

1- Spring概述

1.1 基本概念

1.1.1 基本简介

- Spring : 春天 --->给软件行业带来了春天
- 2002年, Rod Jahnson首次推出了Spring框架雏形interface21框架。
- 2004年3月24日, Spring框架以interface21框架为基础, 经过重新设计, 发布了1.0正式版。
- Spring理念:使现有技术更加实用. 本身就是一个大杂烩, 整合现有的框架技术。

官网: <http://spring.io/>

官方下载地址: <https://repo.spring.io/libs-release-local/org/springframework/spring/>

GitHub: <https://github.com/spring-projects>

1.1.2 优点和总结

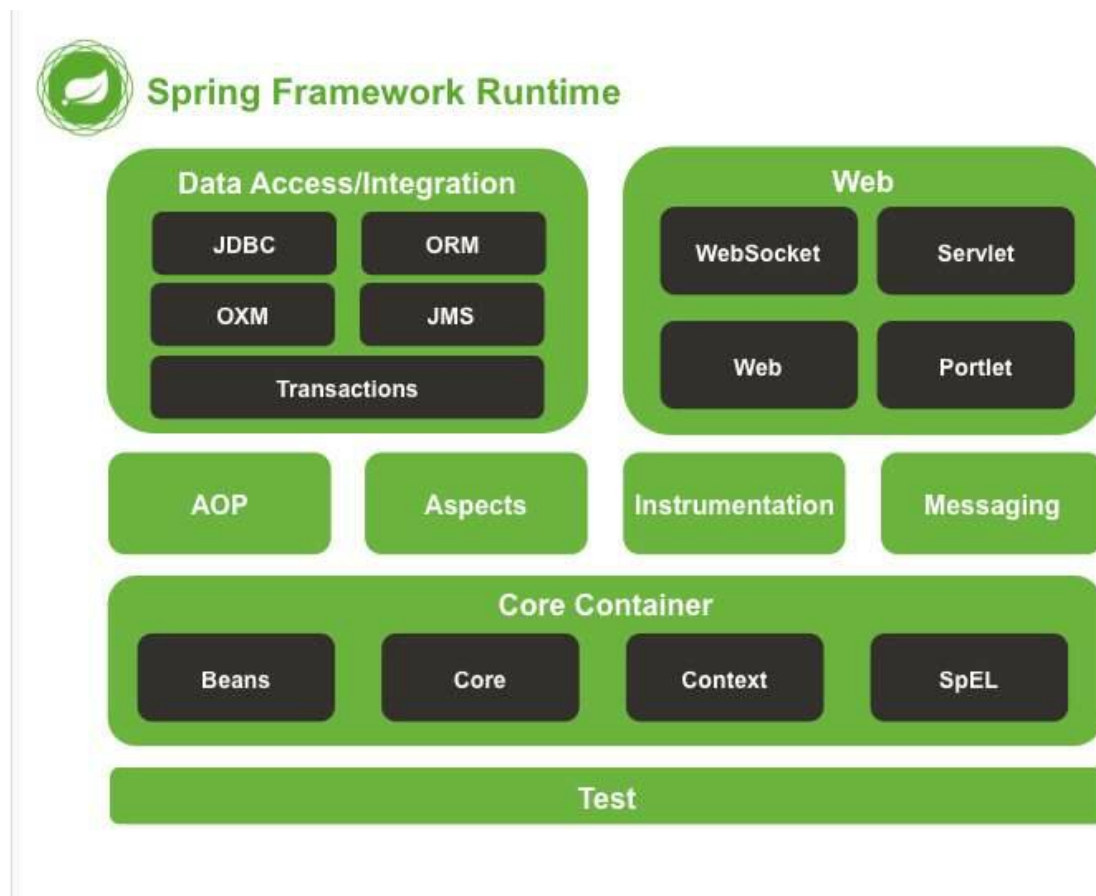
优点

- Spring是一个开源免费的框架, 容器。
- Spring是一个轻量级的框架, 非侵入式的。
- 控制反转 IoC, 面向切面 Aop。对事物的支持, 对框架的支持

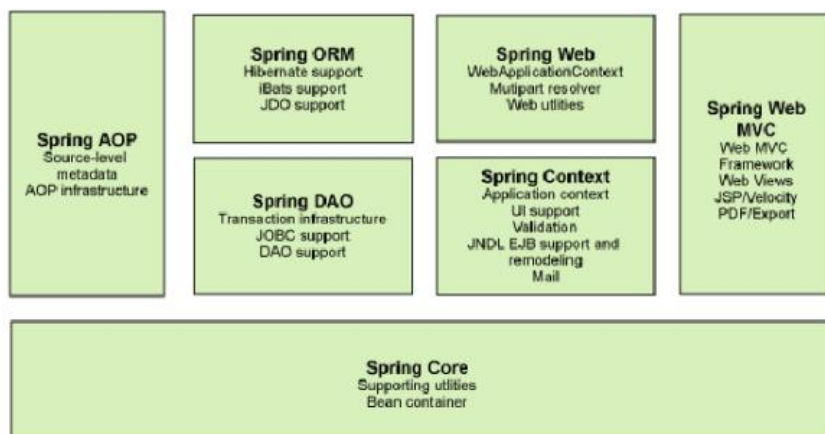
总结

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器 (框架)

1.1.3 框架体系结构



Spring 框架是一个分层架构，由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式。



组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。 每个模块的功能如下：

核心容器

- 核心容器提供 Spring 框架的基本功能。**核心容器的主要组件是 BeanFactory**，它是工厂模式的实现。
- **BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。**

Spring 上下文

- Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。

Spring AOP

- 通过配置管理特性Spring AOP 模块直接将面向切面的编程功能，集成到了 Spring框架中。可以很容易地使 Spring 框架管理任何支持 AOP的对象。
- **Spring AOP 模块为基于Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP不用依赖组件，就可以将声明性事务管理集成到应用程序中。**

Spring DAO

- JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。
- 异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量。
- Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

Spring ORM

- Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。
- 所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

Spring Web 模块

- Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。
- Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。

Spring MVC 框架：

- MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。

- 通过策略接口，MVC 框架变成高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText和 POI。

1.2 JDBC 实现

1.2.1 SQL数据表

```
-- 创建数据库
create database spring;

-- 显示数据库
show databases;

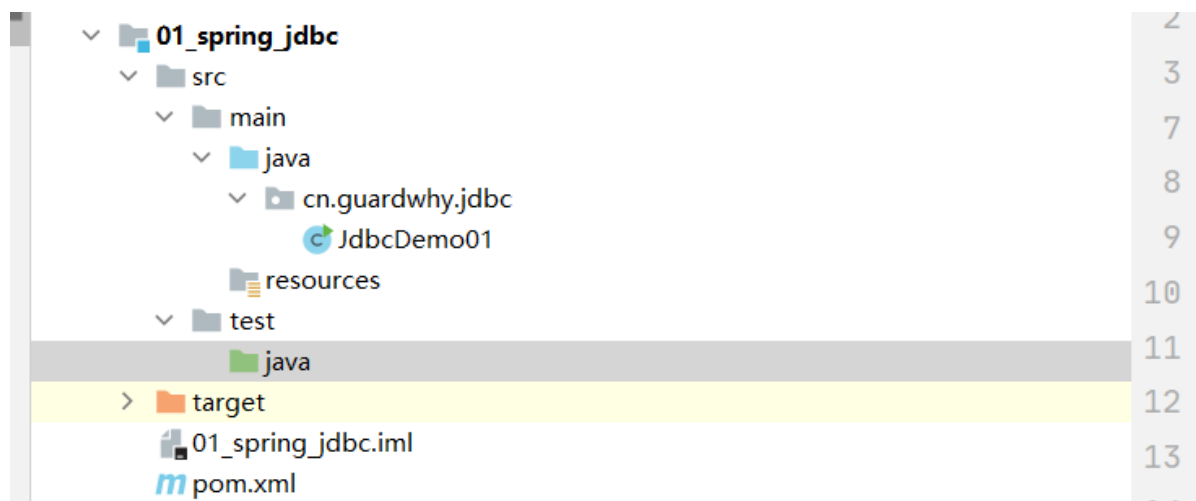
-- 使用数据库
use spring;

-- 创建客户表
CREATE TABLE `cst_customer` (
  `cust_id` bigint(32) NOT NULL AUTO_INCREMENT COMMENT '客户编号(主键)',
  `cust_name` varchar(32) NOT NULL COMMENT '客户名称(公司名称)',
  `cust_source` varchar(32) DEFAULT NULL COMMENT '客户信息来源',
  `cust_industry` varchar(32) DEFAULT NULL COMMENT '客户所属行业',
  `cust_level` varchar(32) DEFAULT NULL COMMENT '客户级别',
  `cust_address` varchar(128) DEFAULT NULL COMMENT '客户联系地址',
  `cust_phone` varchar(64) DEFAULT NULL COMMENT '客户联系电话',
  PRIMARY KEY (`cust_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

-- 添加客户数据
INSERT INTO `cst_customer` VALUES ('1', '美团外卖', '网络营销', '互联网', '普通客户', '北京市海淀区', '7758258');
INSERT INTO `cst_customer` VALUES ('2', '360公司', '网络安全', '互联网', '普通客户', '北京市朝阳区', '0208888887');
INSERT INTO `cst_customer` VALUES ('3', '百度', '网络搜索', '互联网', '普通客户', '北京朝阳区', '389056');
INSERT INTO `cst_customer` VALUES ('4', '小米', '手机制造', '互联网+', 'VIP', '武汉光谷', '2267890');
INSERT INTO `cst_customer` VALUES ('5', '腾讯', '社交', '互联网', 'VIP', '深圳市南山区', '123456');
INSERT INTO `cst_customer` VALUES ('6', '华为', '电视广告', '高科技制造业', 'VIP', '深圳龙岗', '033567');

-- 查询数据库
select * from cst_customer;
```

1.2.2 项目目录



导入父依赖 (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>cn.guardwhy</groupId>
    <artifactId>Spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <!--模块-->
    <modules>
        <module>01_spring_jdbc</module>
    </modules>

    <properties>
        <!-- spring版本 -->
        <spring.version>5.2.9.RELEASE</spring.version>
        <!-- mysql版本 -->
        <mysql.version>5.1.30</mysql.version>
    </properties>

    <dependencies>
        <!--spring 版本-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!--测试依赖-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13</version>
        </dependency>
        <!-- mysql数据库依赖 -->
        <dependency>
            <groupId>mysql</groupId>
```

```

        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
    </dependency>
    <!--lombok插件-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.16</version>
    </dependency>
</dependencies>
</project>

```

倒入子依赖

```

<?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">

    <parent>
        <artifactId>Spring</artifactId>
        <groupId>cn.guardwhy</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>01_spring_jdbc</artifactId>

