

Author: unirithe

WeChat: unirithe

QQ: uni1024

Ctrl +F :

`toSpring` -> you can get to the text about Spring

`toSpringMVC` -> you can get to the text about SpringMVC

`toMyBatis` -> you can get to the text about MyBatis

toSpring

1. Spring开发步骤

2. Spring快速入门

2.1 在IDEA中创建并配置Maven项目

2.2 编写Dao层和Dao层实现代码

2.3 创建SPring配置文件

2.4 编写测试主类 UserDaoDemo.java

3. Spring 配置文件

3.1 Bean标签

3.2 Bean标签范围配置

3.2.1 测试 singleton 和 prototype的区别

3.2.2 总结

3.3 Bean生命周期配置

3.3.1 标签

3.3.2 测试

3.4 Bean 实例化的三种方式

3.4.1 工厂静态

3.4.2 工厂实例

3.5 依赖注入

3.6 Bean的依赖注入方式

3.6.1 构造方法

3.6.2 set方法

3.7 Bean依赖注入三大数据类型

3.7.1 普通+引用数据类型

3.7.2 集合数据类型Map+List+Prop

3.8 引入其他配置文件 (分模块开发)

3.9 配置总结

3.9.1 Spring重点配置

4. Spring 相关API

4.1 ApplicationContext的实现类

4.2 getBean ()

4.3 知识要点

5. Spring 配置数据源

5.1 数据源 / 连接池 的作用

5.2 数据源的开发步骤

5.3 手动创建

5.3.1 设置四个数据源的pom.xml

5.3.2 测试c3p0数据源

5.3.3 测试druid数据源

5.3.4 从配置文件中读取信息测试c3p0

5.4 Spring 配置数据源

5.5 Spring配置文件里加载properties文件

5.5.1 格式

5.5.2 框架

6. Spring 注解开发

6.1 Spring 原始注解

6.1.1 配置组件扫描

6.1.2 未用注解前的依赖注入

- 6.1.3 注解依赖注入
- 6.1.4 Value注解
- 6.1.5 初始化与销毁注解
- 6.2 Spring新注解
- 6.2.1 全注解样例

7. Spring 整合 Junit

- 7.1 原始JUnit 测试Spring存在的问题
- 7.2 解决思路
- 7.3 Spring集成JUnit步骤
- 7.4 配置xml方式测试
- 7.5 全注解方式测试
- 8. Spring集成web环境
- 8.1 环境搭建
 - 8.1.1 配置pom.xml
 - 8.1.2 DAO层和Service层
 - 8.1.3 配置Spring
 - 8.1.4 Servlet层
 - 8.1.5 配置Tomcat
 - 8.1.6 测试
- 8.2 ApplicationContext应用上下文获取方式
 - 8.2.1 编写监听类获取ApplicationContext对象
 - 8.2.2 配置web.xml
 - 8.2.3 Servlet层
- 8.3 Spring提供获取应用上下文的工具
- 8.4 Spring集成web环境步骤总结

9. Spring AOP

- 9.1 简介
- 9.2 AOP 作用及其优势
- 9.3 AOP 的底层实现原理介绍
- 9.4 AOP 动态代理技术
- 9.5 JDK的动态代理
- 9.6 cglib的动态代理
- 9.7 AOP的相关概念
 - * Target (目标对象)
 - * Proxy (代理)
 - * Joinpoint (连接点)
 - * Pointcut (切入点)
 - * Advice (通知 / 增强)
 - * Aspect (切面)
 - * Weaving (织入)
- 9.8 AOP 开发明确的事项
 - 9.8.1 需编写的内容
 - 9.8.2 AOP 技术实现的内容
 - 9.8.3 AOP 底层自动使用代理方式
- 9.9 知识要点

10. 基于 XML 的 AOP 开发

- 10.1 AOP开发六部曲
 - 10.1.1 配置pom.xml
 - 10.1.2 编写目标接口 和 目标类
 - 10.1.3 编写切面类
 - 10.1.4 配置applicationContext.xml
 - 10.1.5 编写测试类
- 10.2 XML配置 AOP 详解
 - 10.2.1 切点表达式的写法
 - 10.2.2 通知的类型
 - 10.2.3 切点表达式的抽取
- 10.3 知识要点

11. 基于注解的 AOP 开发

- 11.1 注解AOP开发六部曲
- 11.2 注解配置 AOP 详解

- 11.2.1 注解通知的类型
- 11.2.2 切点表达式的抽取
- 11.3 知识要点
 - 11.3.1 注解AOP开发三部曲
 - 11.3.2 通知注解类型

12. Spring 的事务控制

- 12.1 编程式事务控制相关对象
 - 12.1.1 接口PlatformTransactionManager
 - 12.1.2 对象TransactionDefinition
 - 1.事务隔离级别
 - 2.事务传播行为
 - 12.1.3 接口TransactionStatus
 - 12.1.4 知识要点
 - 1. 编程式事务控制三大对象
- 12.2 基于 XMI 的声明式事务控制
 - 12.2.1 声明式事务控制概念
 - 12.2.2 声明式事务控制作用
 - 12.2.3 声明式事务控制的实现
 - <tx:method>
 - 12.2.4 知识要点
 - 声明式事务控制的配置要点

12.3 基于注解的声明式事务控制

- 12.3.1 快速入门
- 12.3.2 注解配置的解析
- 12.3.3 知识要点，三部曲

toSpringMVC

- 1. 简介
- 2. SpringMVC 快速入门

- 2.1 开发步骤
- 2.2 代码实现

3. SpringMVC 流程演示

4. SpringMVC 组件解析

- 4.1 注解解析
 - 4.1.1 @RequestMapping
 - 4.1.2 扫描注解分类
- 4.2 XML配置
- 4.3 知识要点
 - 4.3.1 SpringMVC 相关组件
 - 4.3.2 SpringMVC的注解和配置

5 Spring MVC 数据响应

- 5.1 响应方式
- 5.2 页面跳转
 - 5.2.1 返回字符串形式
 - 5.2.2 返回 ModelAndView对象
- 5.3 回写数据
 - 5.3.1 直接返回字符串
 - 5.3.2 返回json类型
 - 5.3.3 返回对象和集合

6. SpringMVC 获得请求数据

- 6.1 获得请求参数
- 6.2 获得基本类型参数
- 6.3 获得POJO类型参数
- 6.4 获得数组类型参数
- 6.5 获得集合类型参数
- 6.6 POST请求数据乱码问题
- 6.7 参数绑定注解@RequestParam
- 6.8 获得 Restful 风格的参数@PathVariable
- 6.9 自定义类型转换器
- 6.10 获得 Servlet 相关API
- 6.11 获得请求头

6.11.1 @RequestHeader

6.11.2 @CookieValue

6.12 文件上传

6.12.1 上传三要素

6.12.2 上传原理

6.13 单文件上传

6.14 多文件上传

6.14.1 方式一：逐一处理

6.14.2 方式二：数组处理

6.15 SpringMVC 获得请求数据 知识要点

7. SpringMVC 拦截器

7.1 拦截器 (interceptor) 的作用

7.2 拦截器和过滤器的区别

7.3 拦截器三部曲

7.4 拦截器三大方法

8. SpringMVC 异常处理

8.1 异常处理的思路

8.2 异常处理的两种方式

8.3 简单异常处理器 SimpleMappingExceptionResolver

8.4 自定义异常四部曲

toMyBatis

1. 前言

1.1 原始 JDBC操作

1.1.1 查询数据

1.1.2 插入数据

1.2 原始JDBC操作分析

1.2.1 存在的问题

1.2.2 针对上述问题的解决方案

2. Mybatis 简介

3. Mybatis开发六部曲 (以查询为例)

3.1. 添加Mybatis的坐标

3.2. 创建user数据表

3.3. 编写User实体类

3.4. 编写映射文件

3.5. 编写核心文件

3.6. 编写测试类

4. MyBatis 映射文件概述

5. MyBatis 增删改操作

5.1 插入操作

5.2 修改操作

5.3 删除操作

6. MyBatis 核心配置文件概述

6.1 层级关系

6.2 MyBatis 常用配置解析

6.2.1 environments 标签

6.2.2 Mappers 标签

6.2.3 Properties 标签

6.2.4 typeAliases 标签

6.2.5 知识小结

7. MyBatis 响应API

7.1 SqlSession 工厂构建器-FactoryBuilder

7.2 SqlSession工厂对象 -Factory

7.3 SqlSession 会话对象

8. MyBatis的DAO层实现

8.1 传统接口开发方式

8.2 代理开发方式

Mapper接口开发规范

Demo：根据 ID 查询信息

Demo : 确定用户名和密码是否匹配

9. MyBatis 映射文件 深入

- 9.1 动态 SQL 语句
 - 9.1.1 < if > 标签
 - 9.1.2 < foreach > 标签
- 9.2 SQL 片断抽取
- 9.3 MyBatis 映射文件配置常用标签
- 9.4 typeHandlers 标签
 - 9.4.1 开发四部曲
 - 9.4.2 测试Demo 实现用户生日信息的类型转换
- 9.5 plugins 标签 测试第三方分页助手插件
- 9.6 知识小结

10. MyBatis 多表操作

- 10.1 一对一 Demo
- 10.2 一对多 Demo
- 10.3 多对多 Demo
- 10.4 知识小结

11. MyBatis 的注解开发

- 11.1 MyBatis 注解实现CURD Demo
- 11.2 注解实现复杂映射开发
 - 11.2.1 一对一 Demo
 - 11.2.2 一对多 Demo
 - 11.2.3 多对多 Demo

toSpring

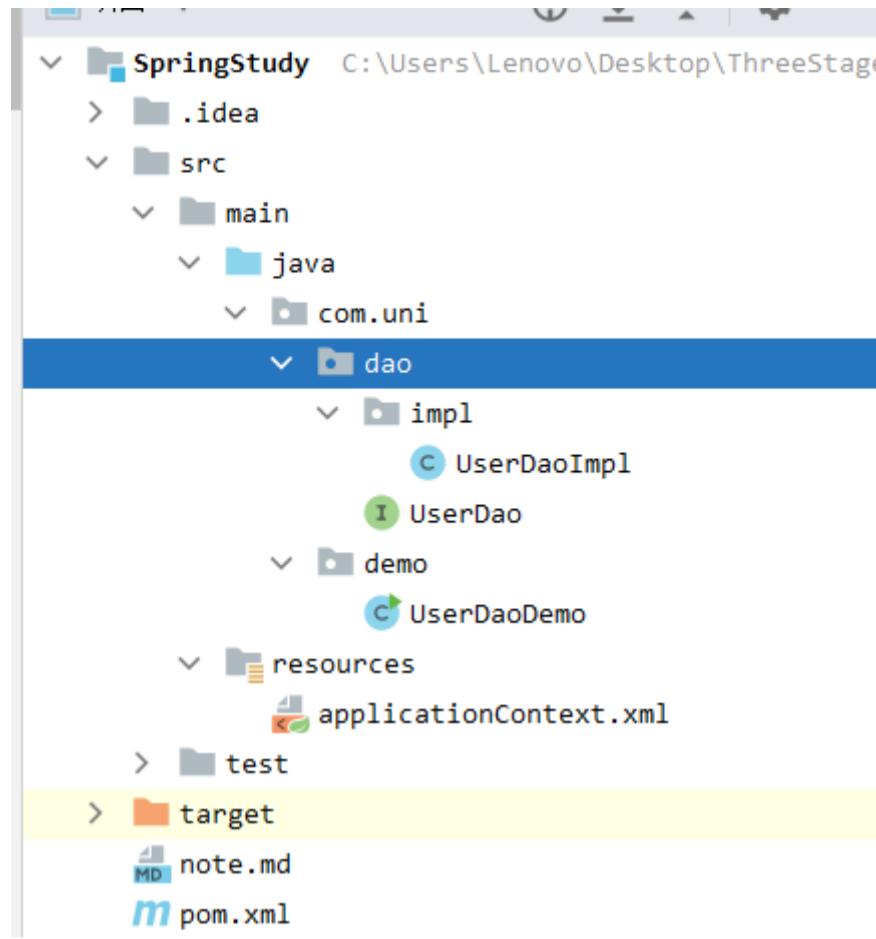
1. Spring 开发步骤

1. 导入坐标
2. 创建Bean
3. 创建applicationContext.xml 配置文件
4. 在配置文件中进行配置
5. 创建ApplicationContext对象getBean

2. Spring 快速入门

开发环境: windows10 + IDEA 2021.1.3 (Ultimate Edition) + Maven + Spring

最终项目结构图:



2.1 在IDEA中创建并配置Maven项目

pom.xml 配置内容:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringStudy</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.5.RELEASE</version>
    </dependency>
</dependencies>
</project>
```

配置成功后点击Maven图标导入依赖包

2.2 编写Dao层和Dao层实现代码

(1) 编写Dao层接口 UserDao.java

```
package com.uni.dao;

public interface UserDao {
    public void save();
}
```

(2) 编写Dao层接口实现 UserDaoImpl.java

```
package com.uni.dao.impl;

import com.uni.dao.UserDao;

public class UserDaoImpl implements UserDao {

    @Override
    public void save() {
        System.out.println("save runing...");
    }
}
```

2.3 创建Spring配置文件

在 resources 文件夹里创建 applicationContext.xml 文件 (文件名可自定义, 标记为AC)

在 beans 和 /beans 之间加入

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
```

这样一来在Service层就无需在创建Dao层对象, 而可以通过该配置文件来获取对应的对象

2.4 编写测试主类 UserDaoDemo.java

```
package com.uni.demo;

import com.uni.dao.UserDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserDaoDemo {
    public static void main(String[] args) {
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao = (UserDao) app.getBean("userDao");
        userDao.save();
    }
}
```

如上述代码, 根据一个xml配置文件创建了ApplicationContext对象, 然后通过该对象可以直接getBean, 即获取Dao层的实现类。

最终的执行结果为:

```
save runing...
```

3. Spring 配置文件

3.1 Bean标签

- id : Bean实例在Spring容器的唯一标识 (不可重复)
- class: Bean的全限定名称 (Dao层的实现类)

可在Service层通过 ApplicationContext.getBean() 方法获取

默认情况调用的是 无参构造函数 , 所以实现类中必须有无参构造

如:

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
```

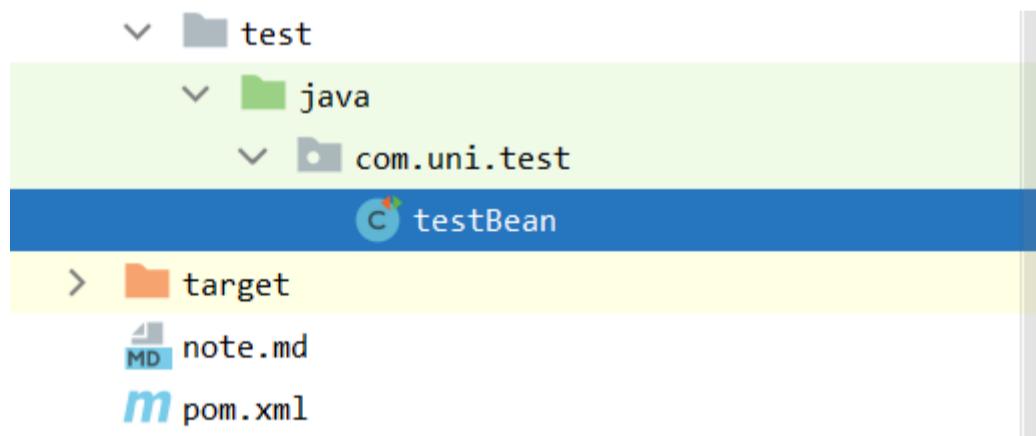
```
package com.uni.dao.impl;
import com.uni.dao.UserDao;
public class UserDaoImpl implements UserDao {
    @Override
    public void save() {
        System.out.println("save runing...");
    }
}
```

3.2 Bean标签范围配置

scope属性: 指对象的作用范围

取值范围	说明
singleton	默认值, 单例的
prototype	多例的
request	WEB项目里, Spring创建一个Bean对象, 并将其存入request域中
session	WEB项目中, Spring创建一个Bean对象, 并将其存入session域中
global session	WEB项目中, 应用在Portlet环境, 如果没有该环境, 则相当于session

3.2.1 测试 singleton 和 prototype的区别



1. 在pom.xml 里加入junit (Java测试框架) 依赖包

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
</dependency>
```

2. getBean创建相同id的Dao类对象, 输出地址观察区别

testBean.java

```
package com.uni.test;
import com.uni.dao.UserDao;
import org.junit.Test;
```

```

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

public class testBean {
    @Test
    // 测试Scope属性
    public void test1(){
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao1 = (UserDao) app.getBean("userDao");
        UserDao userDao2 = (UserDao) app.getBean("userDao");
        System.out.println(userDao1);
        System.out.println(userDao2);
    }
}

```

3. case1: scope 为 singleton

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl" scope="singleton"></bean>
```

结果为

com.uni.dao.impl.UserDaoImpl@1c3a4799
com.uni.dao.impl.UserDaoImpl@1c3a4799

地址相同，为相同的UserDao

case 2: scope 为 prototype

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl" scope="prototype"></bean>
```

结果为

com.uni.dao.impl.UserDaoImpl@67205a84
com.uni.dao.impl.UserDaoImpl@7d0587f1

地址不同，为不同的UserDao

3.2.2 总结

1. scope取值为singleton

- (1) Bean实例化个数: 1 (单例模式)
- (2) Bean实例化时机: 当Spring核心文件被加载时, 实例化配置Bean的实例
- (3) Bean生命周期:

- 对象创建: 当应用加载, 创建容器时, 对象被创建
- 对象运行: 只要容器在, 对象就存在
- 对象销毁: 当应用卸载, 销毁容器时, 对象被销毁

2. scope取值为prototype

- (1) Bean实例化个数: 多个 (多例模式)
- (2) Bean 实例化时机: 当调用getBean () 方法时实例化Bean
 - 对象创建: 当使用对象时, 创建新的对象实例
 - 对象运行: 只要对象在使用, 就一直存在
 - 对象销毁: 当对象长时间空闲, 就会被Java的垃圾回收机制回收

3.3 Bean生命周期配置

3.3.1 标签

- init-method: 指定类中的初始化方法名称
- destroy-method: 指定类中销毁方法名称

3.3.2 测试

applicationContext.xml

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl" init-method="init" destroy-method="destroy"></bean>
```

UserDaoImpl.java

```
package com.uni.dao.impl;
import com.uni.dao.UserDao;
public class UserDaoImpl implements UserDao {
    public UserDaoImpl(){
        System.out.println("UserDaoImpl创建完毕.");
    }
    @Override
    public void save() {
        System.out.println("save runing...");
    }
    public void init(){
        System.out.println("初始化.");
    }
    public void destroy(){
        System.out.println("销毁.");
    }
}
```

testBean.java

```
package com.uni.test;
import com.uni.dao.UserDao;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class testBean {
    @Test
    // 测试Scope属性
    public void test1(){
        ClassPathXmlApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao = (UserDao) app.getBean("userDao");
        userDao.save();
        app.close();
    }
}
```

测试结果

```
UserDaoImpl创建完毕。  
初始化。  
save runing...  
七月 26, 2021 3:48:17 下午 org.springframework.context.support.AbstractApplicationContext doClose  
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@198e2867: startup date [Mon Jul 26  
15:48:16 CST 2021]; root of context hierarchy  
销毁。  
  
进程已结束, 退出代码为 0
```

通过测试可得出，Bean的无参构造比init-method指定的方法更先执行

3.4 Bean 实例化的三种方式

- 无参构造方法实例化
- 工厂静态方法实例化
- 工厂实例方法实例化

之前都是引用无参构造方法，略过

3.4.1 工厂静态

StaticFactory.java

```
package com.uni.factory;  
import com.uni.dao.UserDao;  
import com.uni.dao.impl.UserDaoImpl;  
  
public class StaticFactory {  
    public static UserDao getUserDao(){  
        return new UserDaoImpl();  
    }  
}
```

applicationContext.xml

```
<bean id = "userDao" class = "com.uni.factory.StaticFactory" factory-method="getUserDao"></bean>
```

testBean.java

```
package com.uni.test;  
import com.uni.dao.UserDao;  
import org.junit.Test;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class testBean {  
    @Test  
    public void test1(){  
        ClassPathXmlApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");  
        UserDao userDao = (UserDao) app.getBean("userDao");  
        userDao.save();  
    }  
}
```

3.4.2 工厂实例

applicationContext.xml

```
<bean id = "factory" class = "com.uni.factory.DynamicFactory"></bean>          <bean id = "UserDao" factory-
bean="factory" factory-method="getUserDao"></bean>
```

DynamicFactory.java

```
package com.uni.factory;
import com.uni.dao.UserDao;
import com.uni.dao.impl.UserDaoImpl;

public class DynamicFactory {
    public UserDao getUserDao(){
        return new UserDaoImpl();
    }
}
```

3.5 依赖注入

依赖注入Dependency injection : 是Spring框架核心IOC的具体实现

在编写程序时，通过控制反转，把对象的创建交给了Spring，但代码中不可能出现没有依赖的情况。

IOC解耦知识降低他们之间的依赖关系，但不会消除。

例如：业务层仍会调用持久层的方法

简而言之就是等框架把持久层对象传入业务层，而不用手动获取

3.6 Bean的依赖注入方式

将UserDao 注入到UserService内部

3.6.1 构造方法

applicationContext.xml

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
<bean id = "userService" class = "com.uni.service.impl.UserServiceImpl">
    <constructor-arg name = "userDao" ref = "userDao"></constructor-arg>
</bean>
```

UserServiceImpl.java

```
package com.uni.service.impl;
import com.uni.dao.UserDao;
import com.uni.service.UserService;

public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public UserServiceImpl(UserDao userDao){
        this.userDao = userDao;
    }
    public UserServiceImpl() {
    }
    @Override
    public void save() {
        userDao.save();
    }
}
```

```
}
```

3.6.2 set方法

UserController.java

```
package com.uni.demo;
import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserController {
    public static void main(String[] args) {
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) app.getBean("userService");
        userService.save();
    }
}
```

applicationContext.xml

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
<bean id = "userService" class = "com.uni.service.impl.UserServiceImpl">
    <!-- name 为 set方法后的内容，第一个字母改为小写 -->
    <property name="userDao" ref="userDao"></property>
</bean>
```

UserServiceImpl.java

```
package com.uni.service.impl;
import com.uni.dao.UserDao;
import com.uni.service.UserService;

public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public void setUserDao(UserDao userDao){
        this.userDao = userDao;
    }
    @Override
    public void save() {
        userDao.save();
    }
}
```

简写形式：

applicationContext.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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
    <bean id = "userService" class = "com.uni.service.impl.UserServiceImpl" p:userDao-ref="userDao"/>
</beans>

```

3.7 Bean依赖注入三大数据类型

除了Bean对象以外，普通数据类型、集合等都可以在容器中进行注入

- 普通数据类型
- 引用数据类型
- 集合数据类型

3.7.1 普通+引用数据类型

通过Spring框架从DAO层注入到Service层（Bean采用构造方法注入）

applicationContext.xml

```

<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl">
    <property name = "username" value = "zhangsan"/>
    <property name="age" value = "21"/>
</bean>
<bean id = "userService" class = "com.uni.service.impl.UserServiceImpl">
    <constructor-arg name = "userDao" ref = "userDao"></constructor-arg>
</bean>

```

userDaoImpl.java

```

package com.uni.dao.impl;
import com.uni.dao.UserDao;
public class UserDaoImpl implements UserDao {
    private String username;
    private int age;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public void save() {
        System.out.println(username + ", " + age);
    }
}

```

```
}
```

UserServiceImpl.java

```
package com.uni.service.impl;
import com.uni.dao.UserDao;
import com.uni.service.UserService;

public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public UserServiceImpl(UserDao userDao){
        this.userDao = userDao;
    }
    public UserServiceImpl() { }
    @Override
    public void save() {
        userDao.save();
    }
}
```

