEach>

</body>
</html>
```

2. SpringMVC框架提供的转发和重定向

1. forward请求转发
    1. controller方法返回String类型，想进行请求转发也可以编写成

```
    /**
     * 使用forward关键字进行请求转发
     * "forward:转发的JSP路径"，不走视图解析器了，所以需要编写完整的路径
     * @return
     * @throws Exception
     */
    @RequestMapping("/delete")
    public String delete() throws Exception {
        System.out.println("delete方法执行了...");
        // return "forward:/WEB-INF/pages/success.jsp";
        return "forward:/user/findAll";
    }
```

2. redirect重定向

1. controller方法返回String类型，想进行重定向也可以编写成

```
    /**
     * 重定向
     * @return
     * @throws Exception
     */
    @RequestMapping("/count")
    public String count() throws Exception {
        System.out.println("count方法执行了...");
        return "redirect:/add.jsp";
        // return "redirect:/user/findAll";
    }
```

3. ResponseBody响应json数据

1. DispatcherServlet会拦截到所有的资源，导致一个问题就是静态资源（img、css、js）也会被拦截到，从而不能被使用。解决问题就是需要配置静态资源不进行拦截，在springmvc.xml配置文件添加如下配置
    1. mvc:resources标签配置不过滤
        1. location元素表示webapp目录下的包下的所有文件
        2. mapping元素表示以/static开头的所有请求路径，如/static/a 或者/static/a/b

```
<!-- 设置静态资源不过滤 -->
<mvc:resources location="/css/" mapping="/css/**"/>  <!-- 样式 -->
<mvc:resources location="/images/" mapping="/images/**"/>  <!-- 图片 -->
<mvc:resources location="/js/" mapping="/js/**"/>  <!-- javascript -->
```

2. 使用@RequestBody获取请求体数据

```
// 页面加载
// 页面加载
$(function(){
    // 绑定点击事件
    $("#btn").click(function(){
        $.ajax({
            url:"user/testJson",
            contentType:"application/json;charset=UTF-8",
            data:'{"addressName":"aa","addressNum":100}',
            dataType:"json",
            type:"post",
            success:function(data){
                alert(data);
                alert(data.addressName);
            }
        });
    });
});


/**
```

```
     * 获取请求体的数据
     * @param body
     */
    @RequestMapping("/testJson")
    public void testJson(@RequestBody String body) {
        System.out.println(body);
    }
```

3. 使用@RequestBody注解把json的字符串转换成JavaBean的对象

```
    // 页面加载
    // 页面加载
    $(function(){
        // 绑定点击事件
        $("#btn").click(function(){
            $.ajax({
                url:"user/testJson",
                contentType:"application/json;charset=UTF-8",
                data:'{"addressName":"aa","addressNum":100}',
                dataType:"json",
                type:"post",
                success:function(data){
                    alert(data);
                    alert(data.addressName);
                }
            });
        });
    });

    /**
     * 获取请求体的数据
     * @param body
     */
    @RequestMapping("/testJson")
    public void testJson(@RequestBody Address address) {
        System.out.println(address);
    }
```

4. 使用@ResponseBody注解把JavaBean对象转换成json字符串，直接响应

    1. 要求方法需要返回JavaBean的对象

```
// 页面加载
    $(function(){
        // 绑定点击事件
        $("#btn").click(function(){
            $.ajax({
                url:"user/testJson",
                contentType:"application/json;charset=UTF-8",

                data:'{"addressName":"哈哈","addressNum":100}',
```

```
                dataType:"json",
                type:"post",
                success:function(data){
                    alert(data);
                    alert(data.addressName);
                }
            });
        });
    });

    @RequestMapping("/testJson")
    public @ResponseBody Address testJson(@RequestBody Address address) {
        System.out.println(address);
        address.setAddressName("上海");
        return address;
    }
```

5. json字符串和JavaBean对象互相转换的过程中，需要使用jackson的jar包

```
    <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.9.0</version>
        </dependency>
```

第二章：SpringMVC实现文件上传

1. 文件上传的回顾

1. 导入文件上传的jar包

```
<dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
    </dependency>
    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.4</version>
    </dependency>
```

2. 编写文件上传的JSP页面