</project>

```

1.2.3 测试代码

存在问题: 代码的耦合度太高了

```

package cn.guardwhy.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class JdbcDemo01 {
    public static void main(String[] args) {
        Connection connection = null;
        PreparedStatement psmt = null;
        ResultSet rs = null;
        try {
            // 1.加载驱动
            Class.forName("com.mysql.jdbc.Driver");
            // 2.创建数据库链接对象
            connection = DriverManager.getConnection("jdbc:mysql:///spring",
"root", "root");
            // 3.定义sql语句
            String sql = "select * from cst_customer";
            // 4.创建Statement语句对象

```

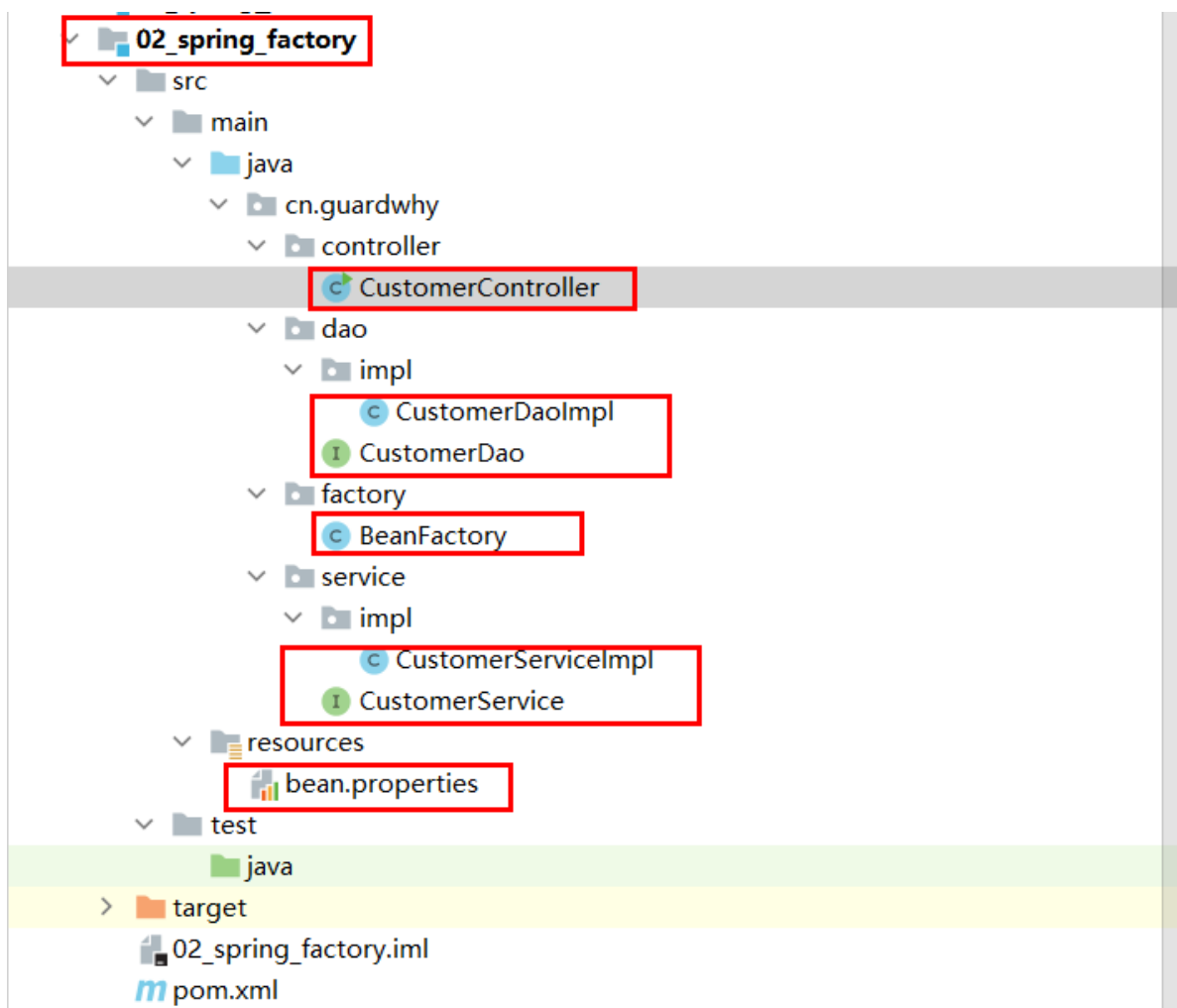
```

        pstmt = connection.prepareStatement(sql);
        // 5. 执行操作
        rs = pstmt.executeQuery();
        // 6. 处理结果集
        while (rs.next()){
            System.out.println("客户Id:" + rs.getInt("cust_id") + ", 客户名称:"
+ rs.getString("cust_name"));
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 7. 释放资源
        try {
            if(rs != null) rs.close();
            if(pstmt != null) pstmt.close();
            if(connection != null) connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
}

```

1.3 工厂模式解耦

1.3.1 项目目录



1.3.2 代码实现

配置文件 (bean.properties)

```
CUSTOMERDAO=cn.guardwhy.dao.impl.CustomerDaoImpl
CUSTOMERSERVICE=cn.guardwhy.service.impl.CustomerServiceImpl
```

BeanFactory代码

```
package cn.guardwhy.factory;

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

/**
 * 工厂类实现
 */
public class BeanFactory {
    // 1.声明一个私有的工厂类对象引用
    private static BeanFactory beanFactory;
    // 2.将构造方法私有化
    private BeanFactory(){

    }
    // 3.通过静态代码块初始化
    // 定义Properties
    private static Properties prop;
    static {
        // 创建工厂类对象
        beanFactory = new BeanFactory();
        // 创建prop对象,加载属性配置文件
        prop = new Properties();
        InputStream inputStream =
BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");
        try {
            prop.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("加载属性资源文件,发生异常:" + e.getMessage());
        }
    }

    // 4.提供一个公有的、静态的方法,获取工厂类对象引用
    public static BeanFactory getBeanFactory(){
        return beanFactory;
    }

    /**
     * 1.通过配置文件,将要创建的目标对象的类型信息,进行配置
     * 2.在工厂类中加载配置文件,通过反射技术实现运行时加载,并创建目标对象
     */
    public Object getBean(String beanName){
        // 目标对象
        Object result = null;
        // 获取类路径
        String className = prop.getProperty(beanName);
```

```

        // 反射技术创建对象
        try {
            result = Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("运行时创建对象,发生异常:" + className);
        }
        return result;
    }
}

```

持久层(Dao层)

dao层接口

```

package cn.guardwhy.dao;

/**
 * 客户dao接口
 */
public interface CustomerDao {

    /**
     * 保存客户操作
     */
    void saveCustomer();
}

```

dao实现类

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {

    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

}

```

业务层(Service)

service接口

```

package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {

    /**
     * 保存客户操作
     */
    void saveCustomer();
}

```


service实现类

```
package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.factory.BeanFactory;
import cn.guardwhy.service.CustomerService;
/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    // 从工厂类获取客户dao对象
    private CustomerDao customerDao = (CustomerDao)
BeanFactory.getBeanFactory().getBean("CUSTOMERDAO");

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}
```

表现层(Controller)

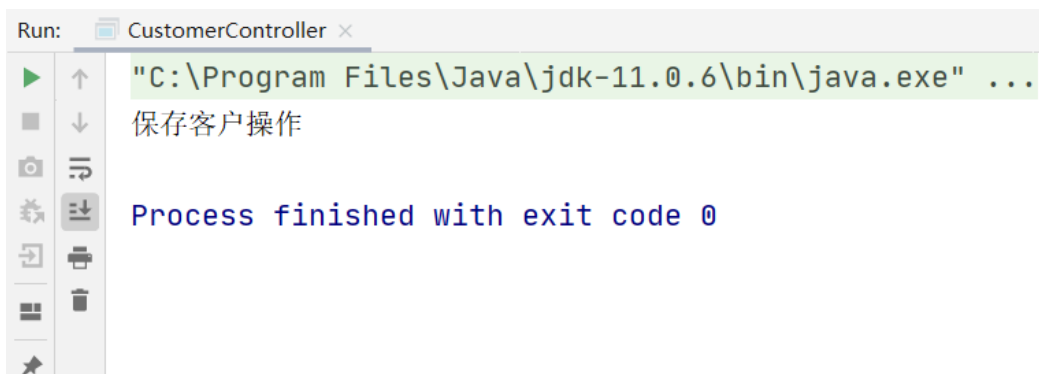
Controller

```
package cn.guardwhy.controller;

import cn.guardwhy.factory.BeanFactory;
import cn.guardwhy.service.CustomerService;

public class CustomerController {
    public static void main(String[] args) {
        // 从工厂类获取客户端service对象
        CustomerService customerService = (CustomerService)
BeanFactory.getBeanFactory().getBean("CUSTOMERSERVICE");
        // 保存客户操作
        customerService.saveCustomer();
    }
}
```

执行结果



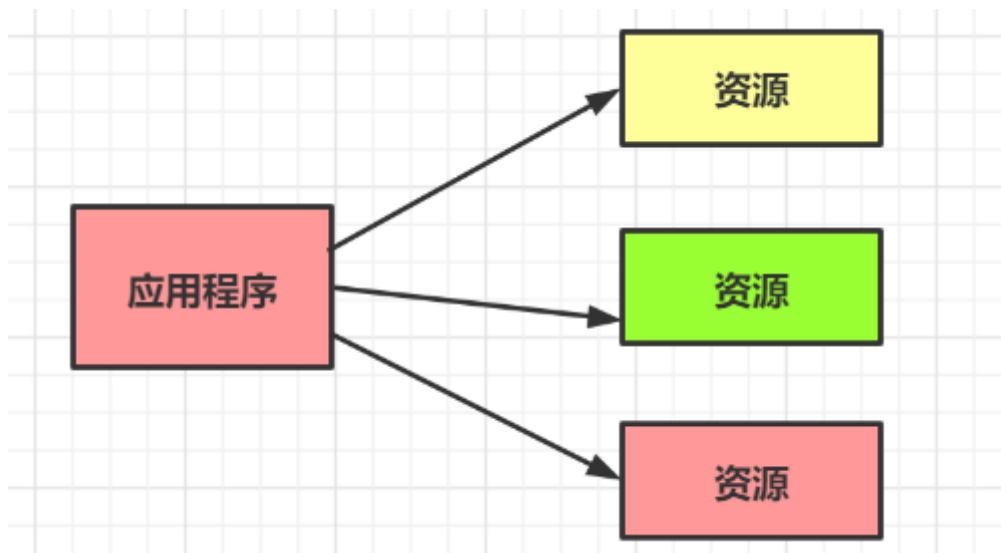
1.3.3 工厂模式解耦

工厂就是负责创建对象，并且把对象放到容器中。在实际使用的时候，帮助我们从容器获取指定的对象。此时获取对象的方式发生了改变。

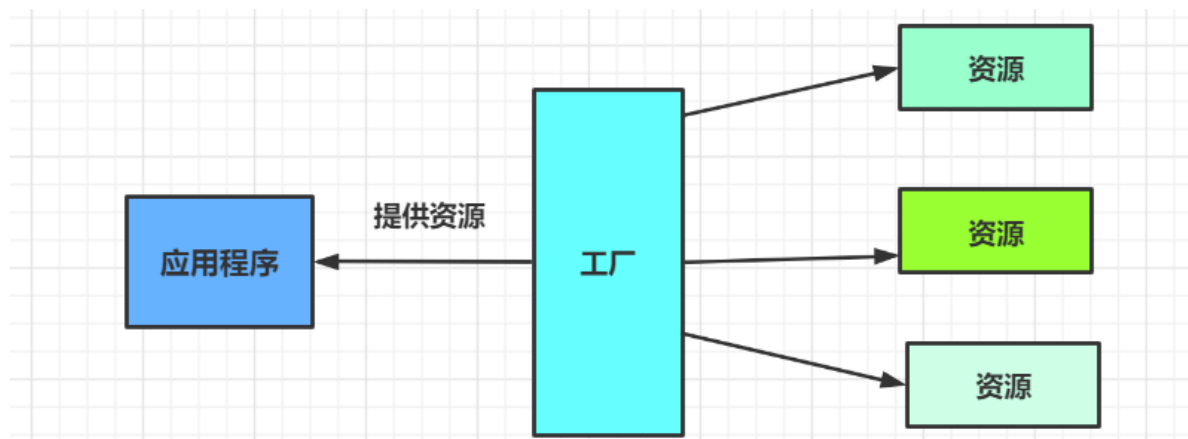
原来获取对象时，都是采用new的方式，是主动获取。现在我们获取对象时，找工厂要，由工厂创建并且提供给，是被动接收。

结论：这种获取对象方式的转变（由原来主动获取，到现在被动接收），我们称它为控制反转。控制反转，即IOC（Inversion Of Control）。

传统获取对象的方式：



通过工厂模式获取对象的方式：



2-Spring-入门案例(XML)

2.1 IOC的本质

控制反转IoC(Inversion of Control)，是一种设计思想，DI(依赖注入)是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：获得依赖对象的方式反转了。