测试类UserController.java

```
package com.uni.demo;
import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserController {
    public static void main(String[] args) {
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService) app.getBean("userService");
        userService.save();
    }
}
```

3.7.2 集合数据类型Map+List+Prop

applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl">
        <property name="strList">
            <list>
                <value>a</value>
                <value>b</value>
                <value>c</value>
            </list>
        </property>
        <property name="userMap">
            <map>
                <entry key = "u1" value-ref="user1"></entry>
                <entry key = "u2" value-ref="user2"></entry>
            </map>
        </property>
    
```

```

<property name="properties">
    <props>
        <prop key = "p1">ppp1</prop>
        <prop key = "p2">ppp2</prop>
        <prop key = "p3">ppp3</prop>
    </props>
</property>
</bean>

<bean id = "user1" class = "com.uni.domain.User">
    <property name="name" value = "xiaoming"/>
    <property name="addr" value = "wenzhou"/>
</bean>
<bean id = "user2" class = "com.uni.domain.User">
    <property name="name" value = "zhangsan"/>
    <property name="addr" value = "beijing"/>
</bean>

<bean id = "userService" class = "com.uni.service.impl.UserServiceImpl">
    <constructor-arg name = "userDao" ref = "userDao"></constructor-arg>
</bean>

</beans>

```

UserDaoImpl.java

```

package com.uni.dao.impl;
import com.uni.dao.UserDao;
import com.uni.domain.User;

import java.util.List;
import java.util.Map;
import java.util.Properties;

public class UserDaoImpl implements UserDao {
    private List<String> strList;
    private Map<String, User> userMap;
    private Properties properties;

    public List<String> getStrList() {
        return strList;
    }

    public void setStrList(List<String> strList) {
        this.strList = strList;
    }

    public Map<String, User> getUserMap() {
        return userMap;
    }

    public void setUserMap(Map<String, User> userMap) {
        this.userMap = userMap;
    }

    public Properties getProperties() {
        return properties;
    }
}

```

```

public void setProperties(Properties properties) {
    this.properties = properties;
}

@Override
public void save() {
    System.out.println(strList);
    System.out.println(userMap);
    System.out.println(properties);
    System.out.println("save running...");
}
}

```

3.8 引入其他配置文件（分模块开发）

```
<import resource="applicationContext-xxx.xml"/>
```

3.9 配置总结

3.9.1 Spring重点配置

<bean> 标签

id 属性：在容器中Bean实例的唯一标识，不能重复
class 属性：要实例化的Bean的全限定名
scope 属性：Bean的作用范围，常用是Singleton（默认）和 prototype

<property> 标签：属性注入

name 属性：属性名称
value 属性：注入的普通属性的值
ref 属性：注入的对象引用至

<list> 标签

<map> 标签

<properties> 标签

<constructor-arg> 标签

<import> 标签：导入其他的Spring分文件（xml）

4. Spring 相关API

4.1 ApplicationContext的实现类

(1) ClassPathXmlApplicationContext

从类的根路径 /src/resources 下加载配置文件（之前的例子都用的这个）

(2) FileSystemXmlApplicationContext

从磁盘路径上加载配置文件（很少使用）

(3) AnnotationConfigApplicationContext

当使用注解配置容器对象时，用此类创建spring容器，可以读取注解

4.2 getBean ()

源码

```
public Object getBean(String name) throws BeansException {
    this.assertBeanFactoryActive();
    return this.getBeanFactory().getBean(name);
}

public <T> T getBean(Class<T> requiredType) throws BeansException {
    this.assertBeanFactoryActive();
    return this.getBeanFactory().getBean(requiredType);
}
```

原先

```
UserService userService = (UserService) app.getBean("userService");
```

可改写为

```
userService userService = (UserService) app.getBean(UserService.class)
```

4.3 知识要点

Spring 重点API

```
ApplicationContext app = new ClasspathXmlApplicationContext("xml文件");
app.getBean("id");
app.getBean(Class);
```

5. Spring 配置数据源

5.1 数据源 / 连接池 的作用

- 提高程序性能
- 实现实例化数据源，初始化部分连接资源
- 使用连接资源时从数据源中获取
- 使用完毕后将连接资源归还给数据源

常见的数据源（连接池）：DBCP、C3P0、BoneCP、Druid等

5.2 数据源的开发步骤

- 导入数据源的坐标和数据库驱动坐标
- 创建数据源对象
- 设置数据源的基本连接数据
- 使用数据源获取连接资源和归还连接资源

5.3 手动创建

创建Maven项目时记得选择模板 org.apache.maven.archetypes:maven-archetype-quickstart

5.3.1 设置四个数据源的pom.xml

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringStudy_ioc_annot</artifactId>
<version>1.0-SNAPSHOT</version>

<name>SpringStudy_ioc_annot</name>
<!-- FIXME change it to the project's website -->
<url>http://www.example.com</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.32</version>
  </dependency>

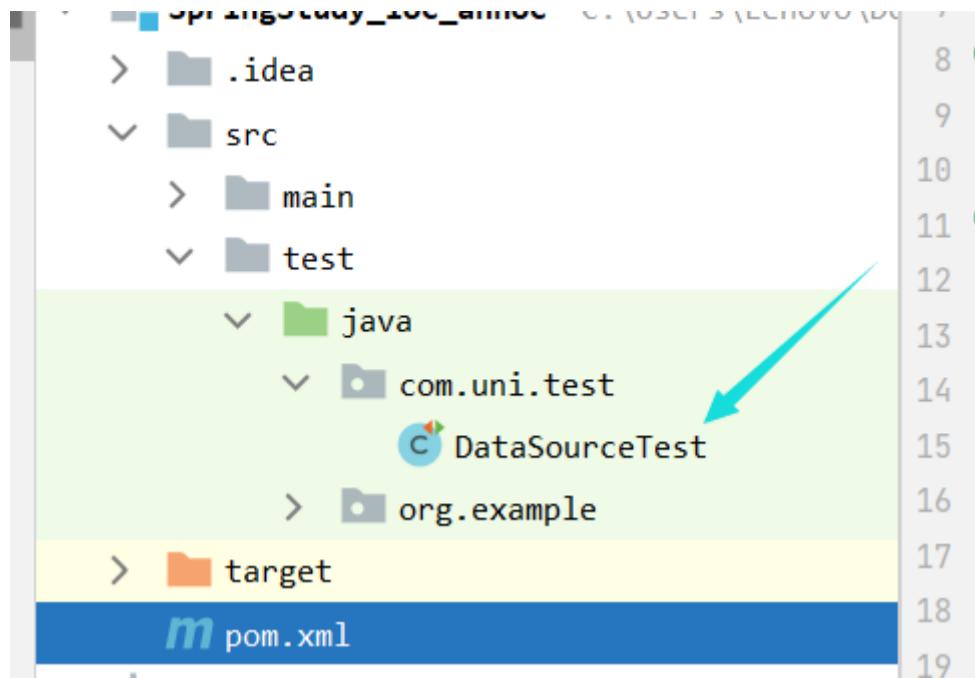
  <dependency>
    <groupId>c3p0</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.1.2</version>
  </dependency>

  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.10</version>
  </dependency>
</dependencies>

</project>

```

直接在test文件夹里借助junit框架创建DataSourceTest.java程序进行测试



5.3.2 测试c3p0数据源

DataSourceTest.java

```
package com.uni.test;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.junit.Test;

import java.sql.Connection;

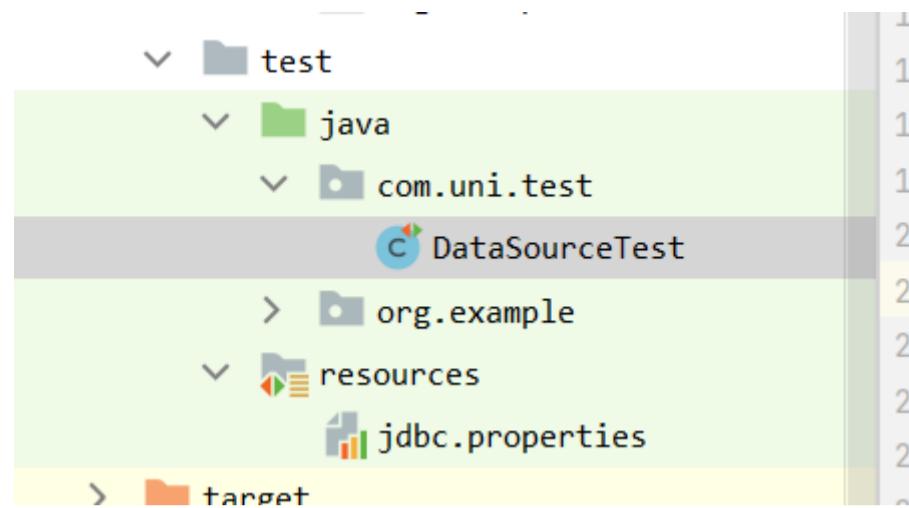
public class DataSourceTest {
    @Test
    // 测试手动创建c3p0 数据源
    public void test1() throws Exception{
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=UTF-8");
        dataSource.setUser("root");
        dataSource.setPassword("");
        Connection connection = dataSource.getConnection();
        System.out.println(connection);
        connection.close();
    }
}
```

5.3.3 测试druid数据源

DataSourceTest.java 部分

```
@Test
// 测试手动创建 druid 数据源
public void test2() throws Exception{
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/test");
    dataSource.setUsername("root");
    dataSource.setPassword("");
    DruidPooledConnection connection = dataSource.getConnection();
    System.out.println(connection);
}
```

5.3.4 从配置文件中读取信息测试c3p0



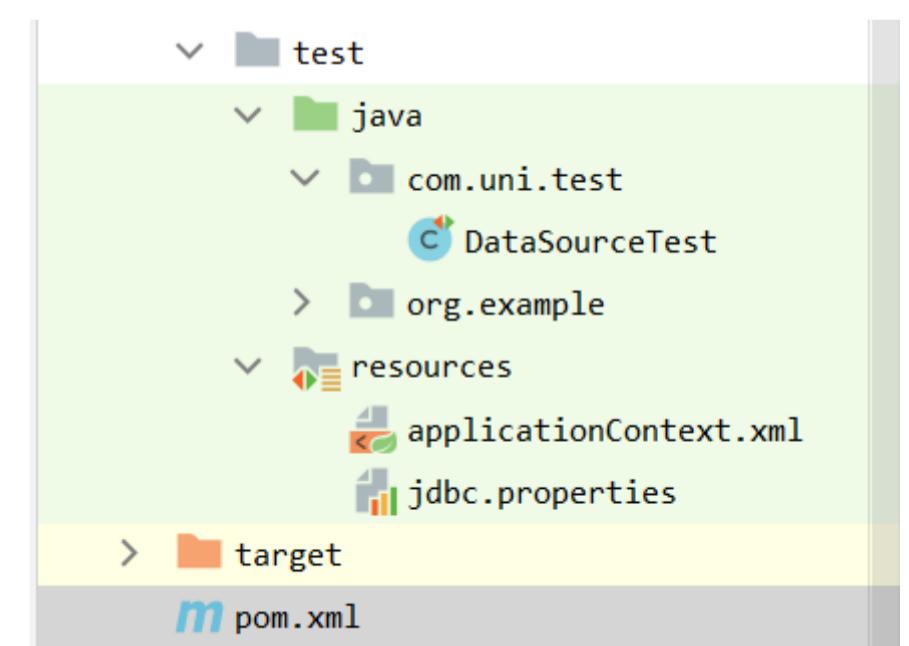
jdbc.properties

```
jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/test  
jdbc.username=root  
jdbc.password=
```

DataSourceTest.java 关键代码

```
@Test  
// 测试手动创建c3p0 数据源 (加载配置文件)  
public void test3() throws Exception{  
    // 读配置文件  
    ResourceBundle rb = ResourceBundle.getBundle("jdbc"); // 无需后缀名properties  
    String driver = rb.getString("jdbc.driver");  
    String url = rb.getString("jdbc.url");  
    String username = rb.getString("jdbc.username");  
    String password = rb.getString("jdbc.password");  
    // 创建数据源对修昂  
    ComboPooledDataSource dataSource = new ComboPooledDataSource();  
    dataSource.setDriverClass(driver);  
    dataSource.setJdbcUrl(url);  
    dataSource.setUser(username);  
    dataSource.setPassword(password);  
    // 获取接口  
    Connection connection = dataSource.getConnection();  
    System.out.println(connection);  
}
```

5.4 Spring 配置数据源



pom.xml 添加Spring框架依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>
```

创建Spring配置文件applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id = "dataSource" class ="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value = "com.mysql.jdbc.Driver"/>
        <property name="jdbcUrl" value = "jdbc:mysql://localhost:3306/test"/>
        <property name="user" value = "root"/>
        <property name="password" value = ""/>
    </bean>
</beans>
```

测试代码DataSourceTest.java

```
@Test
// 测试Spring容器产生数据源对象
public void test4() throws Exception{
    ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
    DataSource dataSource = app.getBean(DataSource.class);
    Connection connection = dataSource.getConnection();
    System.out.println(connection);
    connection.close();
}
```

5.5 Spring配置文件里加载properties文件

5.5.1 格式

```
<context:property-placeholder location="xx.properties"/><property name = "" value = "${key}" />
```

5.5.2 框架

applicationContext.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"
       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">

    <!-- 加载外部的properties文件 -->
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id = "dataSource" class ="com.mchange.v2.c3p0.ComboPooledDataSource">
```

```

<property name="driverClass" value = "${jdbc.driver}"/>
<property name="jdbcUrl" value = "${jdbc.url}"/>
<property name="user" value = "${jdbc.username}"/>
<property name="password" value = "${jdbc.password}"/>
</bean>
</beans>

```

比较之前的xml它加入了两行：

(1)在beans标签内 `xmlns:context="http://www.springframework.org/schema/context"`

(2) 在xsi:schemaLocation内 `http://www.springframework.org/schema/context`

`http://www.springframework.org/schema/context/spring-context.xsd`

6. Spring 注解开发

6.1 Spring 原始注解

Spring是轻代码重配置的框架，配置较为繁重，影响开发效率，所以注解开发是一种趋势，注解代替XML配置文件可以简化配置，提高开发效率。

Spring原始注解 主要是替代 Bean 的配置

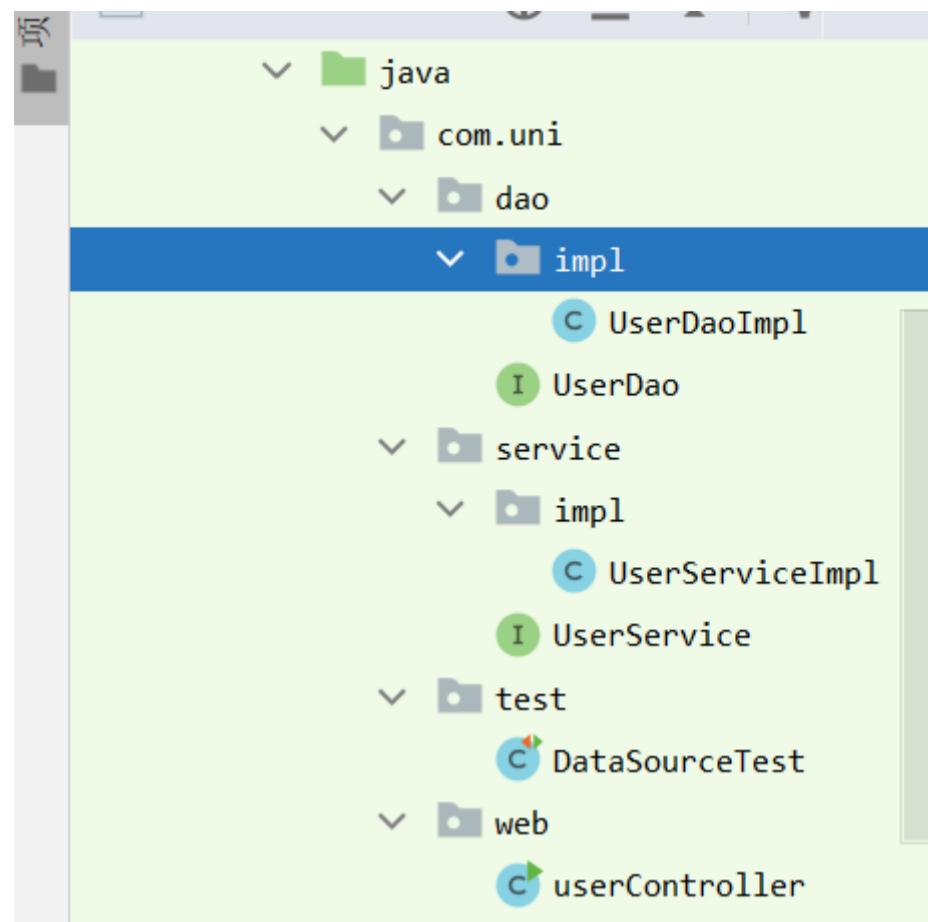
注解	说明
@Component	使用在类上用于实例化Bean
@Controller	使用在Web层类上用于实例化Bean
@Service	使用在service层上用于实例化Bean
@Repository	使用在dao层上用于实例化Bean
@Qualifier	结合@Autowired一起使用，用于根据名称进行依赖注入
@Resource	相当于@Autowried+@Qualifier，按照名称进行注入
@Value	注入普通属性
@Scope	标注Bean的作用范围
@PostConstruct	使用在方法上标注该方法是Bean的初始化方法
@PreDestroy	使用在方法上标注该方法是Bean的销毁方法

6.1.1 配置组件扫描

使用注解开发，需在applicationContext.xml 中配置组件扫描，作用是指定哪个包及其子包下的Bean需要进行扫描以便识别使用注解配置的类、字段和方法

```
<!-- 注解的组件扫描 --><context:component-scan base-package="com.uni" />
```

6.1.2 未用注解前的依赖注入



其中test文件夹无需创建（是上次测试的文件）

applicationContext.xml

```
<bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean>
<bean id = "userService" class ="com.uni.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"></property>
</bean>
```

UserDao.java

```
package com.uni.dao;

public interface UserDao {
    public void save();
}
```

UserDaoImpl.java

```
package com.uni.dao.impl;

import com.uni.dao.UserDao;

public class UserDaoImpl implements UserDao {
    @Override
    public void save() {
        System.out.println("save running...");
    }
}
```

UserService.java

```
package com.uni.service;

public interface UserService {
    public void save();
}
```

UserServiceImpl.java

```
package com.uni.service.impl;

import com.uni.dao.UserDao;
import com.uni.service.UserService;
```

```

public class UserServiceImpl implements UserService {
    private UserDao userDao;
    public void setUserDao(UserDao userDao){
        this.userDao = userDao;
    }
    @Override
    public void save() {
        userDao.save();
    }
}

```

模拟web层 userController.java

```

package com.uni.web;

import com.uni.service.UserService;
import com.uni.service.impl.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserController {
    public static void main(String[] args) {
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}

```

6.1.3 注解依赖注入

applicationContext.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"
       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">

    <!-- 配置组件扫描 -->
    <context:component-scan base-package="com.uni"/>
</beans>

```

UserDaoImpl.java

```

package com.uni.dao.impl;

import com.uni.dao.UserDao;
import org.springframework.stereotype.Component;

/* <bean id = "userDao" class = "com.uni.dao.impl.UserDaoImpl"></bean> */
@Repository("userDao")
public class UserDaoImpl implements UserDao {
    @Override
    public void save() {
        System.out.println("save running...");
    }
}

```

UserServiceImpl.java

```
package com.uni.service.impl;

import com.uni.dao.UserDao;
import com.uni.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

/* <bean id = "userService" class ="com.uni.service.impl.UserServiceImpl"> */
@Service("userService")
public class UserServiceImpl implements UserService {
    /* <property name="userDao" ref="userDao"></property> */
    @Autowired
    @Qualifier("userDao")
    private UserDao userDao;
    @Override
    public void save() {
        userDao.save();
    }
}
```

在注入对象时

如果只有@Autowire, 它代表按照数据类型从Spring容器中进行匹配, 如果只有单个UserDao对象, 可以省略@Qualifier。

@Qualifier 是按照id值从容器中进行匹配, 它必须结合@Autowire使用

二者可以结合成

```
@Resource(name = "userDao")
```

6.1.4 Value注解

value注解可以注入普通属性如:

```
@Value("xiaoming")private String name;
```

通过上述代码可以看出, 可以直接写成

```
private String name = "xiaoming";
```

所以@Value的真正用途并不是直接赋一个具体值, 而是在Spring容器中, 通过key获取value的值。

如:

```
@Value("${jdbc.driver}")
private String driver
```

applicationContext.xml

```
<context:property-placeholder location="classpath:jdbc.properties"/>
```

jdbc.properties

```
jdbc.driver=com.mysql.jdbc.Driver
```

6.1.5 初始化与销毁注解

@PostConstruct 初始化

@PreDestroy 销毁

UserServiceImpl.java

```
package com.uni.service.impl;

import com.uni.dao.UserDao;
import com.uni.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;

@Service("userService")
public class UserServiceImpl implements UserService {

    @Resource(name = "userDao")
    private UserDao userDao;

    @Override
    public void save() {
        userDao.save();
    }

    @PostConstruct
    public void init(){
        System.out.println("init");
    }

    @PreDestroy
    public void destroy(){
        System.out.println("destroy");
    }
}
```

userController.java

```
package com.uni.web;

import com.uni.service.UserService;
import com.uni.service.impl.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class userController {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = app.getBean(UserService.class);
        userService.save();
        app.close();
    }
}
```

运行结果

```
init  
save running...  
destroy
```

6.2 Spring新注解

使用原始注解不能全部替代xml配置文件，比如

- 非自定义的Bean的配置：bean
- 加载properties文件的配置：context:property-placeholder
- 组件扫描的配置：context:component-scan
- 引入其他文件：import

注解	说明
@Configuration	用于指定当前类是一个Spring配置类，当创建容器时会从该类上加载注解
@ComponentScan	用于指定Spring在初始化容器时要扫描的包
@Bean	用于标注方法的返回值存储到Spring容器中
@PropertySource	用于加载.properties文件中的配置
@Import	用于导入其他配置类

6.2.1 全注解样例

创建com.uni.config 包 用于配置Spring

它包含两个文件 SpringConfiguration.java 和DataSourceConfiguration.java

DataSourceConfiguration.java

```
package com.uni.config;  
  
import com.mchange.v2.c3p0.ComboPooledDataSource;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.PropertySource;  
import javax.sql.DataSource;  
// <context:property-placeholder location="classpath:jdbc.properties"/>  
@PropertySource("classpath:jdbc.properties")  
public class DataSourceConfiguration {  
    @Value("${jdbc.driver}")  
    private String driver;  
    @Value("${jdbc.url}")  
    private String url;  
    @Value("${jdbc.username}")  
    private String username;  
    @Value("${jdbc.password}")  
    private String password;  
  
    @Bean("dataSource") // Spring 会将当前方法的返回值 以 指定名称 存储到Spring容器中  
    public DataSource getDataSource () throws Exception{  
        ComboPooledDataSource dataSource = new ComboPooledDataSource();  
        dataSource.setDriverClass(driver);  
        dataSource.setJdbcUrl(url);  
        dataSource.setUser(username);  
        dataSource.setPassword(password);  
        return dataSource;  
    }  
}
```

SpringConfiguration.java

```
package com.uni.config;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;

import javax.sql.DataSource;
import java.sql.Connection;

// 标志该类是Spring的核心配置类
@Configuration
// <context:component-scan base-package="com.uni"/>
@ComponentScan("com.uni")
// <import resource="/">
@Import({DataSourceConfiguration.class})
public class SpringConfiguration {
}
```

测试类 userController.java

```
package com.uni.web;

import com.uni.config.SpringConfiguration;
import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class UserController {

    public static void main(String[] args) {
        //ClassPathXmlApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        ApplicationContext app = new AnnotationConfigApplicationContext(SpringConfiguration.class);
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}
```