```
<h3>文件上传</h3>

<form action="user/fileupload" method="post" enctype="multipart/form-data">
    选择文件：<input type="file" name="upload"/><br/>
    <input type="submit" value="上传文件"/>
</form>
```

3. 编写文件上传的Controller控制器

```
/**
 * 文件上传
 * @throws Exception
 */
@RequestMapping(value="/fileupload")
public String fileupload(HttpServletRequest request) throws Exception {
    // 先获取到要上传的文件目录
    String path = request.getSession().getServletContext().getRealPath("/uploads");
    // 创建File对象，一会向该路径下上传文件
    File file = new File(path);
    // 判断路径是否存在，如果不存在，创建该路径
    if(!file.exists()) {
        file.mkdirs();
    }
    // 创建磁盘文件项工厂
    DiskFileItemFactory factory = new DiskFileItemFactory();
    ServletFileUpload fileUpload = new ServletFileUpload(factory);
    // 解析request对象
    List<FileItem> list = fileUpload.parseRequest(request);
    // 遍历
    for (FileItem fileItem : list) {
        // 判断文件项是普通字段，还是上传的文件
        if(fileItem.isFormField()) {

        }else {
            // 上传文件项
```

```
                // 获取到上传文件的名称
                String filename = fileItem.getName();
                // 上传文件
                fileItem.write(new File(file, filename));
                // 删除临时文件
                fileItem.delete();
            }
        }

        return "success";
    }
```

2. SpringMVC传统方式文件上传

1. SpringMVC框架提供了MultipartFile对象，该对象表示上传的文件，要求变量名称必须和表单file标签的name属性名称相同。
2. 代码如下

```
    /**
     * SpringMVC方式的文件上传
     *
     * @param request
     * @return
     * @throws Exception
     */
    @RequestMapping(value="/fileupload2")
    public String fileupload2(HttpServletRequest request,MultipartFile upload) throws
Exception {
        System.out.println("SpringMVC方式的文件上传...");
        // 先获取到要上传的文件目录
        String path = request.getSession().getServletContext().getRealPath("/uploads");
        // 创建File对象，一会向该路径下上传文件
        File file = new File(path);
        // 判断路径是否存在，如果不存在，创建该路径
        if(!file.exists()) {
            file.mkdirs();
        }
        // 获取到上传文件的名称
        String filename = upload.getOriginalFilename();
        String uuid = UUID.randomUUID().toString().replaceAll("-", "").toUpperCase();
        // 把文件的名称唯一化
        filename = uuid+"_"+filename;
        // 上传文件
        upload.transferTo(new File(file,filename));
        return "success";
    }
```

3. 配置文件解析器对象

```
<!-- 配置文件解析器对象，要求id名称必须是multipartResolver -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <property name="maxUploadSize" value="10485760"/>
    </bean>
```

3. SpringMVC跨服务器方式文件上传

1. 搭建图片服务器
    1. 根据文档配置tomcat9的服务器，现在是2个服务器
    2. 导入资料中day02_springmvc5_02image项目，作为图片服务器使用
2. 实现SpringMVC跨服务器方式文件上传
    1. 导入开发需要的jar包

```
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-core</artifactId>
            <version>1.18.1</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-client</artifactId>
            <version>1.18.1</version>
        </dependency>
```

    2. 编写文件上传的JSP页面

```
<h3>跨服务器的文件上传</h3>

    <form action="user/fileupload3" method="post" enctype="multipart/form-data">
        选择文件：<input type="file" name="upload"/><br/>
        <input type="submit" value="上传文件"/>
    </form>
```

    3. 编写控制器

```
    /**
     * SpringMVC跨服务器方式的文件上传
     *
     * @param request
     * @return
     * @throws Exception
     */

    @RequestMapping(value="/fileupload3")
```

```
    public String fileupload3(MultipartFile upload) throws Exception {
        System.out.println("SpringMVC跨服务器方式的文件上传...");

        // 定义图片服务器的请求路径
        String path = "http://localhost:9090/day02_springmvc5_02image/uploads/";

        // 获取到上传文件的名称
        String filename = upload.getOriginalFilename();
        String uuid = UUID.randomUUID().toString().replaceAll("-", "").toUpperCase();
        // 把文件的名称唯一化
        filename = uuid+"_"+filename;
        // 向图片服务器上传文件