IoC是Spring框架的核心内容，使用多种方式完美的实现了IoC，可以使用XML配置，也可以使用注解，新版本的Spring也可以零配置实现IoC。

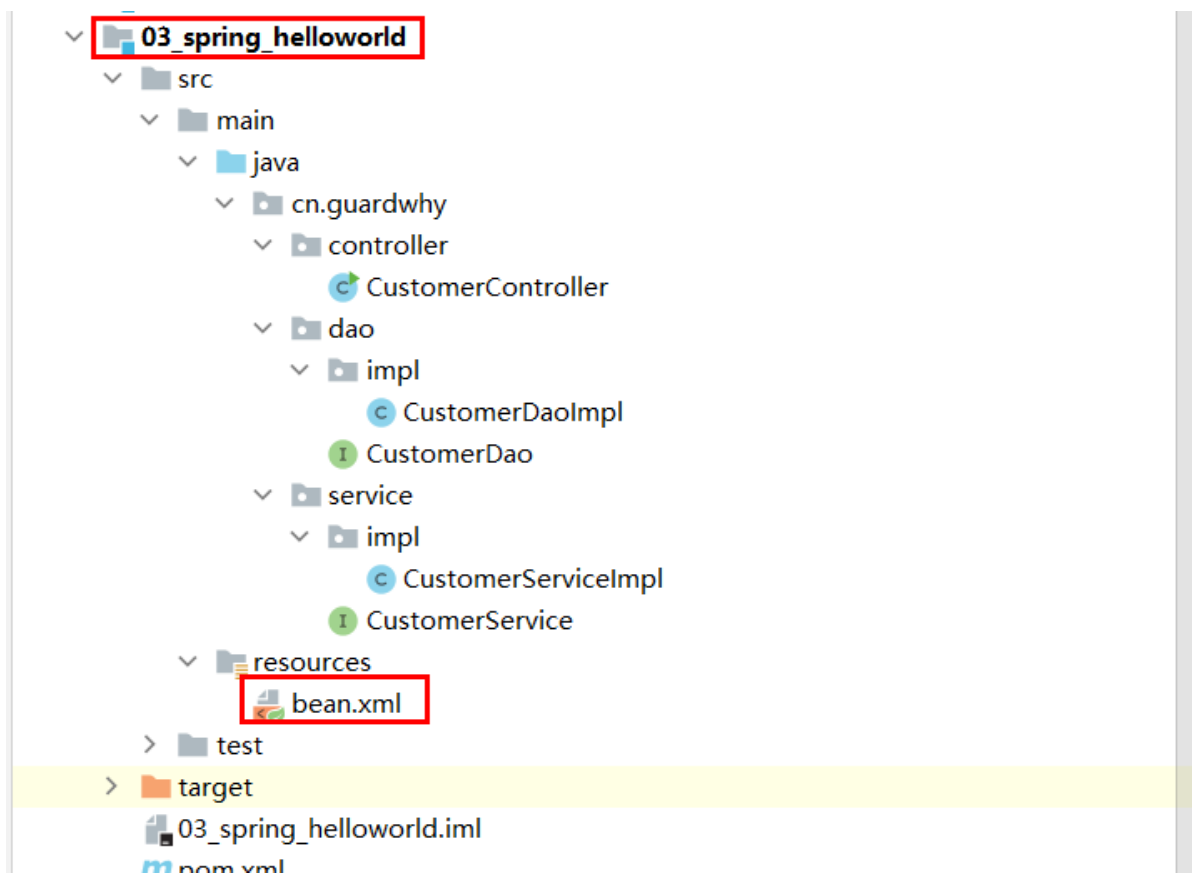
Spring容器在初始化时先读取配置文件，根据配置文件或元数据创建与组织对象存入容器中，程序使用时再从IoC容器中取出需要的对象。

采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。

总结：

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入。（Dependency Injection,DI）。

2.2 项目结构



2.3 代码示例

持久层(Dao层)

dao接口

```
package cn.guardwhy.dao;

/**
 * 客户dao接口
 */
public interface CustomerDao {

    /**
     * 保存客户操作
     */
    void saveCustomer();

}
```

dao实现类

```
package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {

    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

}
```

业务层(Service)

service接口

```
package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {

    /**
     * 保存客户操作
     */
    void saveCustomer();

}
```

service实现类

```
package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;
import cn.guardwhy.service.CustomerService;

/**
```

```

    * 客户service实现类
    */
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    private CustomerDao customerDao = new CustomerDaoImpl();

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

编写bean.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">
    <!--配置service, 说明:
    标签: bean: 配置javaBean对象
    属性:
        id: bean的唯一标识名称
        class: 类的全路径信息
        细节: 默认使用无参数构造方法, 创建对象
    -->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"></bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        /**
         使用spring框架提供的IOC容器, 获取对象:
         1.spring框架提供了一个大工厂接口: ApplicationContext
         2.该工厂接口中提供了一个getBean()方法, 用于根据bean的名称获取bean
         3.该工厂接口提供了常见的实现类: ClassPathXmlApplicationContext: 从类路径下加载
         bean.xml配置文件。创建spring的ioc容器。
        */
    }
}

```

```

    */

    // 1.加载spring配置文件, 创建spring ioc容器
    ApplicationContext context = new
    ClassPathXmlApplicationContext("bean.xml");
    // 2.从容器中获取客户service对象
    CustomerService customerService = (CustomerService)
    context.getBean("customerService");
    // 3.保存客户
    customerService.saveCustomer();
}
}

```

2.4 执行结果

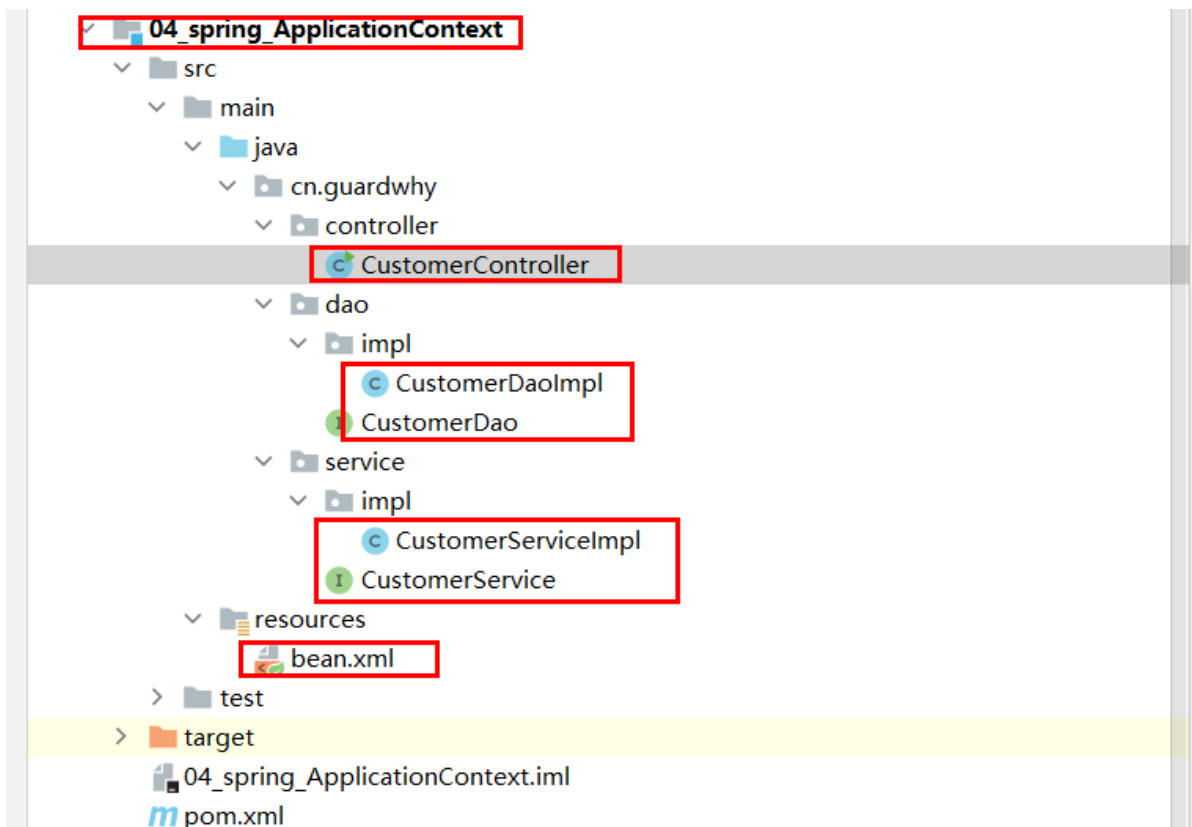
```

Run: CustomerController (1) x
"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" ...
保存客户操作
hello Spring
Process finished with exit code 0

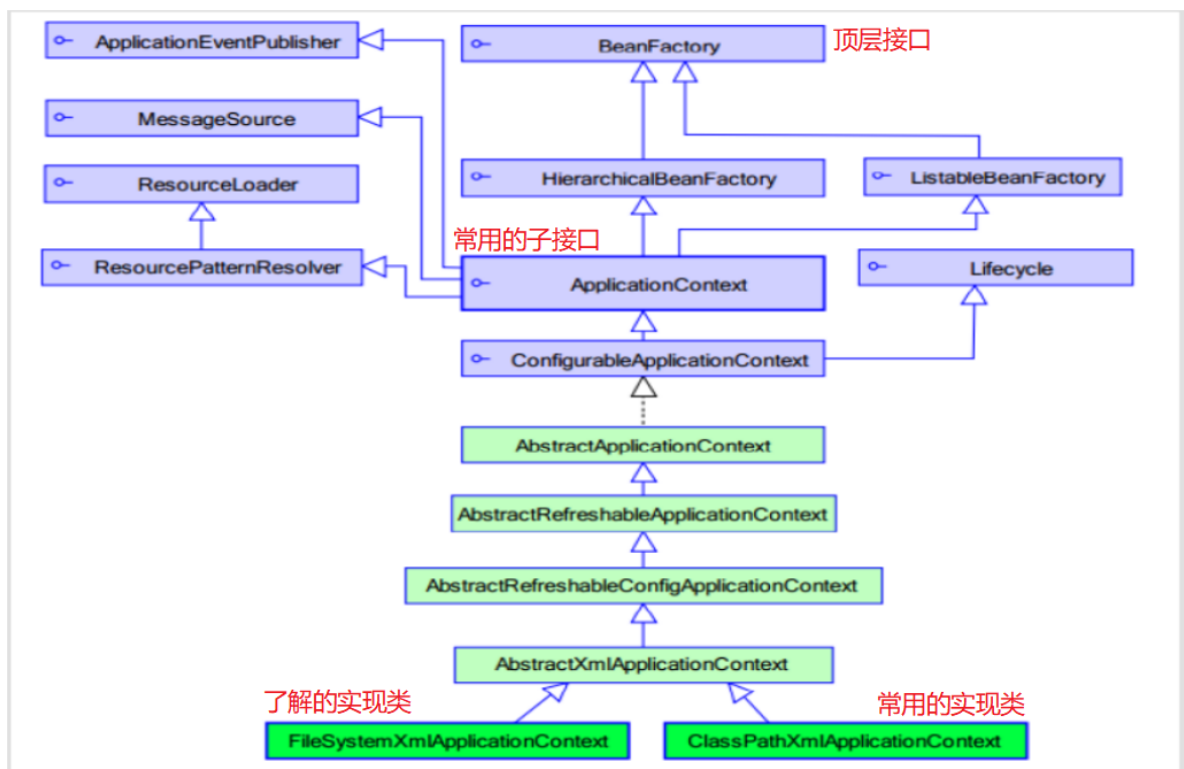
```

3-Spring 工厂类结构

3.1 项目结构



3.2 工厂类体系结构图解



3.3 接口区别

BeanFactory与ApplicationContext区别

BeanFactory是顶层接口,ApplicationContext是子接口。

它们最大的区别是创建对象的时间不一样

- BeanFactory采用的是延迟加载的思想。即什么时候使用对象，什么时候创建。
- ApplicationContext采用立即创建的思想。即一加载配置文件，立即就创建。

3.4 代码示例

持久层(Dao层)

dao接口

```
package cn.guardwhy.dao;
/**
 * 客户dao接口
 */
public interface CustomerDao {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}
```

dao实现类

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }
}

```

业务层(Service)

service接口

```

package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}

```

service实现类

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;
import cn.guardwhy.service.CustomerService;

/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    private CustomerDao customerDao = new CustomerDaoImpl();

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

编写bean.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">
    <!--配置service, 说明:
    标签:
        bean: 配置javaBean对象
    属性:
        id: bean的唯一标识名称
        class: 类的全路径信息
    细节: 默认使用无参数构造方法, 创建对象
    -->
    <bean id="customerService"
    class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"></bean>
</beans>

```

表现层(Controller)

ApplicationContext

```

package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        /**
         BeanFactory与ApplicationContext区别:
         1.BeanFactory是顶层接口
         2.ApplicationContext是子接口
         3.它们的区别是创建对象的时间点不一样:
            a、BeanFactory采用延迟加载的思想。即什么时候使用对象, 什么时候创建
            b、ApplicationContext采用立即创建的思想。即一加载spring配置文件, 立即就创建对
象
        */

        // 1.加载spring配置文件, 创建spring ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.打印容器创建完成的信息
        System.out.println("-----start");
        System.out.println("spring IOC容器创建好了");
        System.out.println("-----end");
        // 2.从容器中获取客户service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.保存客户
        customerService.saveCustomer();
    }
}

```

```
}
```

执行结果

```
"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" ...
正在创建CustomerDaoImpl对象.....
正在创建CustomerDaoImpl对象.....
-----start
spring IOC容器创建好了
-----end
保存客户操作
```

在加载spring配置文件时候，就立即创建了客户dao对象

打印容器创建好的消息

表现层(Controller)