7. Spring 整合 Junit

7.1 原始Junit 测试Spring存在的问题

在测试类中，每个测试方法都有以下两行代码

```
ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
DataSource dataSource = app.getBean(DataSource.class);
```

这两行代码为获取容器，不写的话会显示空指针异常

7.2 解决思路

- 让SpringJunit 负责创建Spring容器，但需将配置文件的名称告诉它
- 将需要进行测试的Bean直接注入在测试类中

7.3 Spring集成Junit步骤

1. 导入Spring集成 Junit 的坐标
2. 使用@Runwith注解替换原来的运行期
3. 使用@ContextConfiguration指定配置文件或配置类

4. 使用@Autowired注入需要测试的对象
5. 创建测试方法进行测试

7.4 配置xml方式测试

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.5.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.0.5.RELEASE</version>
    </dependency>
</dependencies>
```

SpringJunitTest.java

```
package com.uni.test;

import com.uni.service.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringJunitTest {
    @Autowired
    private UserService userService;
    @Test
    public void test1(){
        userService.save();
    }
}
```

7.5 全注解方式测试

SpringJunitTest.java

```
package com.uni.test;

import com.uni.config.SpringConfiguration;
import com.uni.service.UserService;
import org.junit.Test;
```

```

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

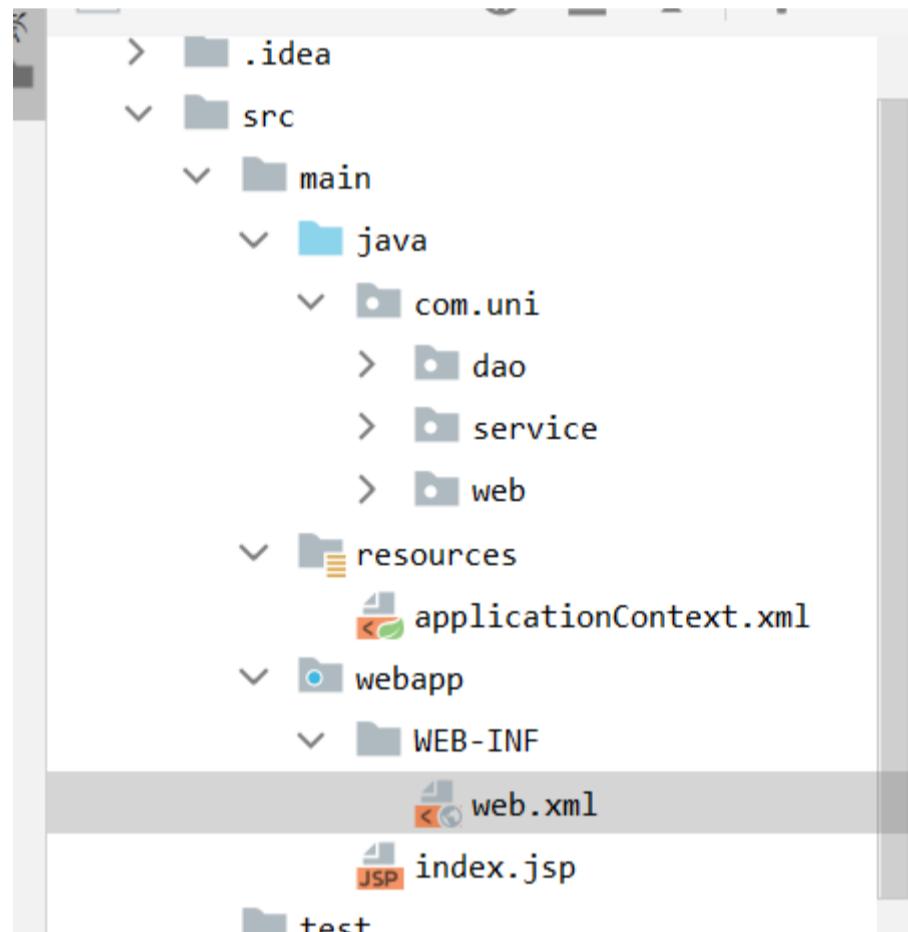
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringConfiguration.class})
public class SpringJunitTest {
    @Autowired
    private UserService userService;
    @Test
    public void test1(){
        userService.save();
    }
}

```

8. Spring集成web环境

8.1 环境搭建

项目结构如图



8.1.1 配置pom.xml

新建Java-Web项目 配置pom.xml

```

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>SpringStudy_mvc</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>SpringStudy_mvc Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

```

```

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
    </dependency>

    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.2.1</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.5.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.0.5.RELEASE</version>
    </dependency>
</dependencies>

</project>

```

8.1.2 DAO层和Service层

DAO层 userDao.java

```

package com.uni.dao;

public interface UserDao {
    public void save();
}

```

DAO层 userDaoImpl.java

```

package com.uni.dao.impl;

import com.uni.dao.UserDao;

public class UserDaoimpl implements UserDao {

    @Override
    public void save() {
        System.out.println("save running...");
    }
}

```

Service层userService.java

```

package com.uni.service;

public interface UserService {
    public void save();
}

```

Service层userServiceImpl.java

```

package com.uni.service.impl;

import com.uni.dao.UserDao;
import com.uni.dao.impl.UserDaoimpl;
import com.uni.service.UserService;

public class UserServiceImpl implements UserService {
    private UserDao userDao = new UserDaoimpl();

    @Override
    public void save() {
        userDao.save();
    }

    public void setUserDao(UserDaoimpl userDao) {
        this.userDao = userDao;
    }
}

```

8.1.3 配置Spring

applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id = "userDao" class = "com.uni.dao.impl.UserDaoimpl"></bean>
    <bean id = "userService" class = "com.uni.service.impl.UserServiceImpl">
        <property name="userDao" ref = "userDao"/>
    </bean>
</beans>

```

8.1.4 Servlet层

UserServlet.java

```
package com.uni.web;

import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}
```

8.1.5 配置Tomcat

略过。

8.1.6 测试

访问网页加上后缀/userServlet

IDEA终端结果

save running...

8.2 ApplicationContext应用上下文获取方式

经过8.1的环境搭建可发现，应用上下文对象是通过

在doGet方法里 new ClassPathXmlApplicationContext(Spring配置文件); 方式获取，但每次从容器获得Bean时都编写这一句，，会加载多次配置文件，应用上下文对象也创建了多次，影响性能。

在Web项目，可使用 `ServletContextListener` 监听web应用的启动，在Web应用启动时，加载Spring的配置文件，创建应用上下文对象ApplicationContext，再将其存储到最大域 `servletContext`，这样就可以在任意位置从域中获得 ApplicationContext

现做一个解耦的测试，仅测试

8.2.1 编写监听类获取ApplicationContext对象

ContextLoaderListener.java

```
package com.uni.listener;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContext;
```

```

public class ContextLoaderListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext servletContext = sce.getServletContext();
        // 读取 web.xml中的全局参数
        String contextConfigLocation = servletContext.getInitParameter("contextConfigLocation");
        ApplicationContext app = new ClassPathXmlApplicationContext(contextConfigLocation);
        // 将 Spring应用上下文对象 存储到 ServletContext域中
        servletContext.setAttribute("app", app);
        System.out.println("Spring容器创建完毕");
    }
}

```

WebApplicationContextUtils.java

```

package com.uni.listener;
import org.springframework.context.ApplicationContext;
import javax.servlet.ServletContext;
public class WebApplicationContextUtils {
    public static ApplicationContext getWebApplicationContext(ServletContext servletContext){
        return (ApplicationContext) servletContext.getAttribute("app");
    }
}

```

8.2.2 配置web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">
    <servlet>
        <servlet-name>UserServlet</servlet-name>
        <servlet-class>com.uni.web.UserServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>UserServlet</servlet-name>
        <url-pattern>/userServlet</url-pattern>
    </servlet-mapping>

    <!-- 配置监听器 -->
    <listener>
        <listener-class>com.uni.listener.ContextLoaderListener</listener-class>
    </listener>
    <!-- 全局初始化参数 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>applicationContext.xml</param-value>
    </context-param>
</web-app>

```

8.2.3 Servlet层

UserServlet.java

```

package com.uni.web;

```

```

import com.uni.listener.WebApplicationContextUtils;
import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ServletContext servletContext = this.getServletContext();
        ApplicationContext app = WebApplicationContextUtils.getWebApplicationContext(servletContext);
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}

```

8.3 Spring提供获取应用上下文的工具

Spring 提供了一个监听器 ContextLoaderListener，内部加载Spring配置文件，创建应用上下文对象，并存储到ServletContext域中，提供了一个客户端工具WebApplicationContextUtils供使用者获得应用上下文对象

故只有两个步骤

1. 在web.xml中配置ContextLoaderListener监听器（导入spring-web坐标）
2. 使用WebApplicationContextUtils获得应用上下文对象ApplicationContext

pom.xml

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

```

web.xml

```

<!-- 配置监听器 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<!-- 全局初始化参数 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>

```

UserServlet.java

```

package com.uni.web;

import com.uni.service.UserService;
import org.springframework.context.ApplicationContext;

```

```

import org.springframework.web.context.support.WebApplicationContextUtils;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ServletContext servletContext = this.getServletContext();
        ApplicationContext app = WebApplicationContextUtils.getWebApplicationContext(servletContext);
        UserService userService = app.getBean(UserService.class);
        userService.save();
    }
}

```

8.4 Spring集成web环境步骤总结

1. 配置ContextLoaderListener监听器
2. 使用WebApplicationContextUtils获得应用上下文

9. Spring AOP

9.1 简介

AOP 为 Aspect Oriented Programming 的 缩写，面向切面编程（切面的暂时理解：功能增强方法和目标方法统一），通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术

AOP是OOP（面向对象编程）的延续，是软件开发的热点，也是 Spring框架的重要内容，是函数式编程的一种衍生泛型。

利用 AOP 可以对业务逻辑的各个部分进行隔离，从而降低业务逻辑各部分之间的耦合度，提高程序可重用性，同时提高开发效率

9.2 AOP 作用及其优势

作用：在程序运行期间，在不修改源码的情况下对方法进行功能增强

优势：减少重复代码，提高开发效率，并且便于维护

9.3 AOP 的底层实现原理介绍

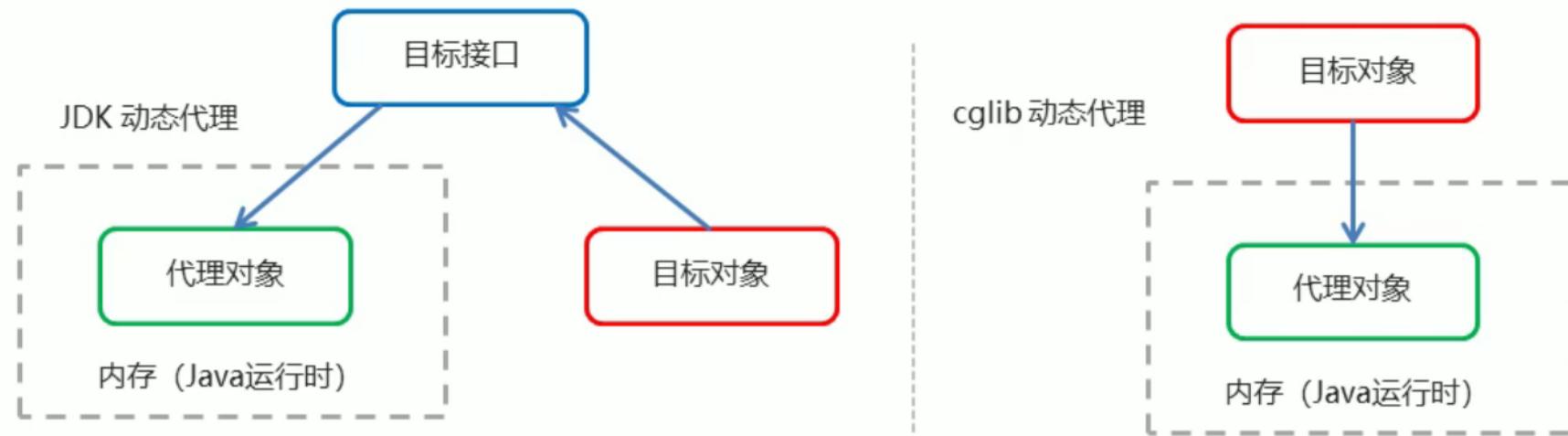
实际上，AOP的底层是通过 Spring 提供的动态代理技术实现的。

在运行期间，Spring通过动态代理技术动态的生成代理对象，代理对象方法执行时进行增强功能的接入，然后再调用目标对象的方法，从而完成功能的增强。

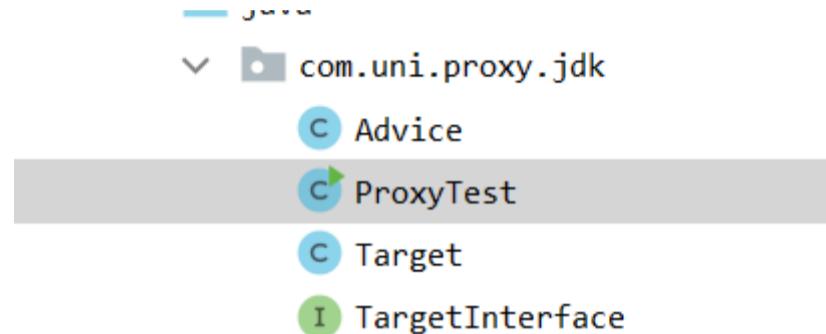
9.4 AOP 动态代理技术

常用动态代理技术

- JDK代理：基于接口的动态代理技术
- cglib 代理：基于父类的动态代理技术



9.5 JDK的动态代理



Advice.java

```

package com.uni.proxy.jdk;

public class Advice {
    public void before(){
        System.out.println("前置增强");
    }
    public void afterReturning(){
        System.out.println("后置增强");
    }
}
  
```

ProxyTest.java

```

package com.uni.proxy.jdk;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyTest {
    public static void main(String[] args) {
        // 目标对象
        final Target target = new Target();
        // 增强对象
        final Advice advice = new Advice();

        // 返回值 就是动态生成的 代理对象
        TargetInterface proxy = (TargetInterface) Proxy.newProxyInstance(
            target.getClass().getClassLoader(), // 目标对象类加载器
            target.getClass().getInterfaces(), // 目标对象相同的接口字节码对象数组
            new InvocationHandler() {
                // 调用代理对象的任何方法 实质执行的都是 invoke 方法
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                    advice.before(); // 前置增强
                    ...
                }
            }
        );
    }
}
  
```

```

        Object invoke = method.invoke(target, args); // 执行目标方法
        advice.afterReturning(); // 后置增强
        return invoke;
    }
}

// 调用 代理对象的方法
proxy.save();
}
}

```

Target.java

```

package com.uni.proxy.jdk;

public class Target implements TargetInterface{
    @Override
    public void save() {
        System.out.println("save Running...");
    }
}

```

TargetInterface.java

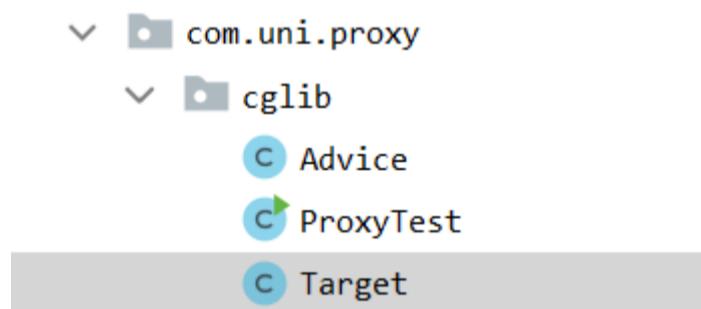
```

package com.uni.proxy.jdk;

public interface TargetInterface {
    public void save();
}

```

9.6 cglib的动态代理



Advice.java

```

package com.uni.proxy.cglib;

public class Advice {
    public void before(){
        System.out.println("前置增强");
    }
    public void afterReturning(){
        System.out.println("后置增强");
    }
}

```

ProxyTest.java

```

package com.uni.proxy.cglib;

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

```

```

public class ProxyTest {
    public static void main(String[] args) {
        // 目标对象
        final Target target = new Target();
        // 增强对象
        final Advice advice = new Advice();
        // 返回值 就是动态生成的代理对象 基于 cglib
        // 1. 创建增强器
        Enhancer enhancer = new Enhancer();
        // 2. 设置父类（目标）
        enhancer.setSuperclass(Target.class);
        // 3. 设置回调
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws
Throwable {
                advice.before(); // 执行前置
                Object invoke = method.invoke(target, args); // 执行目标
                advice.afterReturning(); // 执行后置
                return invoke;
            }
        });
        // 4. 创建代理对象
        Target proxy = (Target) enhancer.create();
        proxy.save();
    }
}

```

Target.java

```

package com.uni.proxy.cglib;

public class Target {
    public void save() {
        System.out.println("save Running...");
    }
}

```

9.7 AOP的相关概念

* **Target** (目标对象)

代理的目标对象

* **Proxy** (代理)

一个类被AOP织入增强后，就产生一个结果代理类

* **Joinpoint** (连接点)

连接点是指被拦截到的点。在Spring中，点指的是方法，因为Spring只支持方法类型的连接点。

* **Pointcut** (切入点)

所谓切入点是指我们要对哪些Joinpoint进行拦截的定义。

* **Advice** (通知 / 增强)

通知是指拦截到 Jointpoint 之后所要做的事情。

* **Aspect** (切面)

是切入点和通知 (引介) 的结合

* **Weaving** (织入)

是指把 增强应用 到 目标对象 来创建新的代理对象的过程。

Spring采用动态代理织入，而 AspectJ采用编译期织入和类装载期织入

9.8 AOP 开发明确的事项

9.8.1 需编写的内容

- 编写业务核心代码（目标类的目标方法）
- 编写切面类，切面类中有通知（增强功能方法）
- 在配置文件中，配置织入关系，即，将对应的通知与连接点进行结合

9.8.2 AOP 技术实现的内容

Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。

9.8.3 AOP 底层自动使用代理方式

在Spring框架中，会根据目标类是否实现了接口来决定采用那种动态代理的方式

9.9 知识要点

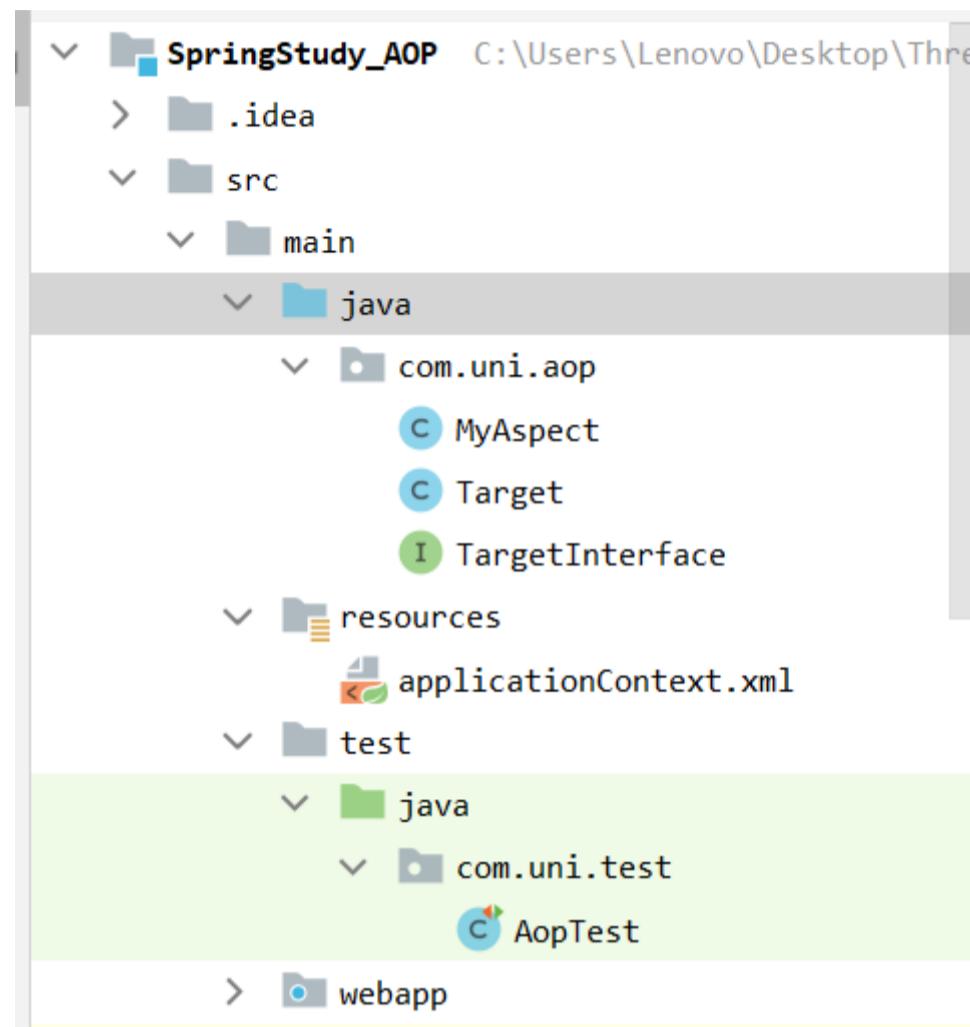
- aop：面向切面编程
- aop底层实现：基于JDK的动态代理和基于Cglib的动态代理
- aop的重点概念：
 - Pointcut (切入点) : 被增强的方法
 - Advice (通知/增强) : 封装增强业务逻辑的方法
 - Aspect (切面) : 切点 + 通知
 - Weaving (织入) : 将切点与通知结合的过程
- 开发明确事项：
 - 谁是切点 (切点表达式配置)
 - 谁是通知 (切面类中的增强方法)
 - 将切点和通知进行织入配置

10. 基于 XML 的 AOP 开发

10.1 AOP开发六部曲

1. 在pom.xml 导入 AOP 相关坐标

2. 创建目标接口 和 目标类 (内部有切点)
3. 创建切面类 (内部有增强方法)
4. 将目标类和切面类的对象创建权交给Spring
5. 在applicationContext.xml 中配置织入关系
6. 测试代码



本次主要在AopTest测试类中测试， webapp文件夹暂时没用

10.1.1 配置pom.xml

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringStudy_AOP</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.5.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.aspectj</groupId>
```

```

<artifactId>aspectjweaver</artifactId>
<version>1.8.4</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

</dependencies>

</project>

```

10.1.2 编写目标接口 和 目标类

TargetInterface.java

```

package com.uni.aop;

public interface TargetInterface {
    public void save();
}

```

Target.java

```

package com.uni.aop;

public class Target implements TargetInterface{
    public void save() {
        System.out.println("save Running...");
    }
}

```

10.1.3 编写切面类

MyAspect.java

```

package com.uni.aop;

public class MyAspect {

    public void before(){
        System.out.println("前置增强.");
    }

    public void after(){
        System.out.println("后置增强.");
    }
}

```

10.1.4 配置applicationContext.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:aop="http://www.springframework.org/schema/aop"

```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
<!-- 目标对象 -->
<bean id = "target" class = "com.uni.aop.Target"></bean>
<!-- 切面对象 -->
<bean id = "myAspect" class = "com.uni.aop.MyAspect"></bean>
<!-- 配置织入 告知Spring框架, 哪些方法(切点)需进行哪些增强(前置、后置...) -->
<aop:config>
    <!-- 声明切面 -->
    <aop:aspect ref = "myAspect">
        <!-- 切面: 切点 + 通知 -->
        <aop:before method="before" pointcut="execution(public void com.uni.aop.Target.save())"></aop:before>
        <aop:after method="after" pointcut="execution(public void com.uni.aop.Target.save())"></aop:after>
    </aop:aspect>
</aop:config>
</beans>

```

10.1.5 编写测试类

AopTest.java