        // 创建客户端对象
        Client client = Client.create();
        // 连接图片服务器
        WebResource webResource = client.resource(path+filename);
        // 上传文件
        webResource.put(upload.getBytes());
        return "success";
    }
```

第三章：SpringMVC的异常处理

1. 异常处理思路

1. Controller调用service，service调用dao，异常都是向上抛出的，最终有DispatcherServlet找异常处理器进行异常的处理。

2. SpringMVC的异常处理

1. 自定义异常类

```
package cn.itcast.exception;

public class SysException extends Exception{

    private static final long serialVersionUID = 4055945147128016300L;

    // 异常提示信息
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public SysException(String message) {
        this.message = message;
    }
```

```
}
```

2. 自定义异常处理器

```
package cn.itcast.exception;

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

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

/**
 * 异常处理器
 * @author rt
 */
public class SysExceptionResolver implements HandlerExceptionResolver{

    /**
     * 跳转到具体的错误页面的方法
     */
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse
response, Object handler,
            Exception ex) {
        ex.printStackTrace();
        SysException e = null;
        // 获取到异常对象
        if(ex instanceof SysException) {
            e = (SysException) ex;
        }else {
            e = new SysException("请联系管理员");
        }
        ModelAndView mv = new ModelAndView();
        // 存入错误的提示信息
        mv.addObject("message", e.getMessage());
        // 跳转的Jsp页面
        mv.setViewName("error");
        return mv;
    }

}
```

3. 配置异常处理器

```
<!-- 配置异常处理器 -->
<bean id="sysExceptionResolver" class="cn.itcast.exception.SysExceptionResolver"/>
```

第四章：SpringMVC框架中的拦截器

1. 拦截器的概述

1. SpringMVC框架中的拦截器用于对处理器进行预处理和后处理的技术。
2. 可以定义拦截器链，连接器链就是将拦截器按着一定的顺序结成一条链，在访问被拦截的方法时，拦截器链中的拦截器会按着定义的顺序执行。
3. 拦截器和过滤器的功能比较类似，有区别
    1. 过滤器是Servlet规范的一部分，任何框架都可以使用过滤器技术。
    2. 拦截器是SpringMVC框架独有的。
    3. 过滤器配置了/*，可以拦截任何资源。
    4. 拦截器只会对控制器中的方法进行拦截。
4. 拦截器也是AOP思想的一种实现方式
5. 想要自定义拦截器，需要实现HandlerInterceptor接口。

2. 自定义拦截器步骤

1. 创建类，实现HandlerInterceptor接口，重写需要的方法

```
package cn.itcast.demo1;

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

import org.springframework.web.servlet.HandlerInterceptor;

/**
 * 自定义拦截器1
 * @author rt
 */
public class MyInterceptor1 implements HandlerInterceptor{

    /**
     * controller方法执行前，进行拦截的方法
     * return true放行
     * return false拦截
     * 可以使用转发或者重定向直接跳转到指定的页面。
     */
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler)
            throws Exception {
        System.out.println("拦截器执行了...");
        return true;
    }

}
```

2. 在springmvc.xml中配置拦截器类

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <!-- 哪些方法进行拦截 -->
        <mvc:mapping path="/user/*"/>
        <!-- 哪些方法不进行拦截
        <mvc:exclude-mapping path=""/>
        -->
        <!-- 注册拦截器对象 -->
        <bean class="cn.itcast.demo1.MyInterceptor1"/>
    </mvc:interceptor>
</mvc:interceptors>
```

3. HandlerInterceptor接口中的方法

1. preHandle方法是controller方法执行前拦截的方法
    1. 可以使用request或者response跳转到指定的页面
    2. return true放行，执行下一个拦截器，如果没有拦截器，执行controller中的方法。
    3. return false不放行，不会执行controller中的方法。
2. postHandle是controller方法执行后执行的方法，在JSP视图执行前。
    1. 可以使用request或者response跳转到指定的页面
    2. 如果指定了跳转的页面，那么controller方法跳转的页面将不会显示。
3. postHandle方法是在JSP执行后执行
    1. request或者response不能再跳转页面了