BeanFactory

```
package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        /**
         BeanFactory与ApplicationContext区别:
         1.BeanFactory是顶层接口
         2.ApplicationContext是子接口
         3.它们的区别是创建对象的时间点不一样:
         a、BeanFactory采用延迟加载的思想。即什么时候使用对象，什么时候创建
         b、ApplicationContext采用立即创建的思想。即一加载spring配置文件，立即就创建对
         象
         */

        // 1.创建资源对象
        Resource resource = new ClassPathResource("bean.xml");
        BeanFactory context = new XmlBeanFactory(resource);

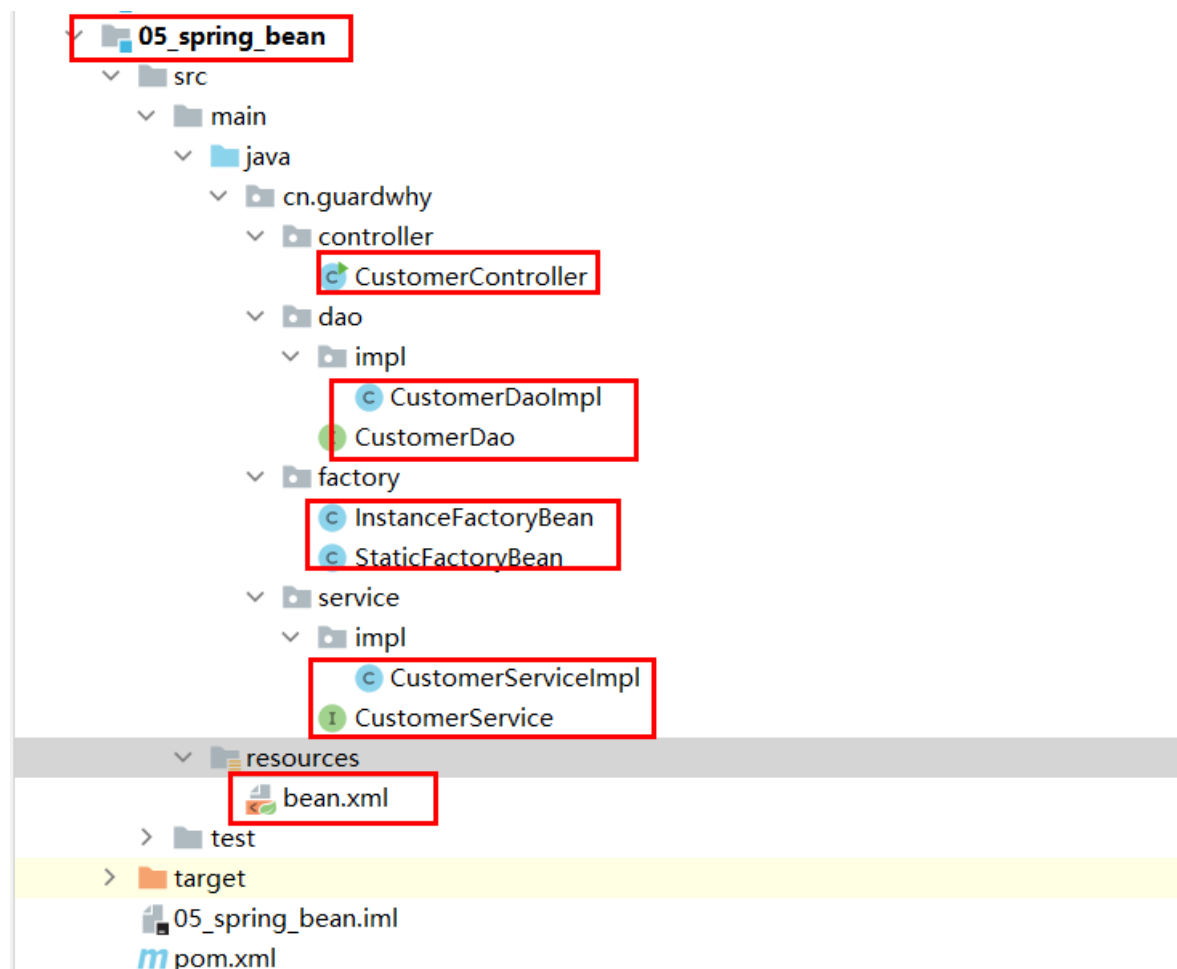
        // 2.打印容器创建的信息
        System.out.println("-----start");
        System.out.println("spring IOC容器创建好了");
        System.out.println("-----end");

        // 3.从容器中获取客户service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 4.保存客户
        customerService.saveCustomer();
    }
}
```

```
-----start
spring IOC容器创建好了
-----end
正在创建CustomerDaoImpl对象.....
保存客户操作
```

4-Spring bean标签

4.1 项目结构



4.1 bean标签

bean标签作用

#bean标签作用：配置javaBean对象。spring框架遇到bean标签，默认调用无参数构造方法实例化对象。

bean标签属性

属性	说明
id	bean的唯一标识名称。
class	类的全限定名称。
scope	设置bean的作用范围。 取值: singleton : 单例, 默认值。 prototype: 多例
init-method	指定类中初始化方法的名称,在构造方法执行完毕后立即执行。
destroy-method	指定类中销毁方法名称,在销毁spring容器前执行。

bean作用范围

	作用范围	生命周期
单例对象: scope="singleton"	一个应用中只有一个对象实例	出生: 加载配置文件, 容器创建, 对象出生 活着: 只要容器存在, 对象就一直活着 死亡: 容器销毁, 对象死亡
多例对象: scope="prototype"	在一次使用过程中	出生: 第一次获取对象, 对象出生 活着: 在一次使用过程中, 对象活着 死亡: 当对象不再使用, 也没有被其它对象引用, 交由垃圾回收器回收

4.3 代码示例

持久层(Dao层)

dao接口

```
package cn.guardwhy.dao;
/**
 * 客户dao接口
 */
public interface CustomerDao {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}
```

dao实现类

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }
}

```

业务层(Service)

service接口

```

package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}

```

service实现类

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;
import cn.guardwhy.service.CustomerService;

/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    private CustomerDao customerDao = new CustomerDaoImpl();

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

编写bean.xml

4.3.1 singleton标签

```
<?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">
    <!--配置dao,说明:
        标签:
            bean: 配置javaBean对象
        属性:
            id: bean的唯一标识名称
            class: 类的全路径信息
            scope: 设置bean的作用范围
        取值:
            singleton: 单例(默认)
            prototype: 多例(原型)
        细节:
            默认使用无参数构造方法,创建对象
    -->
    <!--配置dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
        scope="singleton"></bean>
</beans>
```

表现层(Controller)

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {

        // 1.加载spring配置文件,创建ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");

        // 2.获取两次dao对象,检查是否是同一个对象
        CustomerDao customerDao1 = (CustomerDao) context.getBean("customerDao");
        CustomerDao customerDao2 = (CustomerDao) context.getBean("customerDao");
        System.out.println(customerDao1 == customerDao2);
        // 3.条件判断
        System.out.println(customerDao1.hashCode());
        System.out.println(customerDao2.hashCode());

    }
}
```

执行结果

```
正在创建CustomerDaoImpl对象
正在创建CustomerDaoImpl对象
true
471579726
471579726
Process finished with exit code 0
```

配置scope="singleton"
指定为单列

编写bean.xml

4.3.2 prototype标签

```
<?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">
  <!--配置dao,说明:
    标签:
      bean: 配置javaBean对象
    属性:
      id: bean的唯一标识名称
      class: 类的全路径信息
      scope: 设置bean的作用范围
      取值:
        singleton: 单例 (默认)
        prototype: 多例
    细节:
      默认使用无参数构造方法, 创建对象
  -->
  <!--配置dao-->
  <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
scope="prototype"></bean>
</beans>
```

执行结果

```
正在创建CustomerDaoImpl对象
正在创建CustomerDaoImpl对象
false
1261153343
1309176095
Process finished with exit code 0
```

配置属性scope="prototype"
指定为:多例

4.3.3 初始化和销毁

dao实现类

```
package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {
    // 无参构造器:BeanFactory与ApplicationContext创建对象的区别
    public CustomerDaoImpl() {
        System.out.println("正在创建CustomerDaoImpl对象");
    }

    /**
     * 初始化方法, init-method属性
     */
    public void init(){
        System.out.println("正在执行初始化操作...");
    }

    /**
     * 销毁方法:destory-method属性。
     */
    public void destroy(){
        System.out.println("正在执行销毁操作");
    }

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }
}
```

编写bean.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">
    <!--配置dao,说明:
        标签:
            bean: 配置javaBean对象
        属性:
            id: bean的唯一标识名称
            class: 类的全路径信息
            scope: 设置bean的作用范围
        取值:
            singleton: 单例 (默认)
```


prototype: 多例

init-method: 初始化操作, 在调用构造方法执行后执行

destroy-method: 销毁操作, 在销毁spring IOC容器前执行

注意: 默认使用无参数构造方法, 创建对象

-->

<!--配置dao-->

```
<bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
scope="singleton" init-method="init" destroy-method="destroy"></bean>
</beans>
```

表现层(Controller)

CustomerController

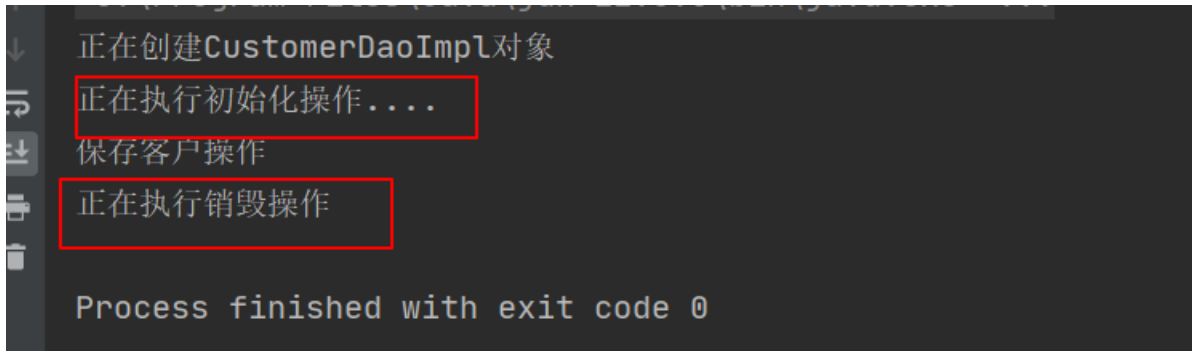
```
package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        /**
         bean标签作用: 配置javaBean对象
         属性: id: 唯一标识名称, class: 指定类的全路径
         scope: 设置bean的作用范围
         属性取值:
         singleton: 单例。默认值
         prototype: 多例
         init-method: 初始化
         destroy-method: 执行销毁操作
         注意: 默认使用无参数构造方法实例化
         */

        // 1.加载spring配置文件, 创建ioc容器,大工厂接口中没有销毁的方法, 在实现类中才存在
        ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.获取dao对象
        CustomerDao customerDao = (CustomerDao) context.getBean("customerDao");
        // 3.保存客户
        customerDao.saveCustomer();
        // 4.销毁spring容器
        context.close();
    }
}
```

执行结果



4.4 import导入

这个import，一般用于团队开发使用，它可以将多个配置文件，导入合并成一个。

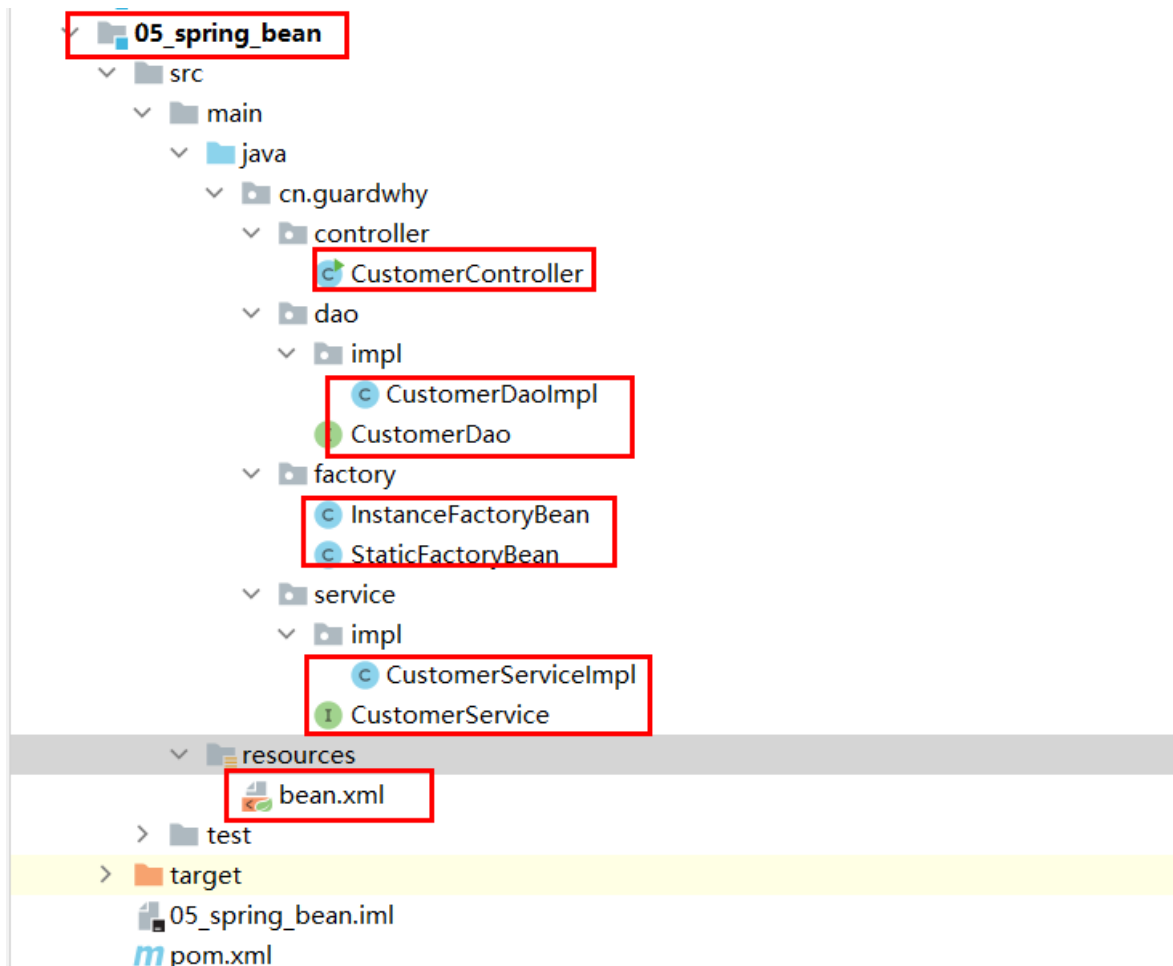
假设，现在项目中有多人开发。这三个人复制不同的类开发，不同的类需要注册在不同的bean中，可以用import将所有人的beans.xml合并成一个总的。

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">
    <!--导入分支-->
    <import resource="bean1.xml"/>
    <import resource="bean2.xml"/>
    <import resource="bean3.xml"/>
</beans>
```