```

package com.uni.test;

import com.uni.aop.TargetInterface;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;
    @Test
    public void test1(){
        target.save();
    }
}

```

运行结果:

前置增强.
save Running...
后置增强.

10.2 XML配置 AOP 详解

10.2.1 切点表达式的写法

表达式语法

`execution([修饰符] 返回值类型 包名.类名.方法名(参数))`

- 访问修饰符可省略

- 返回值类型、包名、类名、方法名可以使用型号 * 代表任意
- 包名与类名之间一个点 . 代表当前包下的类，两个点 .. 表示当前包 及其子包下的类
- 参数列表可以使用两个点 .. 表示任意个数，任意类型的参数列表

例如：

```
execution(public void com.uni.aop.Target.method());
execution(void com.uni.aop.Target.*(..)); // (...) 表示Target类任意参数包括无参 .* 表示任意方法
execution(* com.uni.aop.*.*(..)); // aop 包下的任意类的任意方法
execution(* com.uni.aop..*.*(..)); // aop包及其子包下的任意类的任意方法
```

故

```
<aop:before method="before" pointcut="execution(public void com.uni.aop.Target.save())"/>
```

可写成

```
<aop:before method="before" pointcut="execution(* com.uni.aop.*.*(..))"/>
```

10.2.2 通知的类型

语法格式

```
<aop:通知类型 method = "切面类中的方法名" pointcut = "切点表达式" />
```

名称	标签	说明
前置通知	<aop:before>	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	<aop:after-returning>	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	<aop:around>	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	<aop:throwing>	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	<aop:after>	用于配置最终通知。无论增强方法执行是否有异常都会执行

环绕通知 Demo.

MyAspect.java

```
public Object around(ProceedingJoinPoint pjp) throws Throwable{
    System.out.println("环绕前.");
    Object proceed = pjp.proceed();
    System.out.println("环绕后");
    return proceed;
}
```

applicationContext.xml

```
<aop:config>
    <!-- 声明切面 -->
    <aop:aspect ref = "myAspect">
        <!-- 切面： 切点 + 通知 -->
        <aop:around method="around" pointcut="execution(* com.uni.aop.*.*(..))"/>
    </aop:aspect>
</aop:config>
```

AopTest.java

```

package com.uni.test;

import com.uni.aop.TargetInterface;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;
    @Test
    public void test1(){
        target.save();
    }
}

```

10.2.3 切点表达式的抽取

当多个增强的切点表达式相同时，可将切点表达式进行抽取，在增强中使用 pointcut-ref 属性代替 pointcut 属性来引用抽取后的切点表达式

applicationContext.xml

```

<!-- 配置织入 告知Spring框架，哪些方法（切点）需进行哪些增强（前置、后置...） -->
<aop:config>
    <!-- 声明切面 -->
    <aop:aspect ref = "myAspect">
        <!-- 抽取切点表达式 -->
        <aop:pointcut id = "myPointcut" expression="execution(* com.uni.aop.*.*(..))"/>
        <!-- 切面： 切点 + 通知 -->
        <aop:around method="around" pointcut-ref="myPointcut"/>
        <aop:before method="before" pointcut-ref="myPointcut"/>
    </aop:aspect>
</aop:config>

```

10.3 知识要点

- Aop织入的配置

```

<aop:config>
    <aop:aspect ref = "切面类">
        <aop:before method = "通知方法名称" pointcut = "切点表达式"/>
    </aop:aspect>
</aop:config>

```

- 通知的类型：前置、后置、环绕、异常抛出、最终
- 切点表达式的写法

`execution([修饰符]返回值类型 包名.类名.方法名(参数))`

11. 基于注解的 AOP 开发

11.1 注解AOP开发六部曲

1. 创建目标接口 和 目标类 (内部有切点)
2. 创建切面类 (内部有增强方法)
3. 将目标类和切面类的对象创建权交给Spring
4. 在切面类中使用注解配置织入关系
5. 在配置文件中开启组件扫描和AOP的自动代理
6. 测试

目标接口 TargetInterface.java

```
package com.uni.aop.anno;

public interface TargetInterface {
    public void save();
}
```

目标类 Target.java

```
package com.uni.aop.anno;

import org.springframework.stereotype.Component;

@Component("target")
public class Target implements TargetInterface {

    public void save() {
        System.out.println("save Running...");
    }
}
```

切面类 MyAspect.java

```
package com.uni.aop.anno;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component("myAspect")
@Aspect // 标注当前类 MyAscept 是一个切面类
public class MyAspect {

    @Before(value = "execution(* com.uni.aop.anno.*.*(..))")// 配置前置通知
    public void before(){
        System.out.println("前置增强.");
    }

    public void after(){
        System.out.println("后置增强.");
    }

    public Object around(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("环绕前.");
        Object proceed = pjp.proceed();
        System.out.println("环绕后");
        return proceed;
    }
}
```

配置 spring文件

applicationContext-anno.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:aop="http://www.springframework.org/schema/aop"
    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">
    <!-- 组件扫描 -->
    <context:component-scan base-package="com.uni.aop.anno"/>

    <!-- aop 自动代理 -->
    <aop:aspectj-autoproxy/>
</beans>

```

测试类 AnnoTest.java

```

package com.uni.test;

import com.uni.aop.anno.TargetInterface;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-anno.xml")
public class AnnoTest {
    @Autowired
    private TargetInterface target;

    @Test
    public void test1(){target.save();}
}

```

11.2 注解配置 AOP 详解

11.2.1 注解通知的类型

名称	注解	说明
前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方法执行是否有异常都会执行

11.2.2 切点表达式的抽取

同 XML 配置 AOP 一样，可以将切点表达式抽取，方式是在切面内定义方法，在该方法上使用@Pointcut注解定义切点表达式，然后在增强注解中引用

切面类 MyAspect.java

```
package com.uni.aop.anno;
```

```

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component("myAspect")
@Aspect // 标注当前类 MyAspect 是一个切面类
public class MyAspect {

    @Before("pointcut()")// 配置前置通知
    public void before(){
        System.out.println("前置增强.");
    }

    @AfterReturning("MyAspect.pointcut()")
    public void after(){
        System.out.println("后置增强.");
    }

    // 定义切点表达式
    @Pointcut("execution(* com.uni.aop.anno.*.*(..))\"")
    public void pointcut(){}
}

```

11.3 知识要点

11.3.1 注解AOP开发三部曲

- 使用 @Aspect 标注切面类
- 使用@通知注解标注通知方法 比如 @Before
- 在Spring配置文件中配置aop自动代理 <aop:aspectj-autoproxy/>

11.3.2 通知注解类型

具体参考 11.2.1

@Before、@AfterReturning、@Around、@AfterThrowing、@After

12. Spring 的事务控制

12.1 编程式事务控制相关对象

12.1.1 接口PlatformTransactionManager

该接口是Spring的事务管理器，内部提供了常用操作事务的方法

方法	说明
TransactionStatus getTransaction(TransactionDefination defination)	获取事务的状态信息
void commit(TransactionStatus status)	提交事务
void rollback(TransactionStatus status)	回滚事务

Ps: 不同的Dao层技术有不同的实现类，比如

- (1) 当Dao层技术是 `Jdbc` 或者 `Mybatis` 时, `org.springframework.jdbc.datasource.DataSourceTransactionManager`
- (2) 当Dao层技术是 `hibernate` 时, `org.springframework.orm.hibernate5.HibernateTransactionManager`

12.1.2 对象 TransactionDefinition

事务的定义信息对象

方法	说明
<code>int getIsolationLevel()</code>	获得事务的隔离级别
<code>int getPropogationBehavior()</code>	获得事务的传播行为
<code>int getTimeout()</code>	获得超时时间
<code>boolean isReadOnly()</code>	是否只读

1. 事务隔离级别

设置隔离级别，可解决事务并发产生的问题，如脏读、不可重复读、虚读

- `ISOLATION_DEFAULT`
- `ISOLATION_READ_UNCOMMITTED`
- `ISOLATION_READ_COMMITTED`
- `ISOLATION_REPEATABLE_READ`
- `ISOLATION_SERIALIZABLE`

2. 事务传播行为

- `REQUIRED`: 如果当前没有事务，就新建一个事务，如果已存在一个事务中，加入到这个事务，一般的选择（默认值）
- `SUPPORTS`: 支持当前事务，如果当前没事务，就以非事务方式执行（没有事务）
- `MANDATORY`: 使用当前的事务，如果当前无事务，就抛出异常
- `REQUIRES_NEW`: 新建事务，如果当前在事务中，把事务挂起
- `NOT_SUPPORTED`: 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
- `NEVER`: 以非事务方式运行，如果当前存在事务，抛出异常
- `NESTED`: 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行`REQUIRED`类似的操作
- 超时时间：默认值是-1，没有超时限制。如果有，以秒为单位进行设置
- 是否只读：建议查询时设置为只读

12.1.3 接口 TransactionStatus

该接口提供的是事务具体的运行状态

方法	说明
<code>boolean hasSavepoint()</code>	是否存储回滚点
<code>boolean isCompleted()</code>	事务是否完成
<code>boolean isNewTransaction()</code>	是否是新事物
<code>boolean isRollbackOnly()</code>	事务是否回滚

12.1.4 知识要点

1. 编程式事务控制三大对象

- `PlatformTransactionManager`
- `TransactionDefinition`
- `TransactionStatus`

12.2 基于 XML 的声明式事务控制

12.2.1 声明式事务控制概念

Spring 的声明式事务顾名思义就是采用声明的方式来处理事务。这里所说的声明是指在配置文件中声明用在 Spring 配置文件中 声明式 的处理事务来代替 代码式 的处理事务。

12.2.2 声明式事务控制作用

- 事务管理不侵入开发的组件。具体来说，业务逻辑对象就不会意识到正在事务管理中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可。
- 在不需要事务管理时，只要在设定文件上修改以下，即可移去事务管理服务，无需改变代码重新编译，这样维护起来方便

PS: Spring 声明式 事务控制底层就是 AOP

12.2.3 声明式事务控制的实现

明确事项切点？通知？切面？

以转账效果为准

DAO 层: AccountDao.java 接口、 AccountDaoImpl.java 接口实现类

Service 层: AccountService.java 接口、 AccountServiceImpl.java 接口实现类

样例：转账时，先扣除一个用户的余额，再增添另一个用户的余额，若余额不足，则不增添，同时也不扣除。

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringStudy_AOP</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.5.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.4</version>
  </dependency>
</dependencies>
```

```

</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

</dependencies>
</project>

```

applicationContext.xml

```

<bean id = "accountDao" class = "com.uni.dao.impl.AccountDaoImpl"/>
<!-- 目标对象 内部的方法就是切点 -->
<bean id = "accountService" class = "com.uni.service.impl.AccountServiceImpl">
    <property name = "accountDao" ref = "accountDao"/>
</bean>
<!-- 配置 平台事务管理器 -->
<bean id = "transactionManage" class = "org.springframework.jdbc.datasource.DataSourceTransactionManage.java">
    <property name = "dataSource" ref = "dataSource" />
</bean>
<!-- 通知事务的增强 -->
<tx:advice id = "txAdvice" transaction-manager="transactionManger">
    <!--设置事务的属性信息 -->
    <tx:attributes>
        <tx:method name = "*" />
    </tx:attributes>
</tx:advice>
<!-- 配置事务的AOP织入 -->
<aop:config>
    <aop:advisor advice-ref = "txAdvice" pointcut = "execution(* com.uni.service.impl.*.*(..))">
</aop:config>

```

AccountController.java

```

public static void main(String[] args){
    ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");
    AccountService accountService = app.getBean("AccountService.class");
    accountService.transfer("a", "b", 500);
}

```

AccountServiceImpl.java

```

private AccountDao accountDao;
public void setAccountDao(AccountDao accountDao){this.accountDao = accountDao;}
public void transfer(String outPerson, String inPerson, double money){
    accountDao.out(outPerson, money);
    accountDao.in(inPerson, money);
}

```

```
<tx:method>
```

代表切点方法的事务参数的配置，例如

```
<tx:method name = "transfer" isolation="REPEATABLE_READ" propagation="REQUIRED" timeout="-1" read-only="false"/>
```

- name : 切点方法名称
- isolation: 事务的隔离级别
- propagation: 事务的传播行为
- timeout: 超时时间
- read-only: 是否只读

12.2.4 知识要点

声明式事务控制的配置要点

- 平台事务管理器配置
- 事务通知的配置
- 事务AOP织入的配置

12.3 基于注解的声明式事务控制

12.3.1 快速入门

applicationContext.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:aop="http://www.springframework.org/schema/aop"
       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">
    <!-- 组件扫描 -->
    <context:component-scan base-package="com.uni.aop.anno"/>
    <!-- aop 自动代理 -->
    <aop:aspectj-autoproxy/>

    <!-- 配置 平台事务管理器 -->
    <bean id = "transactionManager" class = "org.springframework.jdbc.datasource.DataSourceTransactionManager.java">
        <property name = "dataSource" ref = "dataSource" />
    </bean>
    <!-- 事务的注解驱动 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

AccountServiceImpl.java

```

@Service("accountService")
@Transactional(isolation = Isolation.REPEATABLE_READ) // 相当于 <tx:advice
public class AccountServiceImpl implements AccountService{
    @Autowired
    private AccountDao accountDao;

    @Transactional(isolation = Isolation.READ_COMMITTED)
    // 相当于 <tx:attributes> 里的 <tx:method name = "*" />
    public void transfer(String outPerson, String inPerson, double money){
        accountDao.out(outPerson, money);
        accountDao.in(inPerson, money);
    }
}

```

13.3.2 注解配置的解析

- 使用@Transactional在需要进行事务控制的类或是方法上修饰，注解可用的属性同 xml配置方式，例如隔离级别，传播行为等
- 注解使用在类上，那么该类下的所有方法都使用同一套注解参数配置
- 使用在方法上，不同的方法可以采用不同的事务参数配置
- XML配置文件中要开启事务的注解驱动 < tx: annotation-driven />

13.3.3 知识要点，三部曲

- 平台事务管理器配置 (XML方式)
- 事务通知的配置 (@Transactional注解配置)
- 事务注解驱动的配置 <tx:annotation-driven/>

toSpringMVC

1. 简介

SpringMVC是一种基于Java的实现MVC 设计模型的请求驱动类型的轻量级Web框架，属于SpringFrameWork的后序产品，已经融合在Spring Web Flow中

SpringMVC已成为目前最主流的MVC框架之一，且随着Spring3.0的发布，全面超越Struts2，成为最优秀的MVC框架。它通过一套注解，让一个简单的Java类成为处理请求的控制器，且无需实现任何接口。

同时还支持RESTful编程风格的请求。

2. SpringMVC 快速入门

2.1 开发步骤

1. 导入SpringMVC包
2. 配置Servlet
3. 编写Controller (POJO)
4. 将Controller使用注射配置到Spring容器中 (@Controller)

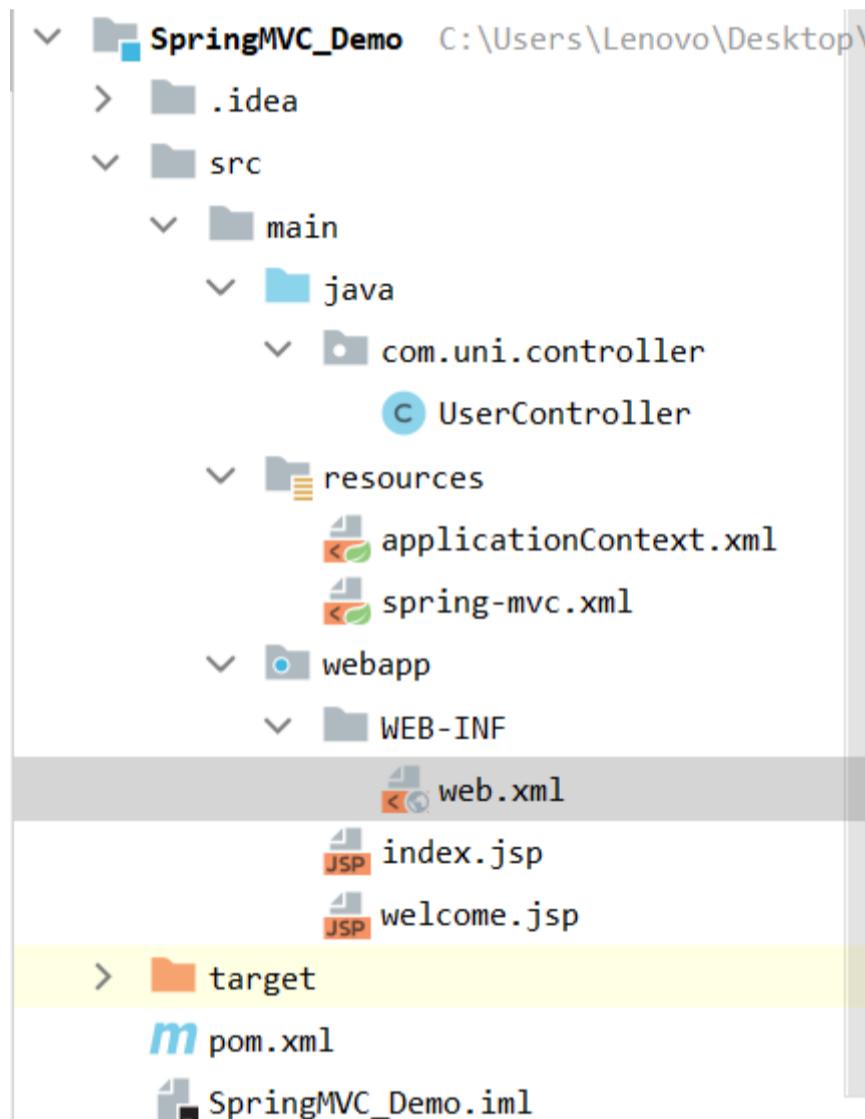
5. 配置spring-mvc.xml 文件 (配置组件扫描)

6. 执行访问测试

需求：客户端发起请求，服务端接受请求，执行逻辑并进行视图跳转

1. 导入SpringMVC相关坐标
2. 配置SpringMVC核心控制器DispatcherServlet
3. 创建Controller类和视图页面
4. 使用注解配置Controller类中业务方法的映射地址
5. 配置SpringMVC核心文件spring-mvc.xml
6. 客户端发起请求测试

2.2 代码实现



web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">
    <!-- 配置 SpringMVC 的前端控制器 -->
    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>SpringMVC_Demo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>SpringMVC_Demo Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.5.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>5.0.5.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.0.5.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
    </dependency>
```

```

<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.2.1</version>
</dependency>
</dependencies>

</project>

```

applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>

```

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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context.xsd">
    <!-- Controller的组件扫描 -->
    <context:component-scan base-package="com.uni.controller"/>
</beans>

```

UserController.java

```

package com.uni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class UserController {

    @RequestMapping("/welcome")
    public String save(){
        System.out.println("Controller save running...");
        return "welcome.jsp";
    }
}

```

welcome.jsp

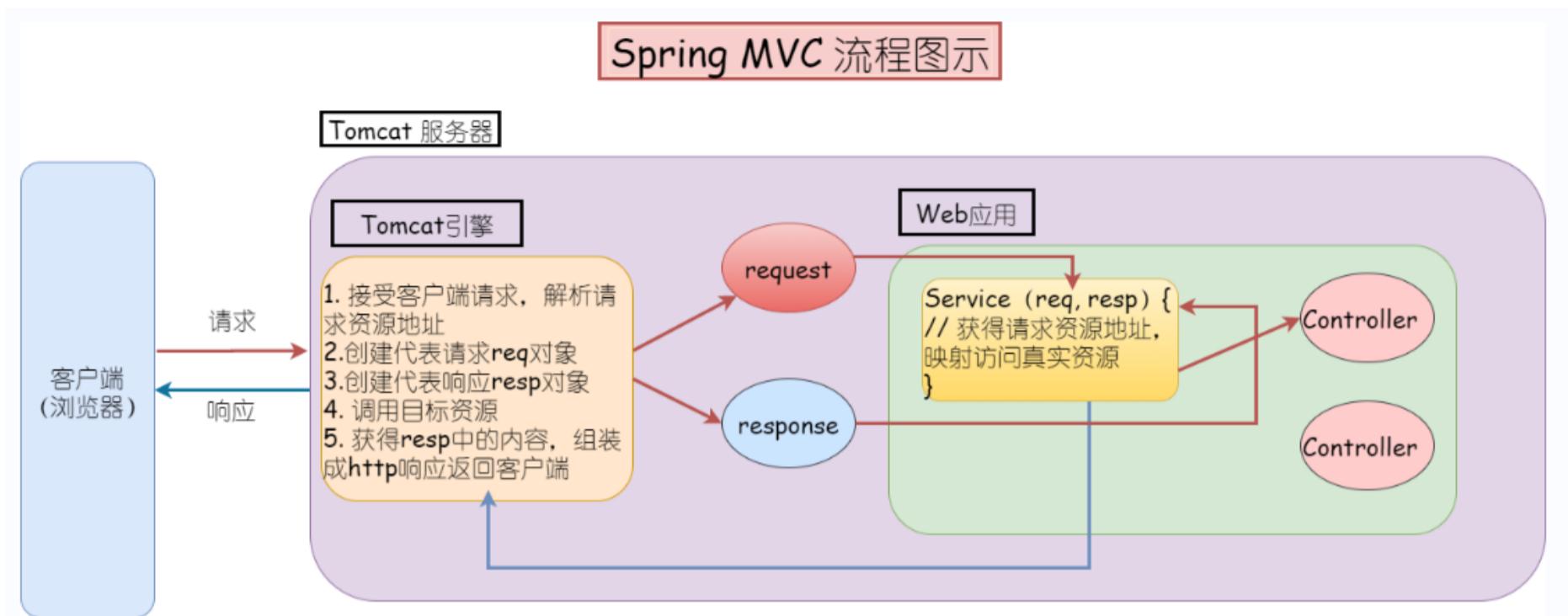
```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>欢迎访问!</h1>
</body>
</html>

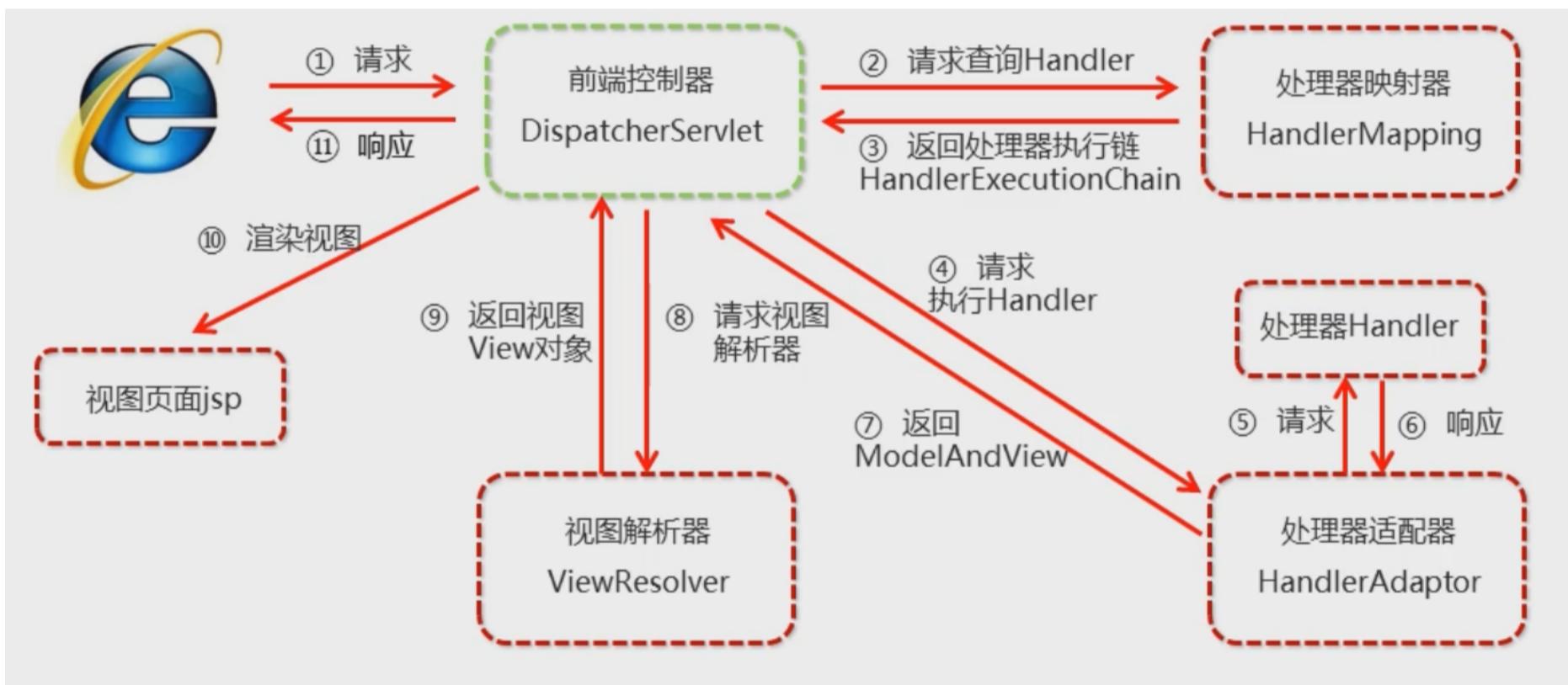
```