3. 配置多个拦截器

1. 再编写一个拦截器的类
2. 配置2个拦截器

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <mvc:interceptor>
        <!-- 哪些方法进行拦截 -->
        <mvc:mapping path="/user/*"/>
        <!-- 哪些方法不进行拦截
        <mvc:exclude-mapping path=""/>
        -->
```

```
            <!-- 注册拦截器对象 -->
            <bean class="cn.itcast.demo1.MyInterceptor1"/>
        </mvc:interceptor>

        <mvc:interceptor>
            <!-- 哪些方法进行拦截 -->
            <mvc:mapping path="/**"/>
            <!-- 注册拦截器对象 -->
            <bean class="cn.itcast.demo1.MyInterceptor2"/>
        </mvc:interceptor>
    </mvc:interceptors>
```

SpringMVC第三天

第一章：搭建整合环境

1. 搭建整合环境

1. 整合说明：SSM整合可以使用多种方式，咱们会选择XML + 注解的方式
2. 整合的思路
    1. 先搭建整合的环境
    2. 先把Spring的配置搭建完成
    3. 再使用Spring整合SpringMVC框架
    4. 最后使用Spring整合MyBatis框架
3. 创建数据库和表结构
    1. 语句

```
create database ssm;
use ssm;
create table account(
    id int primary key auto_increment,
    name varchar(20),
    money double
);
```

4. 创建maven的工程（今天会使用到工程的聚合和拆分的概念，这个技术maven高级会讲）
    1. 创建ssm_parent父工程（打包方式选择pom，必须的）
    2. 创建ssm_web子模块（打包方式是war包）
    3. 创建ssm_service子模块（打包方式是jar包）
    4. 创建ssm_dao子模块（打包方式是jar包）
    5. 创建ssm_domain子模块（打包方式是jar包）
    6. web依赖于service，service依赖于dao，dao依赖于domain
    7. 在ssm_parent的pom.xml文件中引入坐标依赖

```
<properties>
        <spring.version>5.0.2.RELEASE</spring.version>
        <slf4j.version>1.6.6</slf4j.version>
        <log4j.version>1.2.12</log4j.version>
        <mysql.version>5.1.6</mysql.version>
        <mybatis.version>3.4.5</mybatis.version>
    </properties>

    <dependencies>
```

```
        <!-- spring -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.6.8</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>

            <version>4.12</version>
```

```
            <scope>compile</scope>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>jstl</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>

        <!-- log start -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
        <!-- log end -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>${mybatis.version}</version>

        </dependency>
```

```
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.0</version>
        </dependency>

        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>

    </dependencies>

<build>
        <finalName>ssm</finalName>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.2</version>
                    <configuration>
                        <source>1.8</source>
                        <target>1.8</target>
                        <encoding>UTF-8</encoding>
                        <showWarnings>true</showWarnings>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
```

8. 部署ssm_web的项目，只要把ssm_web项目加入到tomcat服务器中即可

5. 编写实体类，在ssm_domain项目中编写

```
package cn.itcast.domain;

import java.io.Serializable;

public class Account implements Serializable{

    private static final long serialVersionUID = 7355810572012650248L;

    private Integer id;
    private String name;
    private Double money;


    public Integer getId() {
```

```
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Double getMoney() {
        return money;
    }
    public void setMoney(Double money) {
        this.money = money;
    }

}
```

6. 编写dao接口

```
package cn.itcast.dao;

import java.util.List;

import cn.itcast.domain.Account;

public interface AccountDao {

    public void saveAccount(Account account);

    public List<Account> findAll();

}
```

7. 编写service接口和实现类

```
package cn.itcast.service;

import java.util.List;

import cn.itcast.domain.Account;

public interface AccountService {
```

```
    public void saveAccount(Account account);

    public List<Account> findAll();

}

package cn.itcast.service.impl;

import java.util.List;

import org.springframework.stereotype.Service;

import cn.itcast.dao.AccountDao;
import cn.itcast.domain.Account;
import cn.itcast.service.AccountService;

@Service("accountService")
public class AccountServiceImpl implements AccountService {

    private AccountDao account;

    public void saveAccount(Account account) {
    }

    public List<Account> findAll() {
        System.out.println("业务层：查询所有账户...");
        return null;
    }