5-Spring实例化bean

5.1 项目结构



5.2 无参构造方法

配置bean.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">
  <!--配置客户service
    标签:
      bean: 配置javaBean对象
    属性:
      id: 唯一标识名称
      class: 类的全路径
    注意: 默认调用类的无参数构造方法, 实例化对象
  -->
  <bean id="customerService"
        class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

  <!--配置客户dao:
    标签:
      bean: 配置javaBean对象
    属性:
      id: 唯一标识名称
      class: 类的全路径
      scope: 设置bean的作用范围
      属性取值: singleton:单例(默认值) prototype: 多例
```

init-method: 执行初始化, 在构造方法执行后立即执行
destroy-method: 执行销毁, 在销毁spring容器的前执行

注意: 默认调用类的无参数构造方法, 实例化对象

```
-->
<bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
      scope="singleton" init-method="init" destroy-method="destroy"></bean>
</beans>
```

5.3 静态工厂方法

静态工厂类

```
package cn.guardwhy.factory;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;

/**
 * 静态工厂方法, 实例化对象
 */
public class StaticFactoryBean {
    /**
     * 静态工厂方法
     */
    public static CustomerDao createCustomerDao(){
        // 返回结果
        CustomerDao customerDao = null;
        System.out.println("静态工厂方法实例化对象-----start");
        customerDao = new CustomerDaoImpl();
        System.out.println("静态工厂方法实例化对象-----end");
        return customerDao;
    }
}
```

配置bean.xml

```
<!--静态工厂方法实例化对象, 说明:
    属性:
        id: 唯一标识名称
        class: 类全路径
        factory-method: 指定工厂方法
-->
<bean id="staticDao" class="cn.guardwhy.factory.StaticFactoryBean" factory-
method="createCustomerDao"></bean>
```

表现层

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.静态工厂实例化对象
        CustomerDao staticDao = (CustomerDao) context.getBean("staticDao");
        // 3.保存用户
        staticDao.saveCustomer();
    }
}

```

执行结果

```

正在创建CustomerDaoImpl对象
正在创建CustomerDaoImpl对象
正在执行初始化操作....
静态工厂方法实例化对象-----start
正在创建CustomerDaoImpl对象
静态工厂方法实例化对象-----end
保存客户操作

```

5.4 实例工厂方法

编写工厂类

```

package cn.guardwhy.factory;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;

/**
 * 实例工厂方法
 */
public class InstanceFactoryBean {
    // 工厂方法
    public CustomerDao createCustomerDao(){
        // 1.返回结果
        CustomerDao customerDao = null;
        System.out.println("实例工厂方法实例化对象-----start");
        customerDao = new CustomerDaoImpl();
        System.out.println("实例工厂方法实例化对象-----end");
        return customerDao;
    }
}

```

配置bean.xml

```

<!--实例工厂方法实例化对象
    第一步：配置静态工厂对象
    第二步：factory-bean: 指定工厂对象
            factory-method: 指定工厂方法
-->
<bean id="instanceFactory" class="cn.guardwhy.factory.InstanceFactoryBean">
</bean>
<bean id="instanceDao" factory-bean="instanceFactory" factory-
method="createCustomerDao"></bean>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.创建spring IOC容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
        // 2.实例工厂方法实例化对象
        CustomerDao instanceDao = (CustomerDao) context.getBean("instanceDao");
        // 3.保存用户
        instanceDao.saveCustomer();
    }
}

```

操作结果

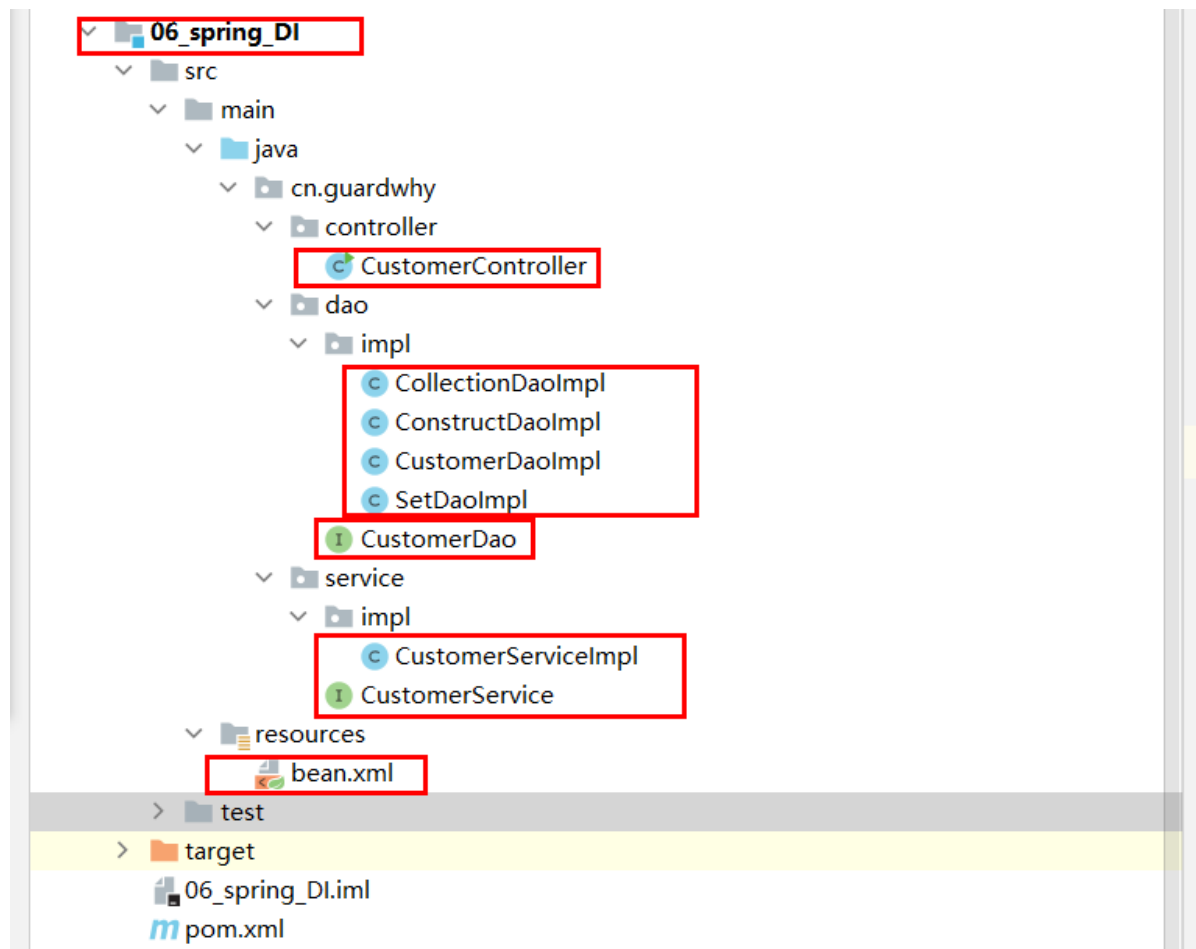
```

实例工厂方法实例化对象-----start
正在创建CustomerDaoImpl对象
实例工厂方法实例化对象-----end
保存客户操作

```

6- 依赖注入(xml)

6.1 项目结构



6.2 构造方法

就是通过构造方法，给类的成员变量赋值

持久层(Dao)

CustomerDao

```
package cn.guardwhy.dao;

/**
 * 客户dao接口
 */
public interface CustomerDao {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}
```

ConstructDaoImpl

```
package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

import java.util.Date;
/**
```

```

    * 构造方法注入数据
    */
public class ConstructDaoImpl implements CustomerDao {
    // 1.类的成员变量
    private int id;
    private String name;
    private Integer age;
    private Date birthday;

    // 2.构造方法
    public ConstructDaoImpl(int id, String name, Integer age, Date birthday) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.birthday = birthday;
    }

    @Override
    /**
     * 保存客户操作
     */
    public void saveCustomer() {
        System.out.println("id="+id+", name=" + name + ",age=" +age+
            ",birthday=" + birthday);
    }
}

```

业务层(Service)

CustomerService

```

package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 保存客户操作
     */
    void saveCustomer();
}

```

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;
import cn.guardwhy.service.CustomerService;

/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    private CustomerDao customerDao;

    /**

```



```

    * 保存客户操作
    */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

配置bean.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">

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置客户dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
        scope="singleton" init-method="init" destroy-method="destroy"
    ></bean>

    <!--构造方法注入数据
    标签:
        constructor-arg: 指定通过构造方法给成员变量赋值
    属性:
        index: 成员变量在构造方法参数列表中的索引
        name: 成员变量的名称（与index使用一个即可，实际项目中用的更多）
        type: 成员变量的类型（默认即可）
        value: 给java简单类型成员变量赋值（八种基本类型+字符串String）
        ref: 给其他bean类型成员变量赋值
    -->
    <bean id="constructDao" class="cn.guardwhy.dao.impl.ConstructDaoImpl">
        <constructor-arg index="0" name="id" type="int" value="1"></constructor-
arg>
        <constructor-arg name="name" value="kobe"></constructor-arg>
        <constructor-arg name="age" value="18"></constructor-arg>
        <constructor-arg name="birthday" ref="now"></constructor-arg>
    </bean>
    <!--配置java.util.Date-->
    <bean id="now" class="java.util.Date"></bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

```

```

/**
 * 客户controller
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.从容器中获取service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.构造方法注入数据,保存用户
        CustomerDao constructDao = (CustomerDao)
        context.getBean("constructDao");
        constructDao.saveCustomer();
    }
}

```

执行结果

```

正在创建客户CustomerDaoImpl对象.
正在执行初始化操作.....
id=1, name=kobe,age=18,birthday=Sat Oct 24 10:15:19 CST 2020

Process finished with exit code 0

```

6.3 Set方法

Set方法注入，就是通过set方法，给类的成员变量赋值。

持久层(Dao)

SetDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

import java.util.Date;

/**
 * set方法注入数据
 */
public class SetDaoImpl implements CustomerDao {
    // 1.类的成员变量
    private int id;
    private String name;
    private Integer age;
    private Date birthday;

    // 2.set方法
    public void setId(int id) {
        this.id = id;
    }
}

```

```

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

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

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

/**
 * 保存客户操作
 */
@Override
public void saveCustomer() {
    System.out.println("id="+id+", name=" + name + ",age=" +age+
        ",birthday=" + birthday);
}
}

```

配置bean.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">

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置客户dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
        scope="singleton" init-method="init" destroy-method="destroy">
    </bean>

    <!--set方法注入数据
    标签:
        property: 指定通过set方法, 给类的成员变量赋值
    属性:
        name: 指定成员变量名称
        value: 给java简单类型成员变量赋值 (八种基本类型+字符串String)
        ref: 给其他bean类型成员变量赋值
    -->
    <bean id="setDao" class="cn.guardwhy.dao.impl.SetDaoImpl">
        <property name="id" value="2" ></property>
        <property name="name" value="Curry"></property>
        <property name="age" value="18"></property>
        <property name="birthday" ref="now"></property>
    </bean>

    <!--配置java.util.Date-->
    <bean id="now" class="java.util.Date"></bean>
</beans>

```

表现层(Controller)

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.从容器中获取service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");

        // 3.set方法注入数据
        CustomerDao setDao = (CustomerDao) context.getBean("setDao");
        setDao.saveCustomer();
    }
}
```

执行结果

正在创建客户CustomerDaoImpl对象.