测试时，在网页后加上welcome，若跳转成功，则会显示 [欢迎访问！] 字样

3. SpringMVC 流程演示



SpringMVC的执行流程图



文字描述：

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器
3. 处理器映射器找到具体的处理器（可根据xml配置、注解进行查找），生成处理器对象及处理器拦截器（如果有则生成）一并返回给DispatcherServlet
4. DispatcherServlet调用HandlerAdapter处理器适配器
5. HandlerAdapter经过适配调用具体的处理器（Controller，也叫后端控制器）
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View
10. DispatcherServlet根据View进行渲染视图（即，将模型数据填充至视图）。DispatcherServlet响应用户

4. SpringMVC 组件解析

4.1 注解解析

4.1.1 @RequestMapping

作用：建立请求URL和处理请求之间的对应关系

位置：

- 类，请求URL的第一级访问目录。不写则默认为应用的根目录
- 方法，请求URL的第二级访问目录，与类上的使用@RequestMapping标注的一级目录一起组成访问虚拟路径。

属性：

- value：用于指定请求的URL，和path属性的作用相同
- method：用于指定请求的方式
- params：用于指定限制请求参数的条件。支持简单的表达式，要求请求参数的key和value必须配置的一模一样。

例如：

- params = {"count"}, 表示请求参数必须有count
- params = {"count!=10"}, 表示请求参数中count不能为10

测试：

UserController.java

```
package com.uni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/welcome", method = RequestMethod.GET) // 默认为GET
    public String save(){
        System.out.println("Controller save running...");
        return "/welcome.jsp";
    }
}
```

4.1.2 扫描注解分类

在spring-mvc.xml 中，对比之前的配置

```
<!-- Controller的组件扫描 -->
<context:component-scan base-package="com.uni.controller"/>
```

新的配置可以扩大包的范围，同时在指定对应的注解类型，其中context:include-filter表示选择当前包下的的注解，而context:excluded-filter 是选择当前包下除了该注解的其他所有注解

```
<!-- Controller的组件扫描 -->
<context:component-scan base-package="com.uni">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

4.2 XML配置

spring-mvc.xml

```

<!-- 配置内部资源视图解析器 -->
<bean id = "viewResolver" class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- /jsp/welcome.jsp -->
    <property name = "prefix" value = "/jsp/" />
    <property name="suffix" value = ".jsp"/>
</bean>

```

UserController.java

```

package com.uni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/welcome")
    public String save(){
        System.out.println("Controller save running...");
        return "welcome";
    }
}

```

4.3 知识要点

4.3.1 SpringMVC 相关组件

- 前端控制器：DispatcherServlet
- 处理器映射器：HandlerMapping
- 处理器适配器：HandlerAdapter
- 处理器：Handler
- 视图解析器：View Resolver
- 视图：View

4.3.2 SpringMVC的注解和配置

- 请求映射注解：@RequestMapping
- 视图解析器配置
 REDIRECT_URL_PREFIX="redirect:"
 FORWARD_URL_PREFIX="forward:"
 prefix="";
 suffix="";

5 Spring MVC 数据响应

5.1 响应方式

1. 页面跳转
 - 直接返回字符串
 - 通过 ModelAndView 对象返回
2. 回写数据
 - 直接返回字符串
 - 返回对象或集合

5.2 页面跳转

5.2.1 返回字符串形式

直接返回字符串：该方法会将返回的字符串与视图解析器的前后缀拼接后跳转

```
@RequestMapping("/welcome")
public String hello(){
    return "welcome";
}

<property name = "prefix" value = "/WEB-INF/views/" />
<property name = "suffix" value = ".jsp" />
```

转发资源地址: /WEB-INF/views/welcome.jsp

返回带有前缀的字符串：

转发: forward:/WEB-INF/views/welcome.jsp

重定向: redirect:/welcome.jsp

5.2.2 返回 ModelAndView 对象

UserController.java

形式一：在方法内部创建并操作 ModelAndView 对象

```
package com.uni.controller;

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

@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/test")
    public ModelAndView go(){
        /*
         Model : 模型 封装数据
         View: 视图 展示数据
        */
        ModelAndView modelAndView = new ModelAndView();
        // 设置模型数据
        modelAndView.addObject("username", "uni");
        // 设置视图名称
        modelAndView.setViewName("welcome");
        return modelAndView;
    }
}
```

形式二：方法传入 ModelAndView 对象，内部操作该对象

```
@RequestMapping("/test2")
public ModelAndView go2(ModelAndView modelAndView){
    modelAndView.addObject("username", "uni");
    modelAndView.setViewName("welcome");
    return modelAndView;
}
```

形式三：方法传入 Model 对象，内部操作后返回跳转的地址

```

@RequestMapping("/test3")
public String go3(Model model){
    model.addAttribute("username", "uni");
    return "welcome";
}

```

形式四：传入HttpServletRequest对象（不常用）（一般使用MVC封装的对象）

```

@RequestMapping("/test4")
public String go4(HttpServletRequest req){
    req.setAttribute("username", "uni");
    return "welcome";
}

```

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"
       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">
    <!-- Controller的组件扫描 -->
    <context:component-scan base-package="com.uni">
        <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置内部资源视图解析器 -->
    <bean id = "viewResolver" class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- /jsp/welcome.jsp -->
        <property name = "prefix" value = "/jsp/">
        <property name="suffix" value = ".jsp"/>
    </bean>

</beans>

```

welcome.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>${username}, 欢迎访问!</h1>
</body>
</html>

```

5.3 回写数据

5.3.1 直接返回字符串

在web基础，客户端访问服务端，若想直接回写字符串作为响应体返回，则：

`resp.getWriter().print("hello, world");` 即可，那么在Controller中：

1. 通过SpringMVC框架注入的response对象， 使用 resp.getWriter().print("hello, world") 回写数据，此时无需视图跳转，业务方法返回类型为 void

```
@RequestMapping("/test5")
public void go5(HttpServletRequest resp) throws IOException {
    resp.getWriter().print("uni, welcome!");
}
```

2. 将需要回写的字符串直接返回， 但此时需通过@ResponseBody注解告知SpringMVC框架，方法返回的字符串不是跳转， 而直接在http响应体中返回

```
@RequestMapping("/test6")
@ResponseBody // 提醒 SpringMVC 框架 不进行视图跳转，直接进行数据响应
public String go6(){
    return "uni, welcome";
}
```

5.3.2 返回json类型

配置JSON相关操作jar包的pom依赖

```
<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-databind</artifactId>
    <version>2.9.0</version>
</dependency>

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

UserController.java

```
@RequestMapping("/test7")
@ResponseBody
public String go7(){
    User user = new User();
    user.setName("uni");
    user.setAge(21);
    // 使用 json 转换工具 将对象转换成JSON格式字符串在返回
    ObjectMapper objectMapper = new ObjectMapper();
    String json = objectMapper.writeValueAsString(user);
    return json;
}
```

5.3.3 返回对象和集合

通过配置SpringMVC框架的处理器映射器RequestMappingHandlerAdapter实现返回一个对象时框架自动转化成对应的数据类型

UserController.java

```
@RequestMapping("/test8")
@ResponseBody
// 预期 SpringMVC 将 User转换成 json 格式的 字符串
public User go8(){
    User user = new User();
    user.setName("uni");
    user.setAge(21);
    return user;
}
```

spring-mvc.xml

```
<!-- 配置处理器映射器 -->
<bean class = "org.springframework.web.servlet.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class = "org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"></bean>
        </list>
    </property>
</bean>
```

之前注解@RequestBody 可返回json格式的字符串，为了简化配置，可使用mvc的注解驱动代替上述配置

```
<mvc:annotation-driven />
```

在SpringMVC的各个组件中，处理器映射器、处理器适配器、视图解析器称为SpringMVC的三大组件。

使用 `<mvc:annotation-driven/>` 可自动加载 处理器映射器 (RequestMappingHandlerMapping) 和 处理器适配器 (RequestMappingHandlerAdapter)，可用在spring-mvc.xml配置文件中使用。

同时使用 `<mvc:annotation-driven/>` 默认底层就会集成 jackson 进行对象或集合的 json 格式字符串的转换。

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.xsd
http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Controller的组件扫描 -->
    <context:component-scan base-package="com.uni">
        <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置内部资源视图解析器 -->
    <bean id = "viewResolver" class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- /jsp/welcome.jsp -->
        <property name = "prefix" value = "/jsp/">
```

```

<property name="suffix" value = ".jsp"/>
</bean>

<!-- mvc 的注解驱动 -->
<mvc:annotation-driven/>
</beans>

```

6. SpringMVC 获得请求数据

6.1 获得请求参数

客户端请求参数的格式是： name = value%age = value... ...

服务端要获得请求的参数，有时还需进行数据封装，SpringMVC可接受如下类型的参数：

- 基本类型参数
- POJO类型参数（简单JavaBean）
- 数组类型参数
- 集合类型参数

6.2 获得基本类型参数

Controller中的业务方法的参数名称要与请求参数的name一致，参数值会自动映射匹配。

测试url: http://localhost:8080/SpringMVC_Demo_war_exploded/user/test9?username=uuni&age=21

UserController.java

```

@RequestMapping("/test9")
@ResponseBody
public void go9(String username, int age){
    System.out.println(username);
    System.out.println(age);
}

```

6.3 获得POJO类型参数

Controller中的业务方法的POJO参数的属性名与请求参数的name一致，参数值会自动映射匹配

测试的Url: http://localhost:8080/SpringMVC_Demo_war_exploded/user/test10?name=uuni&age=21

UserController.java

```

@RequestMapping("/test10")
@ResponseBody
public void go10(User user){
    System.out.println(user);
}

```

User.java

```

package com.uni.domain;

public class User {
    private String name;
    private int age;
    public User(){}
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

@Override
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", age=" + age +
        '}';
}
}

```

6.4 获得数组类型参数

Controller中的业务方法数组名称与请求参数的name一致，参数值会自动映射匹配

测试的url: http://localhost:8080/SpringMVC_Demo_war_exploded/user/test11?strs=a&strs=b&strs=c

UserController.java

```

@RequestMapping("/test11")
@ResponseBody
public void go11(String[] strs){
    System.out.println(Arrays.asList(strs));
}

```

6.5 获得集合类型参数

获得集合参数时，要将集合参数包装到一个POJO中才可以。

测试url: http://localhost:8080/SpringMVC_Demo_war_exploded/form.jsp

form.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <form action = "${pageContext.request.contextPath}/user/test12" method="post">
        <%-- 表明是第几个User对象的username --%>
        用户1名称:<input type = "text" name = "userList[0].name"><br/>
        用户1年龄:<input type = "text" name = "userList[0].age"><br/>
        用户2名称:<input type = "text" name = "userList[1].name"><br/>
        用户2年龄:<input type = "text" name = "userList[1].age"><br/>
    </form>

```

```

<input type = "submit" value = "提交">
</form>
</body>
</html>

```

VO.java

```

package com.uni.domain;

import java.util.List;

public class VO {
    private List<User> userList;

    public List<User> getUserList() {
        return userList;
    }

    public void setUserList(List<User> userList) {
        this.userList = userList;
    }

    @Override
    public String toString() {
        return "VO{" +
            "userList=" + userList +
            '}';
    }
}

```

UserController.java

```

@RequestMapping("/test12")
@ResponseBody
public void go12(VO vo){
    System.out.println(vo);
}

```

当使用 Ajax 提交时，可以指定contentType为 json 格式，故在方法参数位置使用@RequestBody可以直接接受集合数据而无需 POJO 进行包装

测试 url: http://localhost:8080/SpringMVC_Demo_war_exploded/ajax.jsp

UserController.java

```

@RequestMapping("/test13")
@ResponseBody
public void go13(@RequestBody List<User> userList){
    System.out.println(userList);
}

```

ajax.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<script src="https://code.jquery.com/jquery-3.6.0.js"></script>
<head>
    <title>Title</title>
    <script>
        var userList = new Array();
        userList.push({name:"zhangsan", age:18});
    </script>

```

```

userList.push({name:"xiaoming", age:21});

$.ajax({
    type:"POST",
    url:"${pageContext.request.contextPath}/user/test13",
    data:JSON.stringify(userList),
    contentType:"application/json; charset=utf-8"
})
</script>
</head>
<body>

</body>
</html>

```

从本地导入JQuery时，spring-mvc.xml 里需开放对资源的访问

```
<mvc:resources mapping="/js/**" location="/js/" /> (/js 相当于 WEB-INF 同级目录下的js文件夹)
```

更方便的写法是 <mvc:default-servlet-handler/>

如果没有在MVC框架中找到资源则会在Tomcat服务器下查找资源

6.6 POST请求数据乱码问题

当post请求时，可设置一个过滤器对编码进行过滤，防止数据出现乱码

web.xml

```

<!-- 配置全局过滤的filter -->
<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>

```

6.7 参数绑定注解@RequestParam

当请求的参数名称与Controller的业务方法参数名称不一致时，可使用@RequestParam注解

注解@RequestParam 还有如下参数可使用：

- value：与请求参数名称
- required：在指定的请求参数是否必须包括，默认为true，提交时如果没有此参数则报错
- defaultValue：当没有指定请求参数时，则使用指定的默认值赋值

UserController.java

```

@RequestMapping("/test14")
@ResponseBody
public void go14(@RequestParam(value="username", required = false, defaultValue = "uni") String name){
    System.out.println(name);
}

```

有了@RequestParam注解后，该网页接受的参数名为username，默认值为uni。

6.8 获得 Restful 风格的参数@PathVariable

Restful 是一种软件架构风格、设计风格，而不是标准，只是提供了以组设计原则和约束条件。

主要用于客户端和服务器交互类的软件，基于这个风格的软件可以更简洁，更有层次，更易于实现缓存机制等。

Restful 风格的请求是使用 url + 请求方式 表示一次请求目的，HTTP协议里面四个表示操作方式：

- GET：获取资源
- POST：新建资源
- PUT：更新资源
- DELETE：删除资源

例如：

- /user/1 GET: 得到 id = 1 的user
- /user/1 DELETE: 删除 id = 1 的user
- /user/1 PUT: 更新 id = 1 的user
- /user POST: 新增user

上述url地址/user/1中的1就是要获得的请求参数，在SpringMVC中可以使用占位符进行参数绑定。

地址/user/1可以写成/user/{id}, 占位符{id}对应的就是1的值。

在业务方法中可使用@PathVariable注解进行占位符的匹配获取工作(默认为GET方式)

测试url: http://localhost:8080/SpringMVC_Demo_war_exploded/user/test15/uni123

UserController.java

```
@RequestMapping("/test15/{username}")
@ResponseBody
public void go15(@PathVariable(value="username") String name){
    System.out.println(name);
}
```

6.9 自定义类型转换器

SpringMVC默认已提供一些常用的类型转换器，例如客户端提交的字符串转换成int型进行参数设置

但并不是所有的数据类型都提供了转换器，没有提供的就需自定义转换器，例如：日期类型的数据就绪自定义转换器

自定义类型转换器开发步骤

- 定义转换器实现 Converter <Before Type, After Type> 接口
- 在配置文件中声明转换器
- 在 <annotation-driven> 中引用转换器

测试URL : http://localhost:8080/SpringMVC_Demo_war_exploded/user/test17?date=2017-06-01

UserController.java

```
@RequestMapping("/test17")
@ResponseBody
public void go17(Date date){
    System.out.println(date);
}
```

spring-mvc.xml

```

<!-- mvc 的 注解驱动 -->
<mvc:annotation-driven conversion-service="conversionService"/>

<!-- 声明转换器 -->
<bean id = "conversionService" class = "org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class = "com.uni.converter.DateConverter" ></bean>
        </list>
    </property>
</bean>

```

DateConverter.java

```

package com.uni.converter;

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

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

public class DateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String dateStr) {
        // 将日期字符串转换成日期对象后 返回
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
        Date date = null;
        try {
            date = format.parse(dateStr);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return date;
    }
}

```

6.10 获得 Servlet 相关API

SpringMVC 支持使用原始的 Servlet API 对象作为控制器方法的参数进行注入，常用对象有：

- HttpServletRequest
- HttpServletResponse
- HttpSession

UserController.java

```

@RequestMapping("/test18")
@ResponseBody
public void go18(HttpServletRequest req, HttpServletResponse resp, HttpSession session){
    System.out.println(req);
    System.out.println(resp);
    System.out.println(session);
}

```

6.11 获得请求头

6.11.1 @RequestHeader

使用@RequestHeader可获得请求头信息，相当于Web阶段学习的 `request.getHeader(name)`

属性有：

- value：请求头名称
- required：是否必须携带此请求头，默认为true

UserController.java

```
@RequestMapping("/test19")
@ResponseBody
public void go19(@RequestHeader(value = "User-Agent", required = false) String user_agent){
    System.out.println(user_agent);
}
```

6.11.2 @CookieValue

使用该注解可指定Cookie的值

属性有：

- value：指定cookie的名称
- required：是否必须携带此cookie

UserController.java

```
@RequestMapping("/test20")
@ResponseBody
public void go20(@CookieValue(value="JSESSIONID") String jsessionId){
    System.out.println(jsessionId);
}
```

6.12 文件上传

6.12.1 上传三要素

文件上传客户端三要素

- 表单项 `type = "file"`
- 表单提交方式为 post
- 表单的 `enctype` 属性是多部分表单形式，即 `enctype = "multipart/form-data"`

```
<form action = "${pageContext.request.contextPath}/user/test21" method="post" enctype="multipart/form-data">
    名称<input type = "text" name = "username"><br>
    文件<input type = "file" name = "uploadFile"><br>
    <input type = "submit" value = "提交">
</form>
```

6.12.2 上传原理

- 当 form 为多部分表单时，`request.getParameter()` 将失效
- `enctype = "application/x-www-form-urlencoded"`，form表单的正文内容格式是：
`key=value&key=value&key=value`
- 当 form 表单的 `enctype` 取值为 `Multipart/form-data` 时，请求正文内容就会变成多部分形式

6.13 单文件上传

- 导入 fileupload 和 io 坐标
- 配置文件上传解析器
- 编写文件上传代码

pom.xml

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
</dependency>
```

spring-mvc.xml

```
<!-- 配置文件上传解析器 -->
<bean id = "multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 上传文件的编码类型 -->
    <property name="defaultEncoding" value = "UTF-8"/>
    <!-- 上传单个文件的大小 -->
    <property name="maxUploadSizePerFile" value = "1000"/>
    <!-- 上传文件的总大小 -->
    <property name="maxUploadSize" value = "5000"/>
</bean>
```

测试url: http://localhost:8080/SpringMVC_Demo_war_exploded/upload.jsp

UserController.java

```
@RequestMapping("/test22")
@ResponseBody
public void go22(String username, MultipartFile uploadFile) throws IOException { // MultipartFile 的形参 必须和
jsp的input标签对应的name属性相同
    System.out.println(username);
    // 获得 上传文件的名称
    String originalFilename = uploadFile.getOriginalFilename();
    uploadFile.transferTo(new File("D:\\test\\" + originalFilename));
}
```

upload.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <form action = "${pageContext.request.contextPath}/user/test22" method="post" enctype="multipart/form-data">
        名称<input type = "text" name = "username"><br>
        文件<input type = "file" name = "uploadFile"><br>
        <input type = "submit" value = "提交">
    </form>
</body>
</html>
```

6.14 多文件上传

6.14.1 方式一：逐一处理

upload.jsp

```
<form action = "${pageContext.request.contextPath}/user/test22" method="post" enctype="multipart/form-data">
    名称<input type = "text" name = "username"><br>
    文件1<input type = "file" name = "uploadFile1"><br>
    文件2<input type = "file" name = "uploadFile2"><br>
    <input type = "submit" value = "提交">
</form>
```

UserController.java

```
@RequestMapping("/test22")
@ResponseBody
public void go22(String username, MultipartFile uploadFile1, MultipartFile uploadFile2) throws IOException {
    // MultipartFile 的形参 必须和 jsp的input标签对应的name属性相同
    System.out.println(username);
    // 获得 上传文件的名称
    String originalFilename1 = uploadFile1.getOriginalFilename();
    String originalFilename2 = uploadFile2.getOriginalFilename();
    uploadFile1.transferTo(new File("D:\\\\test\\\\" + originalFilename1));
    uploadFile2.transferTo(new File("D:\\\\test\\\\" + originalFilename2));
}
```

6.14.2 方式二：数组处理

upload.jsp

```
<form action = "${pageContext.request.contextPath}/user/test22" method="post" enctype="multipart/form-data">
    名称<input type = "text" name = "username"><br>
    文件1<input type = "file" name = "uploadFile"><br>
    文件2<input type = "file" name = "uploadFile"><br>
    <input type = "submit" value = "提交">
</form>
```

UserController.java