}
```

第二章：Spring框架代码的编写

1. 搭建和测试Spring的开发环境

1. 在ssm_web项目中创建applicationContext.xml的配置文件，编写具体的配置信息。

```
<?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:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
```

```
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- 开启注解扫描，要扫描的是service和dao层的注解，要忽略web层注解，因为web层让SpringMVC框架
去管理 -->
    <context:component-scan base-package="cn.itcast">
        <!-- 配置要忽略的注解 -->
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

</beans>
```

2. 在ssm_web项目中编写测试方法，进行测试

```
package cn.itcast.test;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.itcast.service.AccountService;

public class ServiceTest {

    @Test
    public void run1() {
        ApplicationContext ac = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
        AccountService as = (AccountService) ac.getBean("accountService");
        as.findAll();
    }

}
```

第三章：Spring整合SpringMVC框架

1. 搭建和测试SpringMVC的开发环境

1. 在web.xml中配置DispatcherServlet前端控制器

```
    <!-- 配置前端控制器：服务器启动必须加载，需要加载springmvc.xml配置文件 -->
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 配置初始化参数，创建完DispatcherServlet对象，加载springmvc.xml配置文件 -->
        <init-param>
```

```
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <!-- 服务器启动的时候，让DispatcherServlet对象创建 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

2. 在web.xml中配置DispatcherServlet过滤器解决中文乱码

```
    <!-- 配置解决中文乱码的过滤器 -->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

3. 创建springmvc.xml的配置文件，编写配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 扫描controller的注解，别的不扫描 -->
    <context:component-scan base-package="cn.itcast">
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置视图解析器 -->
    <bean id="viewResolver"
```

```
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- JSP文件所在的目录 -->
        <property name="prefix" value="/WEB-INF/pages/" />
        <!-- 文件的后缀名 -->
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- 设置静态资源不过滤 -->
    <mvc:resources location="/css/" mapping="/css/**" />
    <mvc:resources location="/images/" mapping="/images/**" />
    <mvc:resources location="/js/" mapping="/js/**" />

    <!-- 开启对SpringMVC注解的支持 -->
    <mvc:annotation-driven />

</beans>
```

4. 测试SpringMVC的框架搭建是否成功
    1. 编写index.jsp和list.jsp编写，超链接

    ```
    <a href="account/findAll">查询所有</a>
    ```

    2. 创建AccountController类，编写方法，进行测试

    ```
    package cn.itcast.controller;

    import org.springframework.stereotype.Controller;
    import org.springframework.web.bind.annotation.RequestMapping;

    @Controller
    @RequestMapping("/account")
    public class AccountController {

        /**
         * 查询所有的数据
         * @return
         */
        @RequestMapping("/findAll")
        public String findAll() {
            System.out.println("表现层：查询所有账户...");
            return "list";
        }

    }
    ```

2. Spring整合SpringMVC的框架

1. 目的：在controller中能成功的调用service对象中的方法。

2. 在项目启动的时候，就去加载applicationContext.xml的配置文件，在web.xml中配置ContextLoaderListener监听器（该监听器只能加载WEB-INF目录下的applicationContext.xml的配置文件）。

```
<!-- 配置Spring的监听器 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>
    <!-- 配置加载类路径的配置文件 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
```

3. 在controller中注入service对象，调用service对象的方法进行测试

```
package cn.itcast.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import cn.itcast.service.AccountService;

@Controller
@RequestMapping("/account")
public class AccountController {

    @Autowired
    private AccountService accoutService;

    /**
     * 查询所有的数据
     * @return
     */
    @RequestMapping("/findAll")
    public String findAll() {
        System.out.println("表现层：查询所有账户...");
        accoutService.findAll();
        return "list";
    }

}
```

第四章：Spring整合MyBatis框架

1. 搭建和测试MyBatis的环境

1. 在web项目中编写SqlMapConfig.xml的配置文件，编写核心配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql:///ssm"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 使用的是注解 -->
    <mappers>
        <!-- <mapper class="cn.itcast.dao.AccountDao"/> -->
        <!-- 该包下所有的dao接口都可以使用 -->
        <package name="cn.itcast.dao"/>
    </mappers>
</configuration>
```

2. 在AccountDao接口的方法上添加注解，编写SQL语句