正在执行初始化操作.....

id=2, name=Curry,age=18,birthday=Sat Oct 24 12:24:00 CST 2020

Process finished with exit code 0

6.4 p名称空间

p名称空间注入：就是set方法注入。其本质在于简化配置

配置bean.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">

    <!--配置客户service-->
```

```

<bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

<!--配置客户dao-->
<bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
scope="singleton" init-method="init" destroy-method="destroy"
></bean>
<!--p名称空间注入
    第一步：导入p名称空间
        xmlns:p="http://www.springframework.org/schema/p"
    第二步：
        p:属性名称 ==>给java简单类型成员变量赋值
        p:属性名称-ref ==>给其他bean类型成员变量赋值
-->
<bean id="pDao" class="cn.guardwhy.dao.impl.SetDaoImpl"
p:id="3" p:name="James" p:age="35" p:birthday-ref="now"
>
</bean>

<!--配置java.util.Date-->
<bean id="now" class="java.util.Date"></bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.从容器中获取service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");

        // 3.p名称注入数据
        CustomerDao pDao = (CustomerDao) context.getBean("pDao");
        pDao.saveCustomer();
    }
}

```

执行结果

正在创建客户CustomerDaoImpl对象。

正在执行初始化操作.....

id=3, name=James,age=35,birthday=Sat Oct 24 13:33:29 CST 2020

Process finished with exit code 0

6.5 C名称空间

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

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置客户dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
        scope="singleton" init-method="init" destroy-method="destroy"
    ></bean>

    <!--c名称空间注入
        第一步：导入c名称空间
            xmlns:c="http://www.springframework.org/schema/c"
        第二步：
            c:属性名称 ==>给java简单类型成员变量赋值
            c:属性名称-ref ==>给其他bean类型成员变量赋值
    -->
    <bean id="cDao" class="cn.guardwhy.dao.impl.ConstructDaoImpl"
        c:id="4" c:name="Rondo" c:age="21" c:birthday-ref="now"
    >
    </bean>

    <!--配置java.util.Date-->
    <bean id="now" class="java.util.Date"></bean>
</beans>
```

表现层(Controller)

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```

/**
 * 客户controller
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.从容器中获取service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.名称空间注入数据
        CustomerDao cDao = (CustomerDao) context.getBean("cDao");
        cDao.saveCustomer();
    }
}

```

执行结果

正在创建客户CustomerDaoImpl对象。

正在执行初始化操作.....

id=4, name=Rondo,age=21,birthday=Sat Oct 24 17:08:21 CST 2020

Process finished with exit code 0

6.6 集合属性注入

持久层(Dao)

CollectionDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;

import java.util.*;

/**
 * 集合属性成员变量赋值
 */
public class CollectionDaoImpl implements CustomerDao {
    // 集合类型成员变量
    private String[] array;
    private List<String> list;
    private Set<String> set;

    private Map<String, String> map;
    private Properties prop;

    // set方法注入
    public void setArray(String[] array) {
        this.array = array;
    }
}

```

```

    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }

    public void setMap(Map<String, String> map) {
        this.map = map;
    }

    public void setProp(Properties prop) {
        this.prop = prop;
    }

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        System.out.println(array != null ? Arrays.asList(array): "");
        System.out.println(list);
        System.out.println(set);
        System.out.println(map);
        System.out.println(prop);
    }
}

```

配置bean.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">

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置客户dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl"
        scope="singleton" init-method="init" destroy-method="destroy"
    ></bean>
    <!--集合属性注入
        1.List结构:
            array/list/set
        2.Map结构:
            map/prop
        3.数据结构一致, 标签可以互换
    -->
    <bean id="CollectionDao" class="cn.guardwhy.dao.impl.CollectionDaoImpl">
        <!--array-->
        <property name="array">

```



```

        <array>
            <value>array1</value>
            <value>array2</value>
        </array>
    </property>
    <!--list-->
    <property name="list">
        <list>
            <value>list1</value>
            <value>list2</value>
        </list>
    </property>
    <!--set-->
    <property name="set">
        <set>
            <value>set1</value>
            <value>set2</value>
        </set>
    </property>

    <!--map-->
    <property name="map">
        <map>
            <entry key="map-k1" value="map-v1"></entry>
            <entry key="map-k1" value="map-v1"></entry>
        </map>
    </property>

    <!--prop-->
    <property name="prop">
        <props>
            <prop key="prop-k1"> prop-v1</prop>
            <prop key="prop-k1"> prop-v2</prop>
        </props>
    </property>
</bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.创建spring IOC容器
    }
}

```

```

        ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
        // 2.从容器中获取service对象
        CustomerService customerService = (CustomerService)
context.getBean("customerService");
        // 3.集合类型成员变量注入数据
        CustomerDao collectionDao = (CustomerDao)
context.getBean("CollectionDao");
        collectionDao.saveCustomer();
    }
}

```

执行结果

```

Run: CustomerController (4) x
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
正在创建客户CustomerDaoImpl对象.
正在执行初始化操作.....
[array1, array2]
[list1, list2]
[set1, set2]
{map-k1=map-v1}
{prop-k1=prop-v2}

Process finished with exit code 0

```

7- Bean 自动装配

7.1 项目目录

```

25_spring_auto_bean
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── cn.guardwhy.domain
│   │   │   │   ├── Student
│   │   │   │   ├── Teacher
│   │   │   │   └── User
│   │   │   └── resources
│   │   │       └── beans.xml
│   │   └── test
│   │       ├── java
│   │       │   ├── cn.guardwhy.Test
│   │       │   └── MyTest
│   └── target
│       ├── 25_spring_auto_bean.iml
│       ├── pom.xml
│       └── pom.xml

```

7.2 自动装配概念

- 自动装配是使用spring满足bean依赖的一种方法。
- spring会在应用上下文中为某个bean寻找其依赖的bean。

Spring中bean有三种装配机制，分别是：

1. 在xml中显式配置；
2. 在java中显式配置；
3. 隐式的bean发现机制和自动装配。

Spring的自动装配需要从两个角度来实现，或者说是两个操作：

1. 组件扫描(component scanning)：spring会自动发现应用上下文中所创建的bean。
2. 自动装配(autowiring)：spring自动满足bean之间的依赖，也就是我们说的IoC/DI。

注意:组件扫描和自动装配组合发挥巨大威力，使的显示的配置降低到最少。

7.3 代码示例

新建两个实体类，Student Teacher 都有一个function的方法。

```
package cn.guardwhy.domain;

public class Student {
    public void function(){
        System.out.println("学生学习!!!");
    }
}
```

```
package cn.guardwhy.domain;

public class Teacher {
    public void function(){
        System.out.println("老师教导学生学习!!");
    }
}
```

新建一个用户类 User

```
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Student student;
    private Teacher teacher;
    private String str;
}
```

编写Spring配置文件

```
<?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="student" class="cn.guardwhy.domain.Student"/>
    <bean id="teacher" class="cn.guardwhy.domain.Teacher"/>

    <bean id="user" class="cn.guardwhy.domain.User">
        <property name="student" ref="student"/>
        <property name="teacher" ref="teacher"/>
        <property name="str" value="guardwhy"/>
    </bean>
</beans>
```

测试代码

```
package cn.guardwhy.Test;

import cn.guardwhy.domain.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    @Test
    public void testMethodAutowire(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("beans.xml");
        User user = (User) context.getBean("user");
        user.getTeacher().function();
        user.getStudent().function();
    }
}
```

7.4 byName

autowire byName (按名称自动装配)

由于在手动配置xml过程中，常常发生字母缺漏和大小写等错误，而无法对其进行检查，使得开发效率降低。

采用自动装配将避免这些错误，并且使配置简单化。

测试代码

1. 修改bean配置，增加一个属性 autowire="byName"

```
<!--byName:会自动在容器上下文中查找，和自己对象set方法后面的值对应的beanid-->
<bean id="user" class="cn.guardwhy.domain.User" autowire="byName">
    <property name="student" ref="student"/>
</bean>
```

总结

当一个bean节点带有 autowire byName的属性时。

1. 将查找其类中所有的set方法名，例如setStudent，获得将set去掉并且首字母小写的字符串，即student。
2. 去spring容器中寻找是否有此字符串名称id的对象。
3. 如果有，就取出注入；如果没有，就报空指针异常。

7.5 byType

autowire byType (按类型自动装配)

测试代码

使用autowire byType首先需要保证：同一类型的对象，在spring容器中唯一。如果不唯一，会报不唯一的异常。

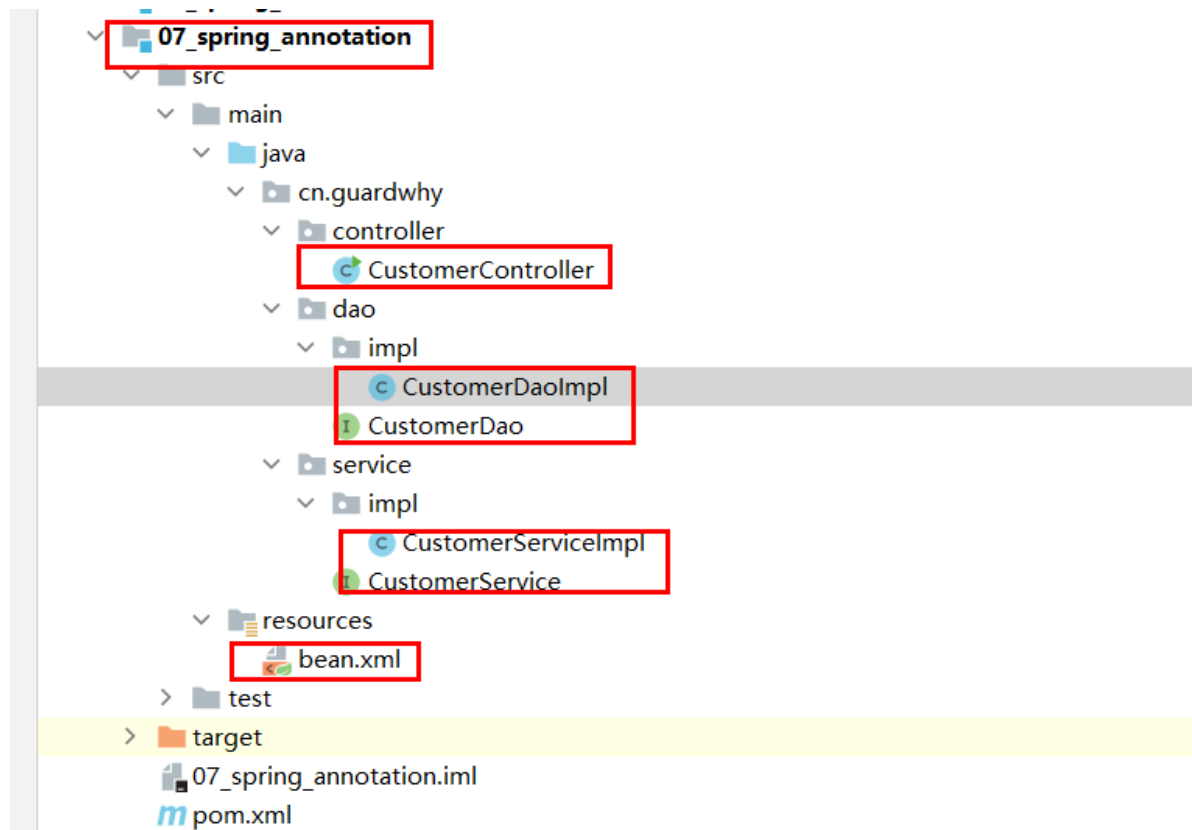
```
<!--byType:会自动在容器上下文中查找，和自己对象属性类型相同的bean!! -->
<bean id="user" class="cn.guardwhy.domain.User" autowire="byType">
    <property name="student" ref="student"/>
</bean>
```

7.6 总结

- byname的时候,需要保证所有的bean的id唯一，并且这个bean需要和自动注入的属性的set方法的值一致。
- bytype的时候,需要保证所有的bean的class唯一，并且这个bean需要和自动注入的属性的类型一致。

7- Spring 入门案例(注解)

7.1 项目目录



7.2 代码示例

持久层(Dao)

CustomerDao

```
package cn.guardwhy.dao;

/**
 * 客户dao接口
 */
public interface CustomerDao {

    /**
     * 保存客户操作
     */
    void saveCustomer();

}
```

CustomerDaoImpl

```
package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;
import org.springframework.stereotype.Component;

/**
 * 客户dao实现类
 */
@Component("customerDao")
public class CustomerDaoImpl implements CustomerDao {