```
@RequestMapping("/test22")
@ResponseBody
public void go22(String username, MultipartFile[] uploadFile) throws IOException {
    // MultipartFile 的形参 必须和 jsp的input标签对应的name属性相同
    System.out.println(username);
    // 获得 上传文件的名称
    for (MultipartFile file : uploadFile){
        String fileName = file.getOriginalFilename();
        file.transferTo(new File("D:\\\\test\\\\" + fileName));
    }
}
```

6.15 SpringMVC 获得请求数据 知识要点

MVC实现数据请求的方式

- 基本类型参数

- POJO类型参数
- 数组类型参数
- 集合类型参数

MVC获取数据细节

- 中文乱码问题
- @RequestParam 和 @PathVariable
- 自定义类型转换器
- 获得 Servlet 相关 API
- @RequestHeader 和 @CookieValue
- 文件上传

7. SpringMVC 拦截器

7.1 拦截器 (interceptor) 的作用

Spring MVC 的拦截器类似于 Servlet 开发中的过滤器 Filter，用于对处理器进行预处理和后处理

将 拦截器 按一定的顺序连接成一条链，这条链称为 拦截器链 (Interceptor Chain)。

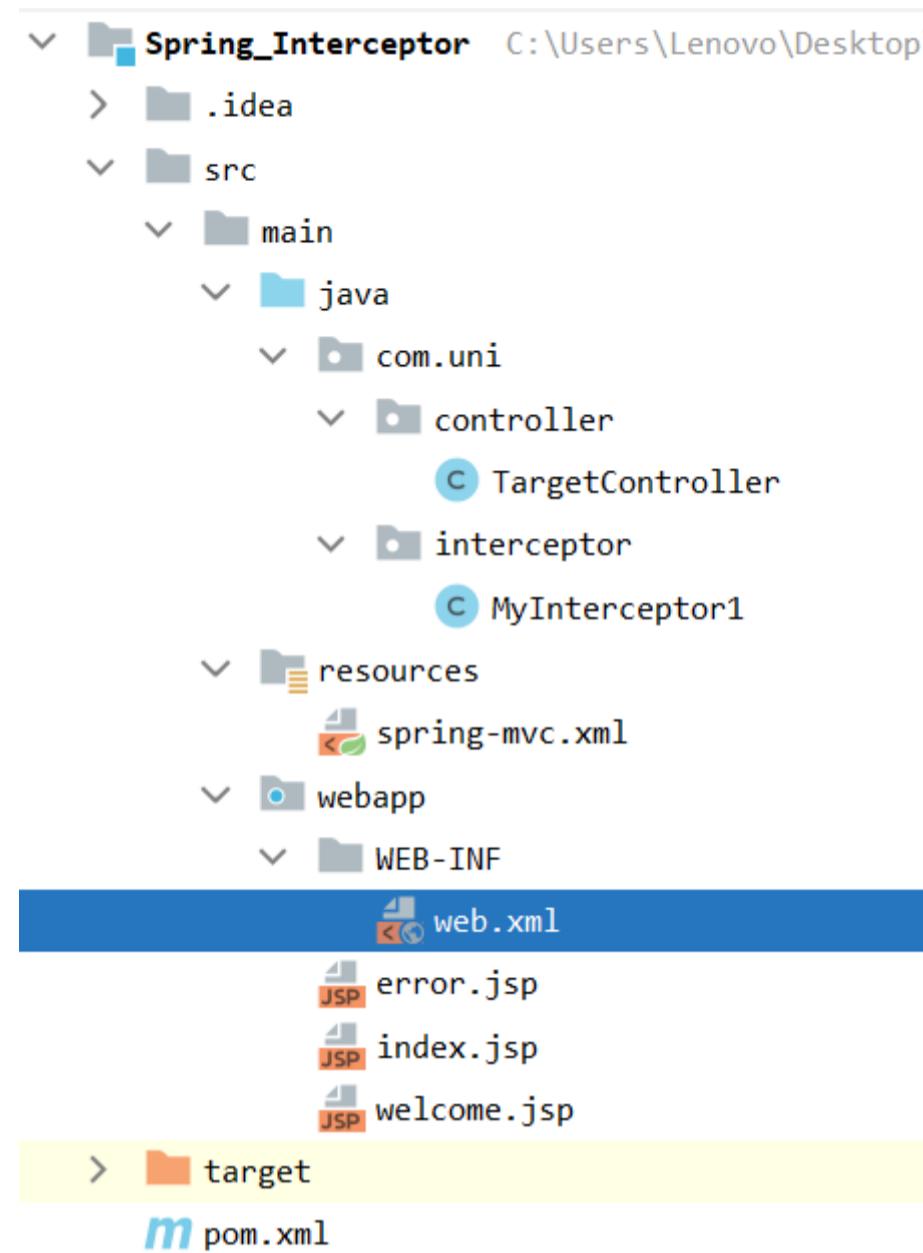
在访问被拦截的方法或字段时，拦截器链中的拦截器就会按之前定义的顺序被调用。拦截器也是AOP思想的具体实现。

7.2 拦截器和过滤器的区别

区别	过滤器	拦截器
使用范围	是servlet规范中的一部分，任何JavaWeb工程都可以使用	属于SpringMVC框架，只有该框架的工程能使用
拦截范围	在url-pattern中配置了 /* 后，可以对所有要访问的资源拦截	只会拦截访问的控制器方法，如果访问的是jsp, html, css, image或者js是不会进行拦截的

7.3 拦截器三部曲

1. 创建拦截器实现HandlerInterceptor接口
2. 配置拦截器
3. 测试拦截器的拦截效果



pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>SpringMVC_Demo</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<name>SpringMVC_Demo Maven Webapp</name>
<!-- FIXME change it to the project's website -->
<url>http://www.example.com</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
```

```

<artifactId>spring-context</artifactId>
<version>5.0.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>

<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.2.1</version>
</dependency>
</dependencies>

</project>

```

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">
    <!-- 配置 SpringMVC 的前端控制器 -->
    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>

```

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:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    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">
    <!-- 注解驱动 -->
    <mvc:annotation-driven/>
    <!-- 配置视图解析器 -->
    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value = "/">
        <property name="suffix" value = ".jsp"/>
    </bean>
    <!-- 静态资源权限开放 -->
    <mvc:default-servlet-handler/>
    <!-- 组件扫描 扫描Controller -->
    <context:component-scan base-package="com.uni.controller"/>

    <!-- 配置拦截器 -->
    <mvc:interceptors>
        <mvc:interceptor>
            <!-- path 表示对那些资源执行拦截操作 /** 表示所有-->
            <mvc:mapping path="/**"/>
            <bean class = "com.uni.interceptor.MyInterceptor1"/>
        </mvc:interceptor>
    </mvc:interceptors>
</beans>

```

MyInterceptor1.java

```

package com.uni.interceptor;

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

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

public class MyInterceptor1 implements HandlerInterceptor {
    // 目标方法执行前
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        System.out.println("preHandle Done.");
        String id = request.getParameter("id");
        if("uni1024".equals(id)){
            return true;
        } else{
            request.getRequestDispatcher("/error.jsp").forward(request, response);
            return false;
        }
    }
    // 目标方法执行后 视图返回前
    @Override

```

```

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
    System.out.println("postHandle Done.");
}

// 整个流程都执行完毕后
@Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
    System.out.println("afterCompletion Done.");
}
}

```

TargetController.java

```

package com.uni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class TargetController {
    @RequestMapping("/target")
    public ModelAndView show(){
        System.out.println("目标资源执行...");
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("age", 21);
        modelAndView.setViewName("welcome");
        return modelAndView;
    }
}

```

index.jsp

```

<html>
<body>
<h2>Hello World!</h2>
</body>
</html>

```

welcome.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>欢迎访问 ${age} 岁的 uni!</h1>
</body>
</html>

```

error.jsp

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>id不匹配!</h1>
</body>
</html>

```

测试URL:

<http://localhost:8080/target>

<http://localhost:8080/target?id=uni1>

<http://localhost:8080/target?id=uni1024>

7.4 拦截器三大方法

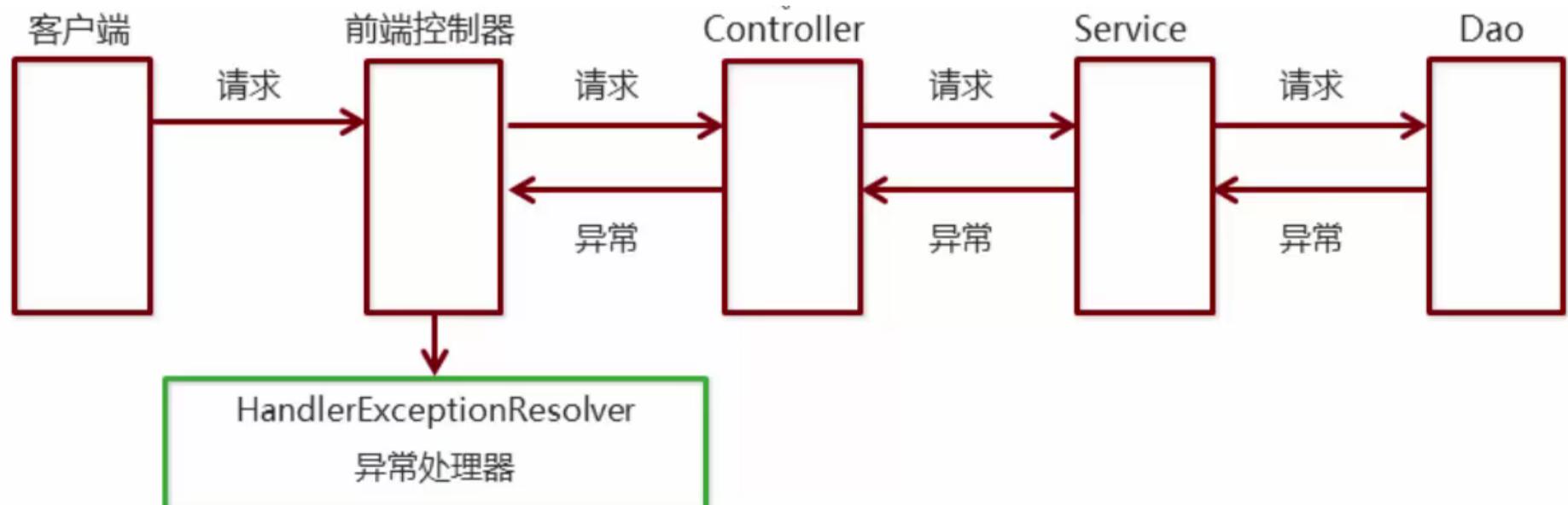
方法名	说明
preHandle()	方法将在请求处理之前进行调用，该方法的返回值是布尔值Boolean类型的，当它返回为false时，表示请求结束，后续的Interceptor和Controller都不会再执行；当返回值为 true 时就会继续调用下一个 Interceptor 的preHandle方法
postHandle()	该方法是在当前请求进行处理后被调用，前提是 preHandle() 方法的返回值为true才能成功调用，且它会在 DispatcherServlet 进行视图返回渲染之前被调用，所以可以在这个方法对 Controller 处理后的 ModelAndView 对象进行操作
afterCompletion()	该方法将在整个请求结束之后，即在DispatcherServlet渲染了对应的视图之后执行，前提是preHandle方法的返回值为true时才能被调用

8. SpringMVC 异常处理

8.1 异常处理的思路

系统中异常包括两类： 预期异常 和 运行时异常 RuntimeException ，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试等手段减少运行时异常的发生。

系统的DAO、Service、Controller 出现都通过 throws Exception 向上抛出，最后由 SpringMVC前端控制器交由异常处理器进行异常处理



8.2 异常处理的两种方式

- 使用SpringMVC 提供的 简单异常处理器 `SimpleMappingExceptionResolver`
- 实现Spring的异常处理接口 `HandlerExceptionResolver` 自定义自己的异常处理器

8.3 简单异常处理器 `SimpleMappingExceptionResolver`

```
<!--配置简单映射异常处理器-->
<bean
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="defaultErrorView" value="error"/> 默认错误视图
    <property name="exceptionMappings">
        <map> 异常类型 错误视图
            <entry key="com.itheima.exception.MyException" value="error"/>
            <entry key="java.lang.ClassCastException" value="error"/>
        </map>
    </property>
</bean>
```

这里直接在Service层测试，所以Dao层的接口实现类为空

`web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">
    <!-- 全局初始化参数 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <!-- 配置 SpringMVC 的前端控制器 -->
    <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

`applicationContext.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"
       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">

    <!-- 组件扫描 -->
    <context:component-scan base-package="com.uni" />
</beans>

```

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:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       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">

    <!-- 注解驱动 -->
    <mvc:annotation-driven/>
    <!-- 配置视图解析器 -->
    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value = "/">
        <property name="suffix" value = ".jsp"/>
    </bean>
    <!-- 静态资源权限开放 -->
    <mvc:default-servlet-handler/>
    <!-- 组件扫描 扫描Controller -->
    <context:component-scan base-package="com.uni.controller"/>

    <!-- 配置简单映射异常处理器 -->
    <bean class = "org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
            <map>
                <entry key = "java.lang.ClassCastException" value = "error1"></entry>
                <entry key = "com.uni.exception.MyException" value = "error2"></entry>
            </map>
        </property>
    </bean>
</beans>

```

DemoController.java

```

package com.uni.controller;

import com.uni.exception.MyException;
import com.uni.service.DemoService;
import com.uni.service.impl.DemoServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

```

```
@Controller  
public class DemoController {  
    @Autowired  
    private DemoService demoService;  
    @RequestMapping(value="/show1")  
    public String show1(){  
        demoService.show1();  
        return "index";  
    }  
    @RequestMapping(value="/show2")  
    public String show2() throws MyException {  
        demoService.show2();  
        return "index";  
    }  
}
```

DemoServiceImpl.java

```
package com.uni.service.impl;  
  
import com.uni.dao.DemoDao;  
import com.uni.exception.MyException;  
import com.uni.service.DemoService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service("demoService")  
public class DemoServiceImpl implements DemoService {  
    @Autowired  
    private DemoDao demoDao;  
    @Override  
    public void show1() {  
        System.out.println("抛出类型异常");  
        Object str = "zhangsan";  
        Integer num = (Integer)str;  
    }  
    @Override  
    public void show2() throws MyException {  
        System.out.println("自定义异常");  
        throw new MyException();  
    }  
}
```

DemoDaoImpl.java

```
package com.uni.dao.impl;  
  
import com.uni.dao.DemoDao;  
import org.springframework.stereotype.Repository;  
  
@Repository("demoDao")  
public class DemoDaoImpl implements DemoDao {  
    @Override  
    public void show1() {  
    }  
}
```

测试：

访问 <http://localhost:8080/show2> 时 显示 自定义异常

访问 <http://localhost:8080/show1> 时 显示 抛出类型异常

8.4 自定义异常四部曲

1. 创建异常处理器类实现 HandlerExceptionResolver
2. 配置异常处理器
3. 编写异常页面
4. 编写异常跳转

MyExceptionResolver.java

```
package com.uni.resolver;

import com.uni.exception.MyException;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

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

public class MyExceptionResolver implements HandlerExceptionResolver {
    // 参数 Exception : 异常对象
    // 返回值 ModelAndView: 跳转的错误信息
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) {
        ModelAndView modelAndView = new ModelAndView();
        if(e instanceof MyException){
            modelAndView.addObject("info", "自定义异常");
        } else if (e instanceof ClassCastException){
            modelAndView.addObject("info", "类转换异常");
        }
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
```

DemoController.java

```
package com.uni.controller;

import com.uni.exception.MyException;
import com.uni.service.DemoService;
import com.uni.service.impl.DemoServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DemoController {
    @Autowired
    private DemoService demoService;
    @RequestMapping(value="/show1")
    public String show1(){
        demoService.show1();
        return "index";
    }
}
```

```
@RequestMapping(value="/show2")
public String show2() throws MyException {
    demoService.show2();
    return "index";
}
```

DemoServiceImpl.java

```
package com.uni.service.impl;

import com.uni.dao.DemoDao;
import com.uni.exception.MyException;
import com.uni.service.DemoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("demoService")
public class DemoServiceImpl implements DemoService {
    @Autowired
    private DemoDao demoDao;
    @Override
    public void show1() {
        Object str = "zhangsan";
        Integer num = (Integer)str;
    }
    @Override
    public void show2() throws MyException {
        throw new MyException();
    }
}
```

spring-mvc.xml

```
<!-- 自定义异常处理器 -->
<bean class = "com.uni.resolver.MyExceptionResolver"/>
```

error.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>通用的错误提示页面</h1>
    <h1>${info}</h1>
</body>
</html>
```

1. 前言

1.1 原始 JDBC 操作

1.1.1 查询数据

```
//1. 注册驱动
Class.forName("com.mysql.jdbc.Driver");
//2. 获得连接
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");
//3. 获得statement
PreparedStatement statement = connection.prepareStatement("select id, username, password from user");
//4. 执行查询
ResultSet resultSet = statement.executeQuery();
//5. 遍历结果集
while(resultSet.next()){
    // 封装实体
    User user = new User();
    user.setId(resultSet.getInt("id"));
    user.setUsername(resultSet.getString("username"));
    user.setPassword(resultSet.getString("password"));
    // user实体封装完毕
    System.out.println(user);
}
// 释放资源
resultSet.close();
statement.close();
connection.close();
```

1.1.2 插入数据

```
//1. 模拟实体对象
User user = new User();
user.setId(2);
user.setUsername("tom");
user.setPassword("123456");

//2. 注册驱动
Class.forName("com.mysql.jdbc.Driver");
//3. 获得连接
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");
//4. 获得statement
PreparedStatement statement = connection.prepareStatement("insert into user(id, username, password)
values(?, ?, ?)");
//5. 设置占位符
statement.setInt(1, user.getId());
statement.setString(2, user.getUsername());
statement.setString(3, user.getPassword());
//6. 执行更新操作
statement.executeUpdate();
//7. 释放资源
statement.close();
connection.close();
```

1.2 原始 JDBC 操作分析

1.2.1 存在的问题

1. 数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能
2. sql语句在代码中硬编码，造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码
3. 查询操作时，需要手动将结果集中的数据手动封装到实体中。插入操作时，需要手动将实体的数据设置到sql语句的占位符位置

1.2.2 针对上述问题的解决方案

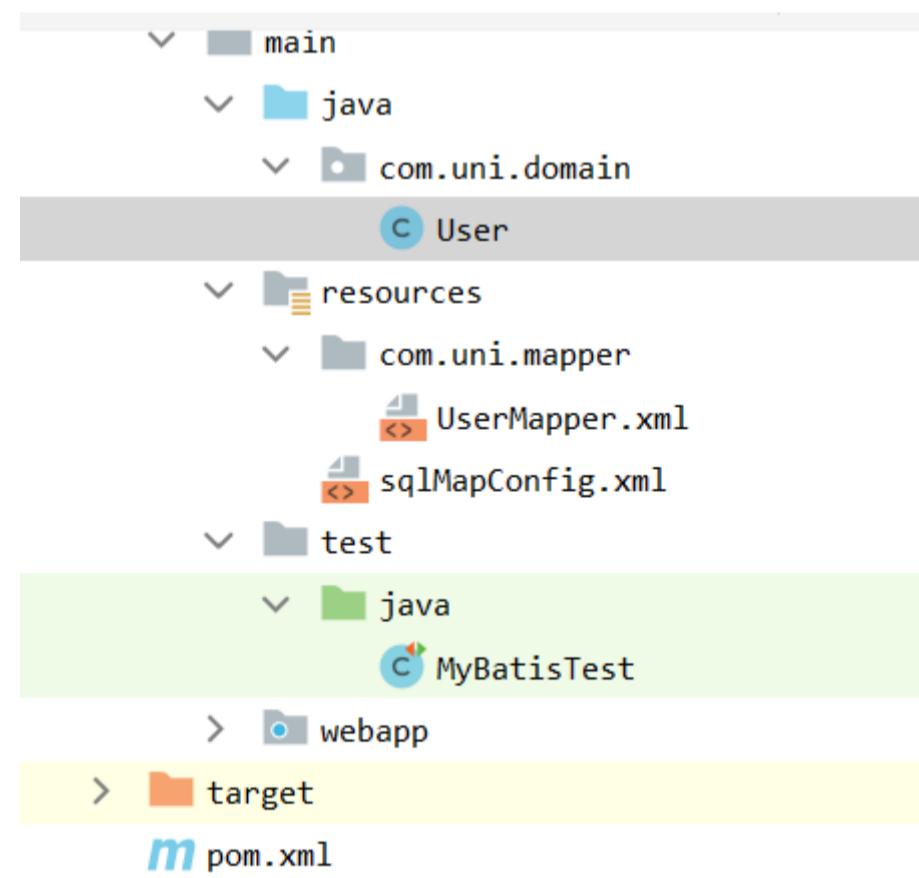
1. 使用数据库连接池初始化连接资源
2. 将sql语句抽取到 xml 配置文件中
3. 使用反射、内省等底层技术，自动将实体与表进行属性与字段的自动映射

2. Mybatis 简介

- 基于Java的持久层（DAO层）框架，内部封装了JDBC，使开发者只需要关注SQL语句本身，而无需花精力处理加载驱动、创建连接、创建statement等繁杂过程
- mybatis 通过XML或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中的sql的动态参数进行映射生成最终执行的sql语句
- 最后， mybatis 框架执行 sql 并将结果映射为java对象并返回。采用ORM思想解决了实体和数据库映射的问题，对 jdbc 进行了封装，屏蔽了 jdbc api 底层访问细节。

3.Mybatis开发六部曲（以查询为例）

项目结构如下图所示



在MyBatisTest.java类中进行测试，故webapp文件夹暂时没用。

3.1. 添加Mybatis的坐标

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>
<packaging>war</packaging>

<name>Mybatis_quick</name>
<groupId>org.example</groupId>
<artifactId>Mybatis_quick</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.32</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.6</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>

</project>

```

3.2. 创建user数据表

Mysql5.7环境

SQL语句

创建test数据库 -> 创建user测试表 -> 插入三条测试数据

```

create database test default character set utf8;

create table user(
    id INT primary key AUTO_INCREMENT,
    username VARCHAR(100),
    password VARCHAR(100)
) AUTO_INCREMENT = 1;

insert into user(username, password) values("uni", "123456")
insert into user(username, password) values("zhangsan", "123456789")
insert into user(username, password) values("lisi", "abc")

```

3.3. 编写User实体类

User.java

```
package com.uni.domain;

public class User {
    private int id;
    private String username;
    private String password;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", password='" + password + '\'' +
            '}';
    }
}
```

3.4. 编写映射文件

UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.userMapper">
    <select id="findAll" resultType="com.uni.domain.User">
        select * from user
    </select>
</mapper>
```

3.5. 编写核心文件

SqlMapConfig.xml

```

<?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>
    <!-- 数据源 环境 -->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/test"/>
                <property name="username" value="root"/>
                <property name="password" value="include666"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="com/uni/mapper/UserMapper.xml"></mapper>
    </mappers>
</configuration>

```

3.6. 编写测试类

MyBatisTest.java

```

import com.uni.domain.User;
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 java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class MyBatisTest {
    @Test
    public void testSelectAll() throws IOException {
        // 1. 获得 核心 配置文件
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        // 2. 获得 session 工厂对象
        SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
        // 3. 获得 session 回话对象
        SqlSession sqlSession = build.openSession();
        // 4. 执行操作 参数: namespace + id
        List<User> userList = sqlSession.selectList("userMapper.findAll");
        // 5. 打印数据
        System.out.println(userList);
        // 6. 释放资源
        sqlSession.close();
    }
}

```

4. MyBatis 映射文件概述



5. MyBatis 增删改操作

5.1 插入操作

基于 3 的内容 添加一部分代码

UserMapper.xml

```
<!-- 插入操作 -->
<select id="save" parameterType="com.uni.domain.User">
    insert into user values(#{id}, #{username}, #{password})
</select>
```

MyBatisTest.java

```
@Test
public void testInsert() throws IOException {
    // 1. 模拟 user对象
    User user = new User();
    user.setUsername("abc");
    user.setPassword("520");
    // 2. 获得 核心 配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 3. 获得 session 工厂对象
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    // 4. 获得 session 回话对象
    SqlSession sqlSession = build.openSession();
    // 5. 执行操作 参数: namespace + id
    sqlSession.insert("userMapper.save", user);
    // MyBatis 执行更新操作, 需提交事务
    // 6. 提交事务
    sqlSession.commit();
    // 7. 释放资源
    sqlSession.close();
}
```

插入操作注意事项:

- 插入语句用 `insert` 标签
- 在映射文件中使用 `parameterType` 属性指定要插入的数据类型
- SQL语句中使用`#`{实体属性名}方式引用实体中的属性值
- 插入操作使用的API是`sqlSession.insert("命名空间.id", 实体对象)`

- 插入操作设计数据库的数据变化，所以要调用sqlSession对象的commit方法提交事务

5.2 修改操作

UserMapper.xml

```
<!-- 修改操作 -->
<update id = "update" parameterType="com.uni.domain.User">
    update user set username=#{username}, password=#{password} where id = #{id}
</update>
```

MyBatisTest.java