```
package cn.itcast.dao;

import java.util.List;

import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;

import cn.itcast.domain.Account;

public interface AccountDao {

    @Insert(value="insert into account (name,money) values (#{name},#{money})")
    public void saveAccount(Account account);

    @Select("select * from account")
    public List<Account> findAll();

}
```

3. 编写测试的方法

```
package cn.itcast.test;

import java.io.InputStream;
import java.util.List;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Test;

import cn.itcast.dao.AccountDao;
import cn.itcast.domain.Account;

public class Demo1 {

    @Test
    public void run1() throws Exception {
        // 加载配置文件
        InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        // 创建工厂
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
        // 创建sqlSession对象
        SqlSession session = factory.openSession();
        // 获取代理对象
        AccountDao dao = session.getMapper(AccountDao.class);
        // 调用查询的方法
        List<Account> list = dao.findAll();
        for (Account account : list) {
            System.out.println(account);
        }
        // 释放资源
        session.close();
        inputStream.close();
    }

    @Test
    public void run2() throws Exception {
        Account account = new Account();
        account.setName("熊大");
        account.setMoney(400d);

        // 加载配置文件
        InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        // 创建工厂
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
        // 创建sqlSession对象
        SqlSession session = factory.openSession();

        // 获取代理对象
        AccountDao dao = session.getMapper(AccountDao.class);
```

```
        dao.saveAccount(account);

        // 提交事务
        session.commit();
        // 释放资源
        session.close();
        inputStream.close();
    }

}
```

2. Spring整合MyBatis框架

1. 目的：把SqlMapConfig.xml配置文件中的内容配置到applicationContext.xml配置文件中

```
    <!-- 配置C3P0的连接池对象 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql:///ssm" />
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>

    <!-- 配置SqlSession的工厂 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- 配置扫描dao的包 -->
    <bean id="mapperScanner" class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="cn.itcast.dao"/>
    </bean>
```

2. 在AccountDao接口中添加@Repository注解
3. 在service中注入dao对象，进行测试
4. 代码如下

```
package cn.itcast.dao;

import java.util.List;

import org.apache.ibatis.annotations.Insert;

import org.apache.ibatis.annotations.Select;
```

```
import org.springframework.stereotype.Repository;

import cn.itcast.domain.Account;

@Repository
public interface AccountDao {

    @Insert(value="insert into account (name,money) values (#{name},#{money})")
    public void saveAccount(Account account);

    @Select("select * from account")
    public List<Account> findAll();

}
```

```
  package cn.itcast.service.impl;

   import java.util.List;

   import org.springframework.beans.factory.annotation.Autowired;

   import org.springframework.stereotype.Service;

   import cn.itcast.dao.AccountDao;

   import cn.itcast.domain.Account;

   import cn.itcast.service.AccountService;

   @Service("accountService")

   public class AccountServiceImpl implements AccountService {



    @Autowired

    private AccountDao accountDao;

    public void saveAccount(Account account) {

    }

    public List<Account> findAll() {

        System.out.println("业务层：查询所有账户...");

        return accountDao.findAll();

    }

   }
```

```
package cn.itcast.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import cn.itcast.domain.Account;

import cn.itcast.service.AccountService;

@Controller

@RequestMapping("/account")

public class AccountController {



 @Autowired

 private AccountService accoutService;



 /**

  *  查询所有的数据
      * @return
         */
         @RequestMapping("/findAll")
         public String findAll() {
             System.out.println("表现层：查询所有账户...");
             List<Account> list = accoutService.findAll();
             for (Account account : list) {
                 System.out.println(account);
             }
             return "list";
         }

}
```

5. 配置Spring的声明式事务管理

<tx:advice id="txAdvice" transaction-manager="transactionManager">

tx:attributes

<tx:method name="find*" read-only="true"/>

<tx:method name="*" isolation="DEFAULT"/>

/tx:attributes

/tx:advice

aop:config

<aop:advisor advice-ref="txAdvice" pointcut="execution(public * cn.itcast.service..ServiceImpl.*(..))"/>

/aop:config

```
6．测试保存帐户的方法
```

姓名：

金额：

保存

@RequestMapping("/saveAccount") public String saveAccount(Account account) {
accoutService.saveAccount(account); return "list"; }

```

```