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

}
```

业务层(Service)

CustomerService

```
package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {

    /**
     * 保存客户操作
     */
    void saveCustomer();

}
```

CustomerServiceImpl

```
package cn.guardwhy.service.impl;
```

```

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.service.CustomerService;

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

/**
 * 客户service实现类
 */
@Component("customerService")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

配置bean.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">

    <!--配置包扫描注解配置dao/service
        第一步：导入context名称空间和约束
        第二步：
            通过<context:component-scan>标签配置包扫描。spring框架在创建IOC容器的时候，
            会扫描指定的包，和它的子包
    -->
    <context:component-scan base-package="cn.guardwhy"></context:component-scan>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller

```

```

*/
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.实例工厂方法实例化对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.保存用户
        customerService.saveCustomer();
    }
}

```

执行结果

```

"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" ...
保存客户操作

Process finished with exit code 0

```

7.3 常用注解

7.3.1 创建对象注解

@Component

#作用:

配置javaBean对象。相当于xml配置方式中的bean标签。

#属性:

value: 给bean一个唯一标识名称。

#细节:

- 1.value属性可以省略。
- 2.默认使用类的名称首字母小写，作为bean的名称。

代码示例


```

/**
 * 客户service实现类
 */
@Component("customerService")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        customerDao.saveCustomer();
    }
}

```

@Controller、@Service、@Repository

@Component演化的三个注解

- @Controller：一般用于表现层
- @Service：一般用于业务层
- @Repository：一般用于持久层

代码示例

```

/**
 * 客户service实现类
 */
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;
}

```

7.3.2 bean作用范围注解

@Scope

#作用：

设置bean的作用范围。相当于xml配置方式中bean标签的scope属性

#属性：

value：指定作用范围取值

#属性取值：

singleton：单例。默认值
prototype：多例

代码示例

```
/**
 * 客户service实现类
 */
@Service("customerService")
@Scope(value = "singleton")
public class CustomerServiceImpl implements CustomerService {
```

7.3.3 注入数据注解

@Autowired

#作用:

默认按照bean的类型注入数据

#属性:

required: 指定目标bean是否必须存在于spring的IOC容器 (true必须存在; false: 可以不存在; 默认true)

#细节:

- 在spring容器中, 如果同一个类型存在多个bean实例对象。
- 则先按照bean的类型进行注入, 再按照bean的名称进行匹配。
- 匹配上注入成功; 匹配不上注入失败。

代码示例

```
@{...}
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;
```

@Resource

#作用:

默认按照bean的名称注入数据

#属性:

name: 指定bean的名称注入数据
type: 指定bean的类型注入数据

#细节:

默认按照bean的名称匹配注入数据。如果注入失败, 再按照bean的类型注入。

代码示例

```

/**
/**
 * 客户service实现类:
 */
@Service("customerService")
@Scope(value = "singleton")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    // @Resource(name="customerDao")
    @Resource(type = CustomerDaoImpl.class)
    private CustomerDao customerDao;

```

@Value

给java简单类型成员变量注入数据

代码示例

```

public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    // @Resource(name="customerDao")
    @Resource(type = CustomerDaoImpl.class)
    private CustomerDao customerDao;

    // 2.简单类型成员变量
    @Value("1")
    private int id;
    @Value("小明")
    private String name;

```

7.3.4 bean生命周期注解

@PostConstruct @PreDestroy

##@PostConstruct:

初始化操作，相当于xml配置方式中bean标签的init-method属性。

##@PreDestroy:

销毁操作，相当于xml配置方式中bean标签的destroy-method属性。

代码示例

```

// 初始化操作
@PostConstruct
public void init(){
    System.out.println("正在执行初始化操作");
}

// 销毁操作
@PreDestroy
public void destroy(){
    System.out.println("正在执行销毁操作....");
}

```

7.4 代码示例

持久层(Dao)

CustomerDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;
import org.springframework.stereotype.Repository;

/**
 * 客户dao实现类
 */
@Repository("customerDao")
public class CustomerDaoImpl implements CustomerDao {
    /**
     * 初始化方法
     */
    public void init(){
        System.out.println("正在执行初始化操作.....");
    }

    /**
     * 销毁方法
     */
    public void destroy(){
        System.out.println("正在执行销毁操作.....");
    }

    /**
     * 无参数构造方法
     */
    public CustomerDaoImpl(){
        System.out.println("正在创建客户CustomerDaoImpl对象.");
    }

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {

```

```

        System.out.println("保存客户操作");
    }
}

```

业务层(Service)

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.dao.impl.CustomerDaoImpl;
import cn.guardwhy.service.CustomerService;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

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

/**
 * 客户service实现类:
 */
@Service("customerService")
@Scope(value = "singleton")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    // @Resource(name="customerDao")
    @Resource(type = CustomerDaoImpl.class)
    private CustomerDao customerDao;

    // 2.简单类型成员变量
    @Value("1")
    private int id;
    @Value("小明")
    private String name;

    // 初始化操作
    @PostConstruct
    public void init(){
        System.out.println("正在执行初始化操作");
    }

    // 销毁操作
    @PreDestroy
    public void destroy(){
        System.out.println("正在执行销毁操作...");
    }

    /**
     * 保存客户操作
     */
    @Override
    public void saveCustomer() {
        System.out.println("id="+id+",name="+name);
    }
}

```

```
        customerDao.saveCustomer();
    }
}
```

表现层(Controller)

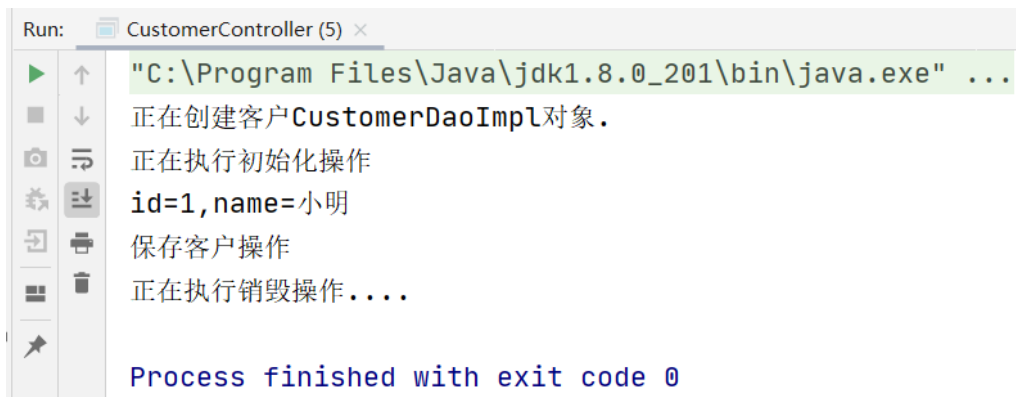
CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户controller
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring ioc容器
        ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("bean.xml");
        // 2.实例工厂方法实例化对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.保存用户
        customerService.saveCustomer();
        // 4.销毁容器
        context.close();
    }
}
```

执行结果

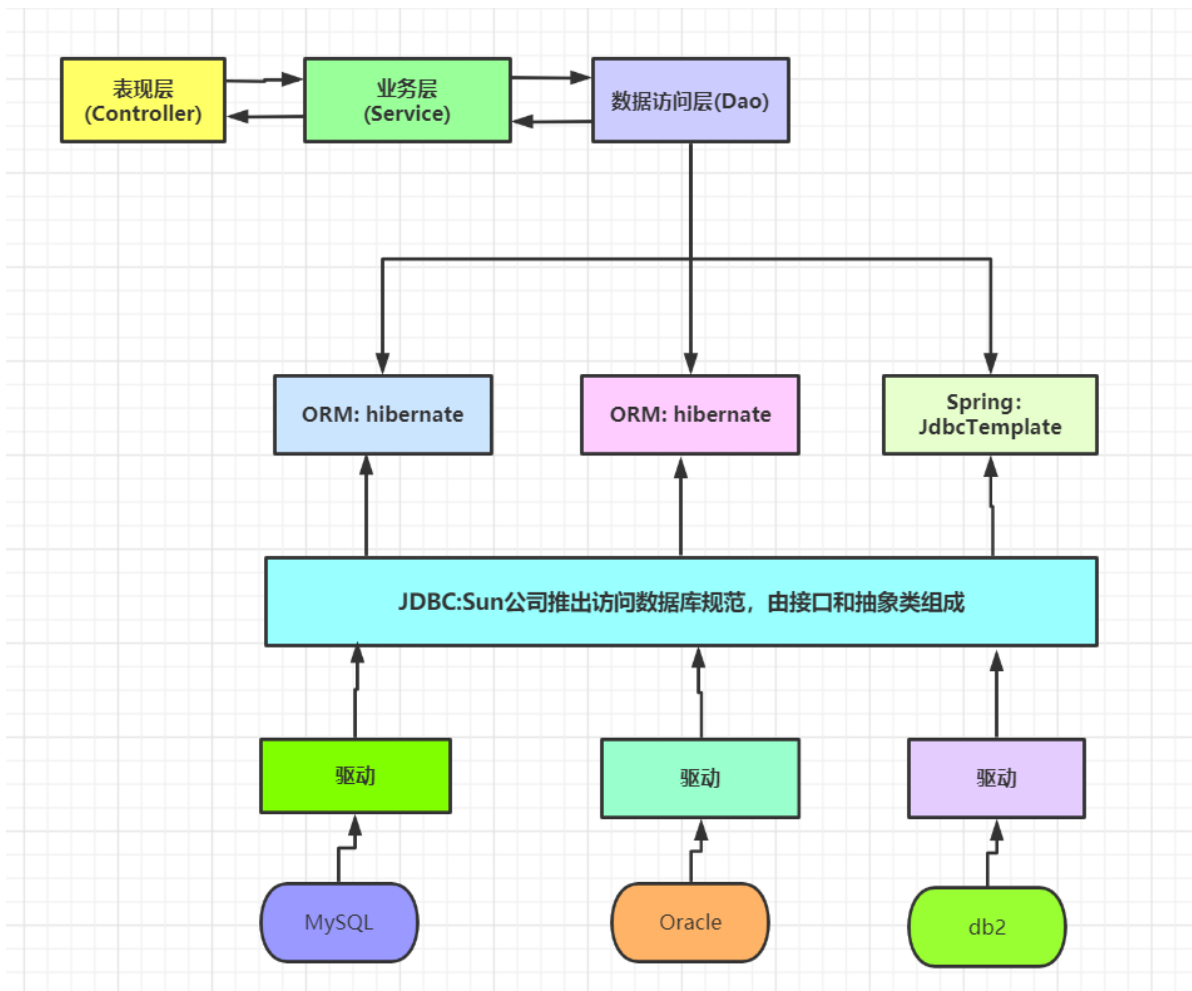


```
Run: CustomerController (5) x
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
正在创建客户CustomerDaoImpl对象.
正在执行初始化操作
id=1,name=小明
保存客户操作
正在执行销毁操作....

Process finished with exit code 0
```

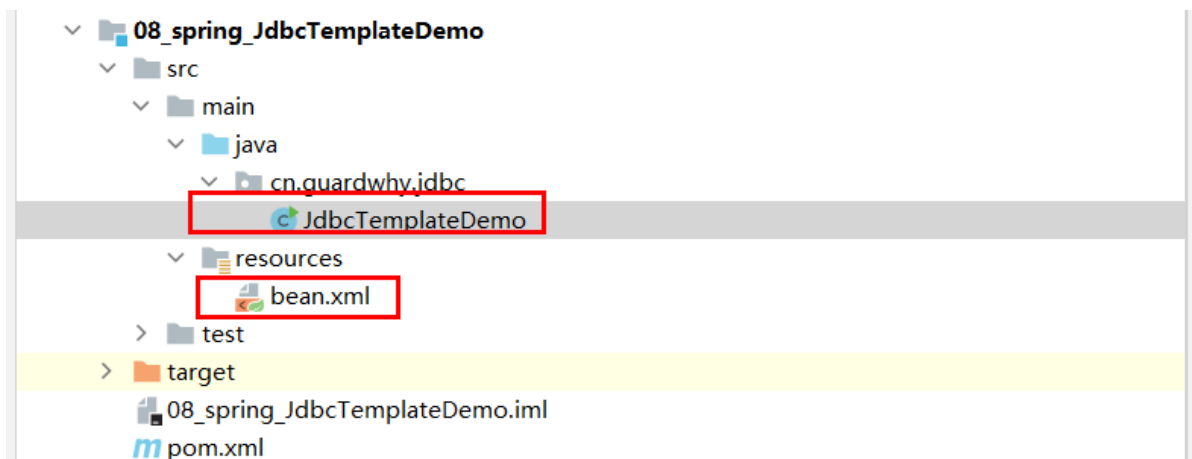
8- JdbcTemplate入门

8.1 基本概述



JdbcTemplate是spring提供的一个模板类，它是对jdbc的封装。用于支持持久层的操作，它的特点是：简单、方便。

8.2 项目目录



pom.xml

```

<!-- 依赖版本 -->
<properties>
    <!-- spring 版本 -->
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<!-- spring jdbc 依赖 -->