```
@Test
public void testUpdate() throws IOException {
    // 1. 模拟 user对象
    User user = new User();
    user.setId(1);
    user.setUsername("woaini");
    user.setPassword("1314");
    // 2. 获得 核心 配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 3. 获得 session 工厂对象
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    // 4. 获得 session 回话对象
    SqlSession sqlSession = build.openSession();
    // 5. 执行操作 参数: namespace + id
    sqlSession.update("userMapper.update", user);
    // MyBatis 执行更新操作， 需提交事务
    // 6. 提交事务
    sqlSession.commit();
    // 7. 释放资源
    sqlSession.close();
}
```

修改操作注意事项

- 修改语句使用 update标签
- 修改操作使用的API是 `sqlSession.update("命名空间.id", 实体对象);`

5.3 删除操作

UserMapper.xml

```
<!-- 删除操作 pt为单个参数时{}里可填写任意变量名-->
<delete id="delete" parameterType="java.lang.Integer">
    delete from user where id = #{id}
</delete>
```

MyBatisTest.java

```
@Test
public void testUpdate() throws IOException {
    // 1. 模拟 user对象
    User user = new User();
    user.setId(1);
    user.setUsername("woaini");
```

```
user.setPassword("1314");
// 2. 获得 核心 配置文件
InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
// 3. 获得 session 工厂对象
SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
// 4. 获得 session 回话对象
SqlSession sqlSession = build.openSession();
// 5. 执行操作 参数: namespace + id
sqlSession.update("userMapper.update", user);
// MyBatis 执行更新操作， 需提交事务
// 6. 提交事务
sqlSession.commit();
// 7. 释放资源
sqlSession.close();
}
```

删除操作注意事项：

- 删除语句使用 delete标签
- Sql语句使用 #{任意字符串}方式引用传递的单个参数
- 删除操作使用的API是 sqlSession.delete("命名空间.id", Object);

6. MyBatis 核心配置文件概述

6.1 层级关系

- configuration 配置
 - properties属性
 - settings属性
 - typeAliases 类型别名
 - typeHandlers 类型处理器
 - objectFactory 对象工厂
 - plugins 插件
 - environments 环境
 - environment 环境变量
 - transactionManager 事务管理器
 - dataSource 数据源
 - databaseIdProvider 数据库厂商标识
 - mappers 映射器

6.2 MyBatis 常用配置解析

6.2.1 environments 标签

数据库环境的配置， 支持多环境配置

```

<environments default="development"> <!-- 指定默认环境名称-->
    <environment id="development"><!--指定当前环境名称 -->
        <transactionManager type="JDBC"/> <!-- 指定事务管理类型为JDBC -->
        <dataSource type="POOLED"><!-- 指定当前数据源类型为连接池 -->
            <!-- 数据源配置的基本参数 -->
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <property name="url" value="jdbc:mysql://localhost:3306/test"/>
            <property name="username" value="root"/>
            <property name="password" value="密码待填写"/>
        </dataSource>
    </environment>
</environments>

```

在上述子标签中

事务管理器 transactionManager 类型 有两种

- JDBC: 直接使用JDBC的提交和回滚设置，依赖于从数据源得到的连接来管理事务作用域
- MANAGED: 几乎不做事。不提交或回滚连接，而让容器来管理事务的整个声明周期（如JEE的应用服务器的上下文）。默认情况下会关闭连接，然而一些容器并不希望这样做，故要将closeConnection属性设置为false来组织它默认的关闭行为。

数据源 dataSource 类型 有三种

- UNPOOLED: 这个数据源的实现知识每次被请求时打开和关闭连接
- POOLED: 这种数据源的实现利用“池”的概念将JDBC连接对象组织起来
- JNDI: 这个数据源的实现是为了能在如EJB或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后防止一个JNDI上下文的应用

6.2.2 Mappers 标签

该标签的作用是加载映射，方式有：

- 使用相对类路径的资源引用， 如

```
<mapper resource = "org/mybatis/builder/AuthorMapper.xml"/>
```

- 使用完全限定资源定位符 (URL) ， 如

```
<mapper url="file:///var/mappers/AuthorMapper.xml"/>
```

- 使用映射器接口实现类的完全限定类名，如

```
<mapper class="org.mybatis.builder.AuthorMapper"/>
```

- 使用包内的映射器接口实现全部注册为映射器，如

```
<package name = "org.mybatis.builder "/>
```

6.2.3 Properties 标签

实际开发中，习惯将数据源的配置信息单独抽取成一个properties文件，该标签可以加载额外配置的properties文件



jdbc.properties

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=密码待填

```

sqlMapConfig.xml

```

<!-- 通过properties标签加载外部properties文件-->
<properties resource="jdbc.properties"></properties>

<!-- 数据源 环境 -->
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"></transactionManager>
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>

```

6.2.4 typeAliases 标签

例子:

```

<typeAliases>
    <typeAlias type = "com.uni.domain.User" alias="user"/>
</typeAliases>

```

除了自定义方式，MyBatis框架还提供一些常用的类名别名

别名	数据类型
string	String
long	Long
int	Integer
double	Double
boolean	Boolean
...	...

6.2.5 知识小结

核心配置文件常用配置

1. properties标签：加载外部的properties文件

```
<properties resource="jdbc.properties"/>
```

2. typeAliases标签：设置类型别名

```
<typeAlias type = "com.uni.domain.User" alias="user"/>
```

3. mappers标签：加载映射配置

```
<mapper resource="com/uni/mapper/UserMapper.xml"/>
```

4. environments标签：数据源环境配置标签

- environments default =

- environment id =

- transactionManager type =

- dataSource type =
 - property name = value =
 - property name= value =
 - ...

7. MyBatis 响应API

7.1 SqlSession 工厂构建器-FactoryBuilder

SqlSessionFactoryBuilder

常用API: `SqlSessionFactory build(InputStream inputStream)`

通过加载 mybatis 的核心文件的输入流的形式构建一个SqlSessionFactory对象

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resource.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

其中, Resource工具类, 在org.apache.ibatis.io 包中, 作用是帮助我们从类路径下、文件系统或者一个web URL中加载资源文件

7.2 SqlSession 工厂对象 -Factory

SqlSessionFactory

常用的两个创建实例的方法

方法	解释
<code>openSession()</code>	会默认开启一个事务, 但事务不会自动提交, 也意味着需手动提交该事务, 更新操作数据才会持久化到数据库中
<code>openSession(boolean autoCommit)</code>	参数为是否自动提交, 为true时则会自动提交事务

MyBatisTest.java

```
@Test
public void testInsert() throws IOException {
    // 1. 模拟 user对象
    User user = new User();
    user.setUsername("xyz");
    user.setPassword("520");
    // 2. 获得 核心 配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 3. 获得 session 工厂对象
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    // 4. 获得 session 回话对象
    SqlSession sqlSession = build.openSession(true);
    // 5. 执行操作 参数: namespace + id
    sqlSession.insert("userMapper.save", user);
    // 6. 释放资源
    sqlSession.close();
}
```

7.3 SqlSession 会话对象

SqlSession 实例在MyBatis中是非常强大的一个类，在这里会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。

执行语句的方法主要有：

```
<T> T selectOne(String statement, Object parameter);
<E> List<E> selectList(String statement, Object parameter);
int insert(String statement, Object parameter);
int update(String statement, Object parameter);
int delete(String statement, Object parameter);
```

操作事务的方法主要有：

```
void commit();
void rollback();
```

示例：selectOne

根据id 查询数据

UserMapper.xml

```
<!-- 根据 id 进行查询 -->
<select id="findById" resultType="com.uni.domain.User" parameterType="int">
    select * from user where id = #{id}
</select>
```

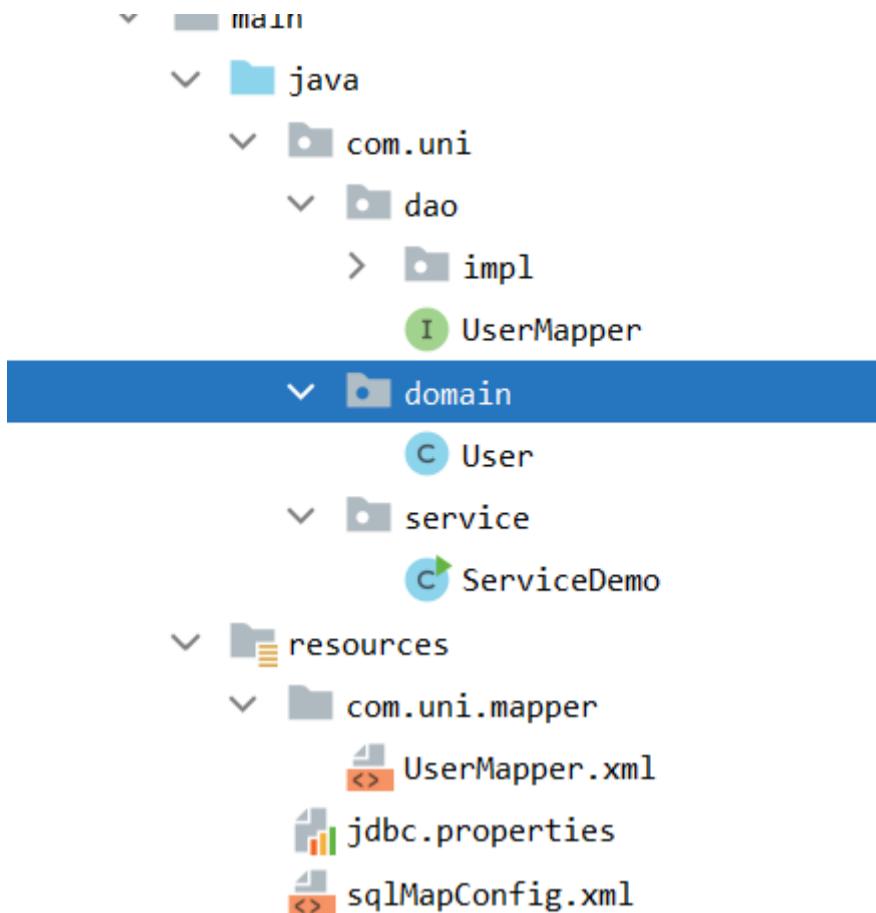
MyBatisTest.java

```
@Test
public void testSelectById() throws IOException {
    // 1. 获得 核心 配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 2. 获得 session 工厂对象
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    // 3. 获得 session 回话对象
    SqlSession sqlSession = build.openSession();
    // 4. 执行操作 参数: namespace + id
    User user = (User) sqlSession.selectOne("userMapper.findById", 2);
    // 5. 打印数据
    System.out.println(user);
    // 6. 释放资源
    sqlSession.close();
}
```

8. MyBatis的DAO层实现

8.1 传统接口开发方式

项目结构如下图所示



UserMapper.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="userMapper">
    <!-- 查询操作 -->
    <select id="findAll" resultType="com.uni.domain.User">
        select * from user
    </select>
</mapper>

```

sqlMapConfig.xml

```

<?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>
    <!-- 通过properties标签加载外部properties文件-->
    <properties resource="jdbc.properties"></properties>
    <!-- 数据源 环境 -->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}" />
                <property name="url" value="${jdbc.url}" />
                <property name="username" value="${jdbc.username}" />
                <property name="password" value="${jdbc.password}" />
            </dataSource>
        </environment>
    </environments>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="com/uni/mapper/UserMapper.xml"></mapper>
    </mappers>
</configuration>

```

UserMapperImpl.java

```

package com.uni.dao.impl;

import com.uni.dao.UserMapper;

```

```

import com.uni.domain.User;
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 java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class UserMapperImpl implements UserMapper {
    public List<User> findAll() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = build.openSession();
        List<User> userList = sqlSession.selectList("userMapper.findAll");
        return userList;
    }
}

```

测试类ServiceDemo.java

```

package com.uni.service;

import com.uni.dao.UserMapper;
import com.uni.dao.impl.UserMapperImpl;
import com.uni.domain.User;

import java.io.IOException;
import java.util.List;

public class ServiceDemo {
    public static void main(String[] args) throws IOException {
        // 创建DAO层对象 DAO层是手动编写的
        UserMapper userMapper = new UserMapperImpl();
        List<User> all = userMapper.findAll();
        System.out.println(all);
    }
}

```

8.2 代理开发方式

采用 Mybatis 的 代理开发实现 DAO层的开发，这种方式为主流

只需编写 Mapper 接口（相当于DAO接口），由 MyBatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 DAO接口实现类方法

Mapper接口开发规范

1. Mapper.xml 文件中的namespace 与 mapper接口的全限定名相同
2. Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql的parameterType的类型相同
4. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的结果resultType的类型相同



Demo：根据 ID 查询信息

UserMapper.xml

```

package com.uni.service;
import com.uni.dao.UserMapper;
import com.uni.domain.User;
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 java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class ServiceDemo {
    public static void main(String[] args) throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = build.openSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        // 测试查询所有
        List<User> all = mapper.findAll();
        System.out.println(all);
        System.out.println("----分割线----");
        // 测试查询单个
        User user = mapper.findById(3);
        System.out.println(user);
    }
}

```

DAO层 UserMapper.java

```

package com.uni.dao;

import com.uni.domain.User;

import java.io.IOException;
import java.util.List;

public interface UserMapper {
    public List<User> findAll() throws IOException;
    public User findById(int id);
}

```

UserMapper.xml

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

    <!-- 查询操作 -->
    <select id="findAll" resultType="com.uni.domain.User">
        select * from user
    </select>

    <!-- 根据id查询 -->
    <select id="findById" parameterType="int" resultType="com.uni.domain.User">
        select * from user where id = #{id}
    </select>
</mapper>
```

Demo : 确定用户名和密码是否匹配

DAO层 UserMapper.java

```
package com.uni.dao;
import com.uni.domain.User;
import java.util.List;
public interface UserMapper {
    public List<User> findByCondition(User user);
}
```

MyBatis配置文件sqlMapConfig.xml

```
<?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>

    <!-- 通过properties标签加载外部properties文件-->
    <properties resource="jdbc.properties"></properties>

    <!-- 设置对象别名 -->
    <typeAliases>
        <typeAlias type="com.uni.domain.User" alias="user"/>
    </typeAliases>

    <!-- 数据源 环境 -->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="com/uni/mapper/UserMapper.xml"></mapper>
    </mappers>
</configuration>
```

日志配置文件log4j.properties，其中模式为debug 可以改成info模式

```
### direct log message to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=$d{ABSOLUTE} %5p %c {1} :%L - %m%n

### direct messages to file mylog.log ####
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=d:/mylog.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1};%L - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ####
log4j.rootLogger=debug, stdout
```

接口与SQL映射文件 UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.UserMapper">
    <select id="findByCondition" parameterType="user" resultType="user">
        select * from user where username = #{username} and password = #{password}
    </select>
</mapper>
```

测试类 MyBatisTest.java

```
@Test
public void testFindByCondition() throws IOException {
    // 获得 核心 配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 获得 session 工厂对象
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    // 获得 session 回话对象
    SqlSession sqlSession = build.openSession();

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    // 模拟条件 User
    User condition = new User();
    condition.setId(2);
    condition.setUsername("abc");
    condition.setPassword("520");

    // 查询 并 输出
    List<User> userList = mapper.findByCondition(condition);
    System.out.println(userList);
    // 释放资源
    sqlSession.close();
}
```

9. MyBatis 映射文件 深入

9.1 动态 SQL 语句

9.1.1 < if > 标签

根据实体类的不同取值，使用不同的SQL语句查询。

比如在 id 如果不为空时可以根据id查询，如果username不为空时，还要加入用户名作为条件

UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.UserMapper">
    <select id="findByCondition" parameterType="user" resultType="user">
        select * from user
        <where>
            <if test="id!=0">
                and id = #{id}
            </if>

            <if test="username!=null">
                and username=#{username}
            </if>

            <if test="password!=null">
                and password=#{password}
            </if>
        </where>
    </select>
</mapper>
```

上述的配置保证，在执行 `sqlSession.getMapper(接口类名.class).getMapper("findByCondition").findByCondition(User user)` 时，

会根据 user对象的具体值来进行查询，(1) user = null，那么执行的SQL语句则为

```
select * from user 即相当于查询表user的所有数据。 (2) user的 id 和 password不为空，那么执行的SQL语句则会加上 and id = #{id} and password = #{password}
```

这实现了一种动态SQL的效果

9.1.2 < foreach > 标签

改标签类似于循环，可以遍历一个对象，比如列表、数组、集合

现给一个测试Demo，可以根据id的集合来查询所有存在该id集合的用户

UserMapper.xml

```
<select id="findByIds" parameterType="list" resultType="user">
    select * from user
    <where>
        <foreach collection="list" open="id in (" close = ")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
</mapper>
```

测试类 MyBatisTest.java

```
@Test
public void testFindByIds() throws IOException {
    // 获得核心配置文件
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    // 获得 session 工厂对象
```

```

SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
// 获得 session 回话对象
SqlSession sqlSession = build.openSession();
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
// 模拟ids数据
List<Integer> ids = new ArrayList<Integer>();
ids.add(1);
ids.add(2);
List<User> userList = mapper.findByIds(ids);
System.out.println(userList);
// 释放资源
sqlSession.close();
}

```

9.2 SQL 片断抽取

语法:

```
<sql id = "sqlid"> sql 语句 </sql>
```

UserMapper.xml

```

<!-- sql 语句抽取 -->
<sql id ="selectUser">select * from user</sql>
<select id="findByCondition" parameterType="user" resultType="user">
    <include refid="selectUser"/>
    <where>
        <if test="id!=0">
            and id = #{id}
        </if>

        <if test="username!=null">
            and username=#{username}
        </if>

        <if test="password!=null">
            and password=#{password}
        </if>
    </where>
</select>

```

9.3 MyBatis 映射文件配置常用标签

<标签名 >	作用
select	查询
insert	插入
update	修改
delete	删除
where	where条件
if	if判断
foreach	循环
sql	sql片断抽取

9.4 typeHandlers 标签

无论是 MyBatis 在预处理语句 (PreparedStatement) 中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器将获取的值以合适的方式转换成Java类型。下表则是一些默认的类型处理器。

类型处理器	Java类型	JDBC类型
BooleanTypeHandler	java.lang.Boolean.boolean	数据库兼容的BOOLEAN
ByteTypeHandler	java.lang.Byte.byte	数据库兼容的NUMERIC或BYTE
ShortTypeHandler	java.lang.Short.short	数据库兼容的NUMERIC或SHORT INTEGER
IntegerTypeHandler	java.lang.Integer.int	数据库兼容的NUMERIC或INTEGER
LongTypeHandler	java.lang.Long.long	数据库兼容的NUMERIC或LONG INTEGER

(1) 实现 `org.apache.ibatis.type.TypeHandler` 接口

(2) 继承 `org.apache.ibatis.type.BaseTypeHandler`

然后选择性将它映射到一个JDBC类型

通过这两种方式创建类型处理器可以处理不支持的或非标准的类型

例如：需求一个Java中的Date数据类型，想将它存到数据库的时候变成时间戳，取出来时转换成Java的Date，即为Java的Date类型和数据库的Varchar毫秒值字符串 之间的转换

9.4.1 开发四部曲

1. 定义转换类继承类 `BaseTypeHandler< T >`
2. 覆盖4个未实现的方法，其中 `setNonNullParameter` 为 Java程序设置数据到数据库的回调方法，`getNullableResult`为查询时 mysql的字符串类型转换成java的Type类型的方法
3. 在MyBatis核心配置文件中进行注册
4. 测试转换是否正确

9.4.2 测试Demo 实现用户生日信息的类型转换

在原有的表user上添加列 birthday

```
alter table user
add birthday bigint null;
```

类型转换类 `DateTypeHandler.java`

```

package com.uni.handler;

import org.apache.ibatis.type.BaseTypeHandler;
import org.apache.ibatis.type.JdbcType;

import java.sql.CallableStatement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

public class DateTypeHandler extends BaseTypeHandler<Date> {
    // 将 Java 转换成数据库需要的类型
    public void setNonNullParameter(PreparedStatement preparedStatement, int i, Date date, JdbcType jdbcType)
throws SQLException {
        long time = date.getTime();
        preparedStatement.setLong(i, time);
    }

    // 将数据库的类型 转换成 Java类型
    // String s 为数据表中要转换的名称这里是birthday
    // ResultSet 查询出的结果集
    public Date getNullableResult(ResultSet resultSet, String s) throws SQLException {
        // 获得 结果集中需要的数据 (long) 转换成 Date 类型 返回
        long aLong = resultSet.getLong(s);
        Date date = new Date(aLong);
        return date;
    }
    // 将数据库的类型 转换成 Java类型

    public Date getNullableResult(ResultSet resultSet, int i) throws SQLException {
        long aLong = resultSet.getLong(i);
        Date date = new Date(aLong);
        return date;
    }

    // 将数据库的类型 转换成 Java类型
    public Date getNullableResult(CallableStatement callableStatement, int i) throws SQLException {
        long aLong = callableStatement.getLong(i);
        Date date = new Date(aLong);
        return date;
    }
}

```

MyBatis配置文件 sqlMapConfig.xml

```

<!-- 注册类型处理器 -->
<typeHandlers>
    <typeHandler handler="com.uni.handler.DateTypeHandler"/>
</typeHandlers>

```

UserMapper.xml

```

<insert id = "save" parameterType="user">
    insert into user values(#{id}, #{username}, #{password}, #{birthday})
</insert>

<select id="findById" parameterType="int" resultType="user">
    select * from user where id = #{id}
</select>

```

测试类 MyBatisTest.java

```
@Test
public void testInsert() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    // 创建 user
    User user = new User();
    user.setUsername("abcde");
    user.setPassword("5201314");
    user.setBirthday(new Date());
    // 执行保存操作
    mapper.save(user);
    sqlSession.close();
}

@Test
public void testSelectById() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    User user = mapper.findById(6);
    System.out.println("user的birthday为: " + user.getBirthday());
    sqlSession.close();
}
```

查询功能:

MyBatisTest.java

9.5 plugins 标签 测试第三方分页助手插件

MyBatis 可以使用第三方的插件来对功能进行扩展，分页助手PageHelper是将分页的复杂操作进行封装，使用简单的方式即可获得分页的相关数据

1. 导入通用PageHelper的坐标
2. 在MyBatis核心配置文件中配置PageHelper插件
3. 测试分页数据获取

pom.xml

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>3.7.5</version>
</dependency>

<dependency>
    <groupId>com.github.jsqlparser</groupId>
    <artifactId>jsqlparser</artifactId>
    <version>0.9.1</version>
</dependency>
```

MyBatis配置文件 sqlMapConfig.xml

```
<!-- 配置分页助手插件 -->
<plugins>
    <plugin interceptor="com.github.pagehelper.PageHelper">
        <property name="dialect" value="mysql"/>
    </plugin>
</plugins>
```

测试类MyBatisTest.java

```
@Test
public void testFindAll() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    // 设置分页相关参数，当前页 + 每页显示的条数
    PageHelper.startPage(2, 2);
    List<User> userList = mapper.findAll();
    for (User user : userList) System.out.println(user);

    // 获得与分页相关的参数
    PageInfo<User> pageInfo = new PageInfo<User>(userList);
    System.out.println("当前页数: " + pageInfo.getPageNum());
    System.out.println("每页显示的条数: " + pageInfo.getPageSize());
    System.out.println("总条数: " + pageInfo.getTotal());
    System.out.println("总页数: " + pageInfo.getPages());
    System.out.println("上一页: " + pageInfo.getPrePage());
    System.out.println("下一页: " + pageInfo.getNextPage());
    System.out.println("是否为第一页: " + pageInfo.isIsFirstPage());
    System.out.println("是否为最后一页: " + pageInfo.isIsLastPage());
    sqlSession.close();
}
```

数据表User的内容：

id	username	password	birthday
2	zhangsan	123456789	<null>
3	lisi	abc	<null>
5	abc	520	<null>
6	abcde	5201314	1627792954973

测试结果：

```
User{id=5, username='abc', password='520', birthday=null}
User{id=6, username='abcde', password='5201314', birthday=Sun Aug 01 12:42:34 CST 2021}
当前页数: 2
每页显示的条数: 2
总条数: 4
总页数: 2
上一页: 1
下一页: 0
是否为第一页: false
是否为最后一页: true

进程已结束，退出代码为 0
```

9.6 知识小结

MyBatis 核心配置文件常用标签

标签名	作用
properties	加载外部的properties文件
typeAliases	设置类型别名
environments	数据源环境配置
typeHandlers	配置自定义类型处理器
plugins	配置MyBatis插件

10. MyBatis 多表操作

10.1 一对多 Demo

比如 用户表 和 订单表

一个用户有多个订单，一个订单只从属于一个用户

一对一查询需求：实现查询每个订单对应的用户信息

创建订单表其中的uid要添加user表id的外键约束：

```
create table orders(
    id int not null primary key auto_increment,
    ordertime bigint,
    total double,
    uid int,
    constraint uid foreign key (uid) references test.user(id)
)auto_increment = 1;

insert into orders(uid, ordertime, total) values(2, 123456, 15.00);
insert into orders(uid, ordertime, total) values(2, 1234567, 30.00);
insert into orders(uid, ordertime, total) values(3, 335653, 50.00);
insert into orders(uid, ordertime, total) values(3, 12516566, 50.00);
```

OrderMapper.java

```
package com.uni.dao;

import com.uni.domain.Order;

import java.util.List;

public interface OrderMapper {
    public List<Order> findAll();
}
```

实体 Order.java

```
package com.uni.domain;

import java.util.Date;

public class Order {
```

```

private int id;
private Date ordertime;
private double total;
// 当前订单 属于拿一个用户
private User user;

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public Date getOrdertime() {
    return ordertime;
}

public void setOrdertime(Date ordertime) {
    this.ordertime = ordertime;
}

public double getTotal() {
    return total;
}

public void setTotal(double total) {
    this.total = total;
}

public User getUser() {
    return user;
}

public void setUser(User user) {
    this.user = user;
}

@Override
public String toString() {
    return "Order{" +
        "id=" + id +
        ", ordertime=" + ordertime +
        ", total=" + total +
        ", user=" + user +
        '}';
}
}

```

类型转换类参考 9.4.2

OrderMapper.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.OrderMapper">
    <resultMap id="orderMap" type="order">
        <!-- 手动指定字段与实体属性的映射关系
            column: 数据表的字段名称

```

```

        property: 实体的属性名称

-->
<id column="oid" property="id"/>
<result column="ordertime" property="ordertime"/>
<result column="total" property="total" />
<result column="uid" property="user.id"/>
<result column="username" property="user.username"/>
<result column="password" property="user.password"/>
<result column="birthday" property="user.birthday"/>
</resultMap>
<select id="findAll" resultMap="orderMap">
    select *, o.id oid from orders o, user u where o.uid=u.id
</select>
</mapper>

```

MyBatis配置文件sqlMapConfig.xml

```

<?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>

    <!-- 通过properties标签加载外部properties文件-->
    <properties resource="jdbc.properties"></properties>
    <!-- 设置对象别名 -->
    <typeAliases>
        <typeAlias type="com.uni.domain.User" alias="user"/>
        <typeAlias type="com.uni.domain.Order" alias="order"/>
    </typeAliases>
    <!-- 注册类型处理器 -->
    <typeHandlers>
        <typeHandler handler="com.uni.handler.DateTypeHandler"/>
    </typeHandlers>

    <!-- 配置分页助手插件 -->
    <plugins>
        <plugin interceptor="com.github.pagehelper.PageHelper">
            <property name="dialect" value="mysql"/>
        </plugin>
    </plugins>
    <!-- 数据源 环境 -->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="com/uni/mapper/UserMapper.xml"></mapper>
        <mapper resource="com/uni/mapper/OrderMapper.xml"></mapper>
    </mappers>
</configuration>

```

测试类 MyBatisTest.java

```

@Test
public void testfindAll() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);
    OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);
    List<Order> orderList = mapper.findAll();
    for(Order order : orderList){
        System.out.println(order);
    }
    sqlSession.close();
}

```

表user

	id	username	password	birthday
1	2	zhangsan	123456789	<null>
2	3	lisi	abc	<null>
3	5	abc	520	<null>
4	6	abcde	5201314	1627792954973

表orders

	id	ordertime	total	uid
1	1	123456	15	2
2	2	1234567	30	2
3	3	335653	50	3
4	4	12516566	50	3

测试结果：

```

Order{id=1, ordertime=Thu Jan 01 08:02:03 CST 1970, total=15.0, user=User{id=2, username='zhangsan', password='123456789', birthday=null}}
Order{id=2, ordertime=Thu Jan 01 08:20:34 CST 1970, total=30.0, user=User{id=2, username='zhangsan', password='123456789', birthday=null}}
Order{id=3, ordertime=Thu Jan 01 08:05:35 CST 1970, total=50.0, user=User{id=3, username='lisi', password='abc', birthday=null}}
Order{id=4, ordertime=Thu Jan 01 11:28:36 CST 1970, total=50.0, user=User{id=3, username='lisi', password='abc', birthday=null}}

```

UserMapper.xml 中 resultMap标签的第二种写法

```

<resultMap id="orderMap" type="order">
    <!-- 手动指定字段与实体属性的映射关系
        column: 数据表的字段名称
        property: 实体的属性名称
    -->
    <id column="oid" property="id"/>
    <result column="ordertime" property="ordertime"/>
    <result column="total" property="total" />
    <!--
        property: 当前实体 (order) 中的属性名称 (private User user)
        javaType: 当前实体 (order) 中的属性的类型 (User)
    -->
    <association property="user" javaType="user">
        <id column="uid" property="id"/>
        <id column="username" property="username"/>
        <id column="password" property="password"/>
    </association>

```

```
<id column="birthday" property="birthday"/>
</association>
</resultMap>
```

10.2 一对多 Demo

同样是以用户表和订单表为例

Demo为查询每个用户对应的所有订单信息

实体类 User.java

```
package com.uni.domain;

import java.util.Date;
import java.util.List;

public class User{
    private int id;
    private String username;
    private String password;
    private Date birthday;
    // 描述的是当前用户 存在哪些订单
    private List<Order> orderList;

    public List<Order> getOrderList() {
        return orderList;
    }

    public void setOrderList(List<Order> orderList) {
        this.orderList = orderList;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }
}
```

```

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", birthday=" + birthday +
        ", orderList=" + orderList +
        '}';
}
}

```

实体类 Order.java

```

package com.uni.domain;

import java.util.Date;

public class Order {
    private int id;
    private Date ordertime;
    private double total;
    // 当前订单 属于拿一个用户
    private User user;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Date getOrdertime() {
        return ordertime;
    }

    public void setOrdertime(Date ordertime) {
        this.ordertime = ordertime;
    }

    public double getTotal() {
        return total;
    }

    public void setTotal(double total) {
        this.total = total;
    }

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }
}

```

```

@Override
public String toString() {
    return "Order{" +
        "id=" + id +
        ", ordertime=" + ordertime +
        ", total=" + total +
        ", user=" + user +
        '}';
}

```

UserMapper.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.UserMapper">
    <resultMap id="userMap" type="user">
        <id column="uid" property="id"></id>
        <result column="username" property="username"/>
        <result column="password" property="password"/>
        <result column="birthday" property="birthday"/>
        <!-- 配置集合信息
            property: 集合名称
            ofType: 当前集合中的数据类型
        -->
        <collection property="orderList" ofType="order">
            <!-- 封装 order的数据-->
            <id column="oid" property="id"/>
            <id column="ordertime" property="ordertime"/>
            <id column="total" property="total"/>
        </collection>
    </resultMap>
    <select id="findAll" resultMap="userMap">
        select *, o.id oid from user u, orders o where u.id = u.id
    </select>
</mapper>

```

测试类 MapBatisTest.java

```

@Test
public void testSelectOrderByID() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    List<User> userList = mapper.findAll();
    for (User user : userList) {
        System.out.println(user);
    }
    sqlSession.close();
}

```

10.3 多对多 Demo

```

create table role(
    id int not null primary key auto_increment,
    roleName VARCHAR(30) not null ,
    roleDesc VARCHAR(30) not null
)auto_increment=1

insert into role(roleName, roleDesc) values ('班长', '负责班级通知工作');
insert into role(roleName, roleDesc) values ('纪律委员', '负责班级考勤工作');
insert into role(roleName, roleDesc) values ('心理委员', '负责心理指导工作');
insert into role(roleName, roleDesc) values ('学习委员', '负责学习指导工作');

```

创建 用户-身份 映射表 user_role

```

create table user_role(
    userid int,
    roleid int,
    constraint userid foreign key (userid) references test.user(id),
    constraint roleid foreign key (roleid) references test.role(id)
)

insert into user_role (userid, roleid) VALUES (2, 1);
insert into user_role (userid, roleid) VALUES (3, 2);
insert into user_role (userid, roleid) VALUES (3, 3);
insert into user_role (userid, roleid) VALUES (5, 4);

```

实体类 User.java

```

package com.uni.domain;

import java.util.Date;
import java.util.List;

public class User{
    private int id;
    private String username;
    private String password;
    private Date birthday;
    // 描述的是当前用户 存在哪些订单
    private List<Order> orderList;
    // 描述的是当前用户 具备哪些角色
    private List<Role> roleList;

    public List<Role> getRoleList() {
        return roleList;
    }

    public void setRoleList(List<Role> roleList) {
        this.roleList = roleList;
    }

    public List<Order> getOrderList() {
        return orderList;
    }

    public void setOrderList(List<Order> orderList) {

```

```

    this.orderList = orderList;
}

public Date getBirthday() {
    return birthday;
}

public void setBirthday(Date birthday) {
    this.birthday = birthday;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", birthday=" + birthday +
        ", roleList=" + roleList +
        '}';
}
}

```

实体类 Role.java

```

package com.uni.domain;

public class Role {
    private int id;
    private String roleName;
    private String roleDesc;

    public int getId() {
        return id;
    }
}

```

```

public void setId(int id) {
    this.id = id;
}

public String getRoleName() {
    return roleName;
}

public void setRoleName(String roleName) {
    this.roleName = roleName;
}

public String getRoleDesc() {
    return roleDesc;
}

public void setRoleDesc(String roleDesc) {
    this.roleDesc = roleDesc;
}

@Override
public String toString() {
    return "Role{" +
        "id=" + id +
        ", roleName='" + roleName + '\'' +
        ", roleDesc='" + roleDesc + '\'' +
        '}';
}
}

```

配置类 UserMapper.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.uni.dao.UserMapper">
    <resultMap id="userRoleMap" type="user">
        <!-- user 的信息-->
        <id column="userid" property="id"/>
        <result column="username" property="username"/>
        <result column="password" property="password"/>
        <result column="birthday" property="birthday"/>
        <!-- user 内部的roleList信息 -->
        <collection property="roleList" ofType="role">
            <id column="roleid" property="id"/>
            <id column="rolename" property="rolename"/>
            <id column="roledesc" property="roledesc"/>
        </collection>
    </resultMap>

    <select id="findUserAndRoleAll" resultMap="userRoleMap">
        select * from user u, user_role ur, role r where u.id = ur.userid and ur.roleid = r.id
    </select>
</mapper>

```

MyBatis 配置文件 sqlMapConfig.xml

```

<!-- 设置对象别名 -->
<typeAliases>
    <typeAlias type="com.uni.domain.User" alias="user"/>
    <typeAlias type="com.uni.domain.Order" alias="order"/>
    <typeAlias type="com.uni.domain.Role" alias="role"/>
</typeAliases>

```

测试类MyBatisTest.java

```

@Test
public void testSelectRoleById() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = build.openSession(true);
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    List<User> userList = mapper.findUserAndRoleAll();
    for (User user : userList) {
        System.out.println(user);
    }
    sqlSession.close();
}

```

表user

	id	username	password	birthday
1	2	zhangsan	123456789	<null>
2	3	lisi	abc	<null>
3	5	abc	520	<null>
4	6	abcde	5201314	1627792954973

表role

	id	roleName	roleDesc
1	1	班长	负责班级通知工作
2	2	纪律委员	负责班级考勤工作
3	3	心理委员	负责心理指导工作
4	4	学习委员	负责学习指导工作

表user_role

	userid	roleid
1	2	1
2	3	2
3	3	3
4	5	4

运行结果：

```
User{id=2, username='zhangsan', password='123456789', birthday=null, roleList=[Role{id=1, roleName='班长', roleDesc='负责班级通知工作'}]}
User{id=3, username='lisi', password='abc', birthday=null, roleList=[Role{id=2, roleName='纪律委员', roleDesc='负责班级考勤工作'}, Role{id=3, roleName='心理委员', roleDesc='负责心理指导工作'}]}
User{id=5, username='abc', password='520', birthday=null, roleList=[Role{id=4, roleName='学习委员', roleDesc='负责学习指导工作'}]}
```

10.4 知识小结

MyBatis多表配置方式

一对多配置：

一对多配置： +

多对多配置： +

11.MyBatis 的注解开发

常见注解

注解	作用
@Insert	新增
@Update	更新
@Delete	删除
@Select	查询
@Result	结果集封装
@Results	可以和@Result一起使用，封装多个结果集
@One	实现一对结果集封装
@Many	实现一对多结果集封装

11.1 MyBatis 注解实现CURD Demo

测试表参考之前的Demo部分

本次测试Demo没有Mapper配置文件

MyBatis 配置文件 sqlMapConfig.xml

```
<?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>
    <!-- 通过properties标签加载外部properties文件-->
    <properties resource="jdbc.properties"></properties>
    <!-- 设置对象别名 -->
    <typeAliases>
        <typeAlias type="com.uni.domain.User" alias="user"/>
    </typeAliases>
    <!-- 注册类型处理器 -->
    <typeHandlers>
        <typeHandler handler="com.uni.handler.DateTypeHandler"/>
    </typeHandlers>
    <!-- 配置分页助手插件 -->
```

```

<plugins>
    <plugin interceptor="com.github.pagehelper.PageHelper">
        <property name="dialect" value="mysql"/>
    </plugin>
</plugins>
<!-- 数据源 环境 -->
<environments default="developement">
    <environment id="developement">
        <transactionManager type="JDBC"></transactionManager>
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>

<!-- 加载映射关系 -->
<mappers>
    <!-- 指定接口所在的包 -->
    <package name="com.uni.dao"/>
</mappers>

</configuration>

```

接口类 UserMapper.java

```

package com.uni.dao;
import com.uni.domain.User;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import java.util.List;
public interface UserMapper {
    @Insert("insert into user values (#{id}, #{password}, #{birthday})")
    public void save(User user);

    @Update("update user set username = #{username}, password=#{password} where id=#{id}")
    public void update(User user);

    @Delete("delete from user where id=#{id}")
    public void delete(int id);

    @Select("select * from user where id=#{id}")
    public User findById(int id);

    @Select("select * from user")
    public List<User> selectAll();
}

```

测试类 MyBatisTest.java

```

import com.uni.dao.UserMapper;
import com.uni.domain.User;

```

```

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

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

public class MyBatisTest {
    private UserMapper mapper;
    @Before
    public void before() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        mapper = sqlSession.getMapper(UserMapper.class);
    }
    @Test
    public void testSelectAll(){
        List<User> userList = mapper.selectAll();
        for (User user : userList) {
            System.out.println(user);
        }
    }
}

```

测试类中仅测试了查询所有数据的方法，其他方法效果类似，略过。

11.2 注解实现复杂映射开发

注解	说明
@Results	代替的是标签，该注解中可使用单个@Result注解，也可使用@Result集合。
@Result	代替了 标签和 标签
@One(一对一)	代替了标签，是多表查询的关键，在注解中用来指定子查询返回单一对象。
@Many(多对一)	代替了标签，是多表查询关键，在注解中用来指定子查询返回对象集合。

@Results注解使用格式

- (1) @Results{@Result(), @Result()})
- (2) @Results(@Result())

@Result注解属性介绍

- column: 数据库列名
- property: 需要装配的属性名
- one: 需要使用的@One注解 (@Result (one=@One())))
- many: 需要使用的@Many注解 (@Result (many = @many) ())

@One注解属性介绍

- select: 用来指定多表查询的 sqlmapper
- 使用格式: @Result(column="", property="", one=@One(select=""))

@Many注解使用格式

```
@Result(property="", column="", many=@Many(select=""))
```

11.2.1 一对一 Demo

同 10.1部分，实现查询每个订单对应的用户信息

实体类Order.java

```
package com.uni.domain;

import java.util.Date;

public class Order {
    private int id;
    private Date ordertime;
    private double total;
    // 当前订单 属于拿一个用户
    private User user;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Date getOrdertime() {
        return ordertime;
    }

    public void setOrdertime(Date ordertime) {
        this.ordertime = ordertime;
    }

    public double getTotal() {
        return total;
    }

    public void setTotal(double total) {
        this.total = total;
    }

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    @Override
    public String toString() {
        return "Order{" +
            "id=" + id +
            ", ordertime=" + ordertime +
            ", total=" + total +
            ", user=" + user +
            '}';
    }
}
```

```
}
```

接口类OrderMapper.java

```
package com.uni.dao;

import com.uni.domain.Order;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface OrderMapper {
    @Select("select *,o.id oid from orders o, user u where o.uid=u.id")
    @Results({
        @Result(column = "oid", property = "id"),
        @Result(column = "ordertime", property = "ordertime"),
        @Result(column = "total", property = "total"),
        @Result(column = "uid", property = "user.id"),
        @Result(column = "username", property = "user.username"),
        @Result(column = "password", property = "user.password"),
    })
    public List<Order> findAll();
}
```

测试类 MyBatisTest.java

```
import com.uni.dao.OrderMapper;
import com.uni.domain.Order;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Before;
import org.apache.ibatis.io.Resources;
import org.junit.Test;

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

public class MyBatisTest {
    private OrderMapper mapper;
    @Before
    public void before() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        mapper = sqlSession.getMapper(OrderMapper.class);
    }
    @Test
    public void testSelectOrderAll(){
        List<Order> orderList = mapper.findAll();
        for (Order order : orderList) {
            System.out.println(order);
        }
    }
}
```

接口类的第二种写法，引用类似于 标签 去遍历一个集合对象

OrderMapper.java

```
public interface OrderMapper {
    @Select("select * from orders")
    @Results({
        @Result(column = "oid", property = "id"),
        @Result(column = "ordertime", property = "ordertime"),
        @Result(column = "total", property = "total"),
        @Result(
            property = "user", // 要封装的属性名称
            column = "uid", // 根据哪个字段去查询user表的数据
            javaType = User.class, // 要封装的实体类型
            // select 属性表示查询哪一个接口的方法获得数据
            one = @One(select = "com.uni.dao.UserMapper.findById")
        )
    })
    public List<Order> findAll();
}
```

11.2.2 一对多 Demo

和10.2部分一样，查询每个用户对应的所有订单信息

实体类 User.java

```
package com.uni.domain;

import java.util.Date;
import java.util.List;

public class User{
    private int id;
    private String username;
    private String password;
    private Date birthday;
    private List<Order> orderList;
    public List<Order> getOrderList() {
        return orderList;
    }

    public void setOrderList(List<Order> orderList) {
        this.orderList = orderList;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public int getId() {
        return id;
    }
}
```

```

public void setId(int id) {
    this.id = id;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", birthday=" + birthday +
        ", orderList=" + orderList +
        '}';
}
}

```

接口类 UserMapper.java

```

package com.uni.dao;
import com.uni.domain.User;
import org.apache.ibatis.annotations.*;

import java.util.List;
public interface UserMapper {
    @Insert("insert into user values (#{id}, #{password}, #{birthday})")
    public void save(User user);

    @Update("update user set username = #{username}, password=#{password} where id=#{id}")
    public void update(User user);

    @Delete("delete from user where id=#{id}")
    public void delete(int id);

    @Select("select * from user where id=#{id}")
    public User findById(int id);

    @Select("select * from user")
    public List<User> selectAll();

    @Select("select * from user")
    @Results({
        @Result(id=true, column = "id", property = "id"),

```

```

        @Result(column = "username", property = "username"),
        @Result(column = "password", property = "password"),
        @Result(
            property = "orderList",
            column = "id",
            javaType = List.class,
            many = @Many(select = "com.uni.dao.OrderMapper.findByUid")
        )
    })
    public List<User> findUserAndOrderAll();
}

```

接口类 OrderMapper.java

```

package com.uni.dao;

import com.uni.domain.Order;
import com.uni.domain.User;
import org.apache.ibatis.annotations.One;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface OrderMapper {
    @Select("select * from orders")
    @Results({
        @Result(column = "oid", property = "id"),
        @Result(column = "ordertime", property = "ordertime"),
        @Result(column = "total", property = "total"),
        @Result(
            property = "user", // 要封装的属性名称
            column = "uid", // 根据哪个字段去查询user表的数据
            javaType = User.class, // 要封装的实体类型
            // select 属性表示查询哪一个接口的方法获得数据
            one = @One(select = "com.uni.dao.UserMapper.findById")
        )
    })
    public List<Order> findAll();

    @Select("select * from orders where uid=#{uid}")
    public List<Order> findByUid(int uid);
}

```

测试类 MyBatisTest.java

```

import com.uni.dao.OrderMapper;
import com.uni.dao.UserMapper;
import com.uni.domain.Order;
import com.uni.domain.User;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Before;
import org.apache.ibatis.io.Resources;
import org.junit.Test;

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

```

```

public class MyBatisTest {
    private UserMapper mapper;
    @Before
    public void before() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        mapper = sqlSession.getMapper(UserMapper.class);
    }
    @Test
    public void testSelectOrderAll(){
        List<User> userList = mapper.findUserAndOrderAll();
        for (User user : userList) {
            System.out.println(user);
        }
    }
}

```

11.2.3 多对多 Demo

查询每个用户通过用户角色映射表查询对应的所有角色

实体表 User.java

```

package com.uni.domain;

import java.util.Date;
import java.util.List;

public class User{
    private int id;
    private String username;
    private String password;
    private Date birthday;
    private List<Role> roleList;

    public List<Role> getRoleList() {
        return roleList;
    }

    public void setRoleList(List<Role> roleList) {
        this.roleList = roleList;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}

```

```

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", birthday=" + birthday +
        ", roleList=" + roleList +
        '}';
}
}

```

接口类 RoleMapper.java

```

package com.uni.dao;

import com.uni.domain.Role;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface RoleMapper {
    @Select("select * from user_role ur, role r where ur.roleid=r.id and ur.userid= #{uid}")
    List<Role> findByUid(int uid);
}

```

接口类 UserMapper.java

```

package com.uni.dao;
import com.uni.domain.User;
import org.apache.ibatis.annotations.*;

import java.util.List;
public interface UserMapper {
    @Insert("insert into user values (#{id}, #{password}, #{birthday})")
    public void save(User user);

    @Update("update user set username = #{username}, password=#{password} where id=#{id}")
    public void update(User user);

    @Delete("delete from user where id=#{id}")
    public void delete(int id);

    @Select("select * from user where id=#{id}")
}

```

```

public User findById(int id);

@Select("select * from user")
public List<User> selectAll();

@Select("select * from user")
@Results({
    @Result(id = true, column = "id", property = "id"),
    @Result(column = "username", property = "username"),
    @Result(column = "password", property = "password"),
    @Result(
        property = "roleList",
        column = "id",
        javaType = List.class,
        many = @Many(select = "com.uni.dao.RoleMapper.findByUid")
    )
})
public List<User> findUserAndRoleAll();
}

```

测试类 MyBatisTest.java

```

import com.uni.dao.OrderMapper;
import com.uni.dao.UserMapper;
import com.uni.domain.Order;
import com.uni.domain.User;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Before;
import org.apache.ibatis.io.Resources;
import org.junit.Test;

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

public class MyBatisTest {
    private UserMapper mapper;
    @Before
    public void before() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession(true);
        mapper = sqlSession.getMapper(UserMapper.class);
    }
    @Test
    public void test(){
        List<User> userList = mapper.findUserAndRoleAll();
        for (User user : userList) {
            System.out.println(user);
        }
    }
}

```