```

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring.version}</version>
  <!--lombok插件-->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.16</version>
  </dependency>
</dependency>

```

8.3 SQL数据表结构

```

-- 创建账户表
create table account(
  id int primary key auto_increment,
  name varchar(40),
  money float
)ENGINE=InnoDB character set utf8 collate utf8_general_ci;

-- 初始化新增三个账户
insert into account(name,money) values('kebe',1000);
insert into account(name,money) values('curry',1500);
insert into account(name,money) values('james',2000);

select * from account;

```

8.4 代码示例

传统方式实现

```

package cn.guardwhy.jdbc;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

/**
 * spring JdbcTemplate入门案例
 */
public class JdbcTemplateDemo {
  public static void main(String[] args) {
    // 1.创建JdbcTemplate对象
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    // 2.设置数据源对象
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/spring");
    dataSource.setUsername("root");
    dataSource.setPassword("root");

    jdbcTemplate.setDataSource(dataSource);
    // 3.添加账户信息
    jdbcTemplate.update("insert into account(name,money)
values('Duncan',2500)");
  }
}

```



```
}
```

IOC方式实现

bean.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">

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--配置数据源对象-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--注入连接数据库-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
</beans>
```

IOC管理JdbcTemplate

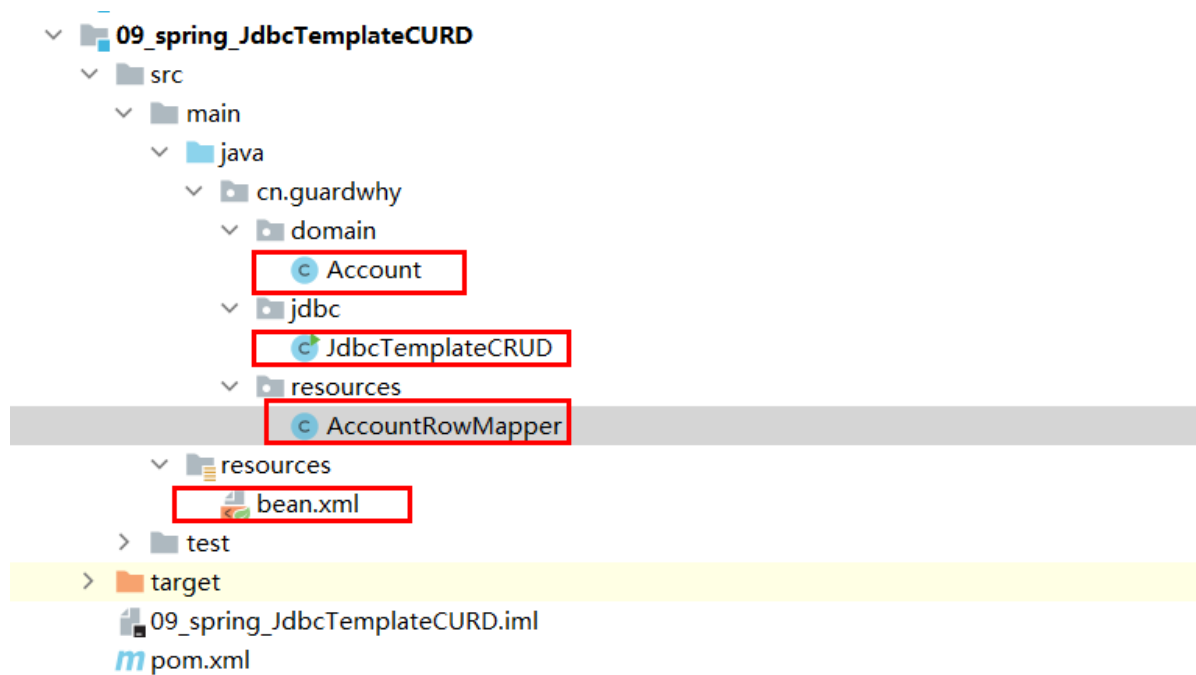
```
package cn.guardwhy.jdbc;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

/**
 * spring JdbcTemplate入门案例
 */
public class JdbcTemplateDemo {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring IOC容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.从spring IOC容器中,获取JdbcTemplate
        JdbcTemplate jdbcTemplate = (JdbcTemplate)
context.getBean("jdbcTemplate");
        // 3.添加账户信息
        jdbcTemplate.update("insert into account(name,money) values('候大
利',3500)");
    }
}
```

9- JdbcTemplate实现CRUD

9.1 项目目录



9.2 代码示例

账户实体类

```
package cn.guardwhy.domain;

/**
 * 账户实体类
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Account {
    private Integer id;
    private String name;
    private Float money;
}
```

测试代码

```
package cn.guardwhy.jdbc;

import cn.guardwhy.domain.Account;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
```

```

/**
 * spring JdbcTemplate实现完整CRUD操作
 */
public class JdbcTemplateCRUD {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.从spring IOC容器中,获取JdbcTemplate
        JdbcTemplate jdbcTemplate = (JdbcTemplate)
        context.getBean("jdbcTemplate");
        // 3.查询多行
        List<Account> list = jdbcTemplate.query("select id,name,money from
        account", new RowMapper<Account>() {
            @Override
            public Account mapRow(ResultSet rs, int index) throws SQLException {
                // 创建账户对象
                Account account = new Account();
                account.setId(rs.getInt("id"));
                account.setName(rs.getString("name"));
                account.setMoney(rs.getFloat("money"));
                return account;
            }
        });
        // 4.打印结果集
        for(Account account:list){
            System.out.println(account);
        }
        // 5.查询一行一列
        System.out.println("查询一行一列-----");
        Integer accountNum = jdbcTemplate.queryForObject("select count(*) from
        account", Integer.class);
        System.out.println("当前账户数量:" + accountNum);
    }
}

```

执行结果

```

Account{id=1, name='kebe', money=1000.0}
Account{id=2, name='curry', money=1500.0}
Account{id=3, name='james', money=2000.0}
Account{id=4, name='Duncan', money=2500.0}
Account{id=5, name='Gasol', money=3500.0}
Account{id=6, name='Gasol', money=3500.0}
查询一行一列-----
当前账户数量:6

```

9.3 自定义RowMapper

```
package cn.guardwhy.resources;

import cn.guardwhy.domain.Account;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public class AccountRowMapper implements RowMapper<Account> {
    /**
     * 结果集映射的方法：
     *      结果集中的每一行记录，都会调用一次该方法
     */
    public Account mapRow(ResultSet rs, int index) throws SQLException{
        // 创建账户对象
        Account account = new Account();
        account.setId(rs.getInt("id"));
        account.setName(rs.getString("name"));
        account.setMoney(rs.getFloat("money"));
        return account;
    }
}
```

9.4 测试代码

```
package cn.guardwhy.jdbc;

import cn.guardwhy.domain.Account;
import cn.guardwhy.resources.AccountRowMapper;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import java.util.List;

/**
 * spring JdbcTemplate实现完整CRUD操作
 */
public class JdbcTemplateCRUD {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring IOC容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.从spring IOC容器中,获取JdbcTemplate
        JdbcTemplate jdbcTemplate = (JdbcTemplate)
        context.getBean("jdbcTemplate");
        // 3.使用自定义的RowMapper
        List<Account> list = jdbcTemplate.query("select * from account", new
        AccountRowMapper());

        // 4.打印结果集
        for(Account account:list){
            System.out.println(account);
        }
    }
}
```

```

// 5.查询一行一列
System.out.println("查询一行一列-----");
Integer accountNum = jdbcTemplate.queryForObject("select count(*) from
account", Integer.class);
System.out.println("当前账户数量:" + accountNum);
}
}

```

9.5 执行结果

```

Run: JdbcTemplateCRUD x
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Account{id=1, name='curry', money=1000.0}
Account{id=2, name='james', money=2000.0}
Account{id=3, name='小王', money=2200.0}
Account{id=4, name='候大利', money=3500.0}
查询一行一列-----
当前账户数量:4

Process finished with exit code 0

```

10- JdbcTemplate整合数据源

10.1 常见数据源

在使用JdbcTemplate的时候，需要设置一个数据源对象，才能完成数据库操作

#常见的数据源：

内置数据源：

org.springframework.jdbc.datasource.DriverManagerDataSource

c3p0数据源：

com.mchange.v2.c3p0.ComboPooledDataSource

dbcp数据源：

org.apache.commons.dbcp.BasicDataSource

druid数据源：

com.alibaba.druid.pool.DruidDataSource

10.2 内置数据源

```

<?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">

<!--配置JdbcTemplate-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
<!--注入数据源对象-->

```

```

        <property name="dataSource" ref="dataSource"></property>
    </bean>

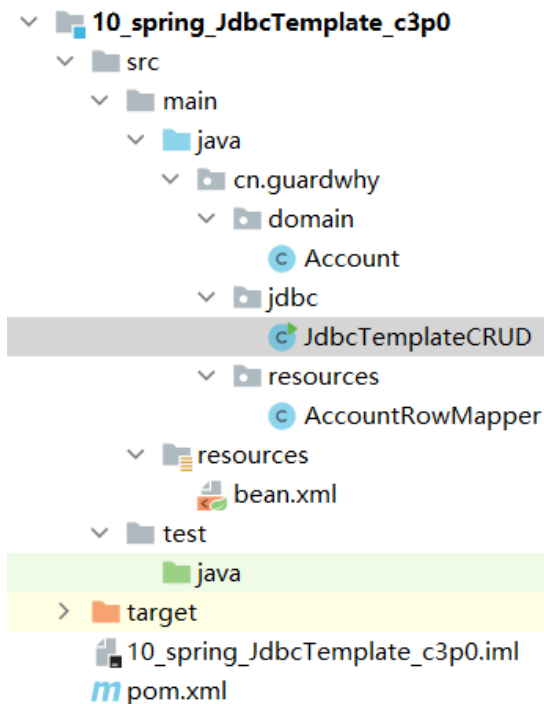
    <!--配置数据源对象-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--注入连接数据库的四个基本要素-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/1_spring">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="admin"></property>
    </bean>

</beans>

```

10.3 整合c3p0

10.3.1 项目结构



10.3.2 配置pom.xml

```

<!--c3p0版本-->
<c3p0.version>0.9.5</c3p0.version>

<!--c3p0依赖-->
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>${c3p0.version}</version>
</dependency>

```

10.3.3 配置bean.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">

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

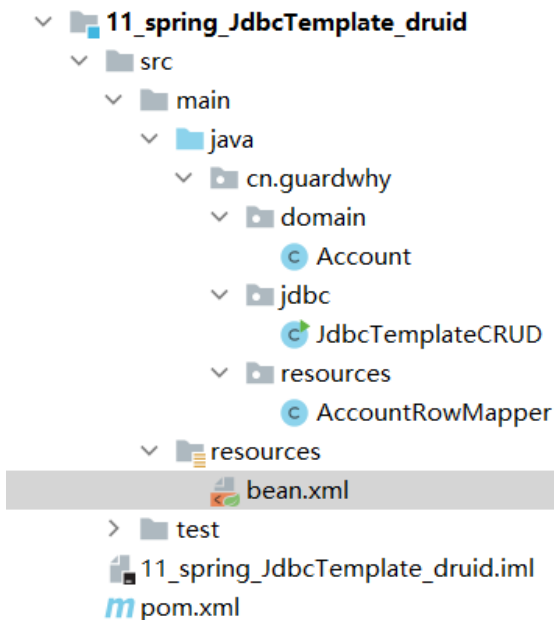
    <!--配置数据源(c3p0)对象-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <!--注入连接数据库-->
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3306/spring">
</property>
        <property name="user" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
</beans>
```

10.3.4 执行结果

```
信息: Initializing c3p0-0.9.5 [built 02-January-20
十月 25, 2020 6:11:44 下午 com.mchange.v2.c3p0.imp
信息: Initializing c3p0 pool... com.mchange.v2.c3p
Account{id=1, name='kebe', money=1000.0}
Account{id=2, name='curry', money=1500.0}
Account{id=3, name='james', money=2000.0}
Account{id=4, name='Duncan', money=2500.0}
Account{id=5, name='Gasol', money=3500.0}
Account{id=6, name='Gasol', money=3500.0}
查询一行一列-----
当前账户数量:6
```

10.4 整合druid数据源

10.4.1 项目结构



10.4.2 配置pom.xml

```
<!--druid版本-->
<druid.version>1.0.29</druid.version>

<!--druid依赖-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>${druid.version}</version>
</dependency>
```

10.4.3 配置bean.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">

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

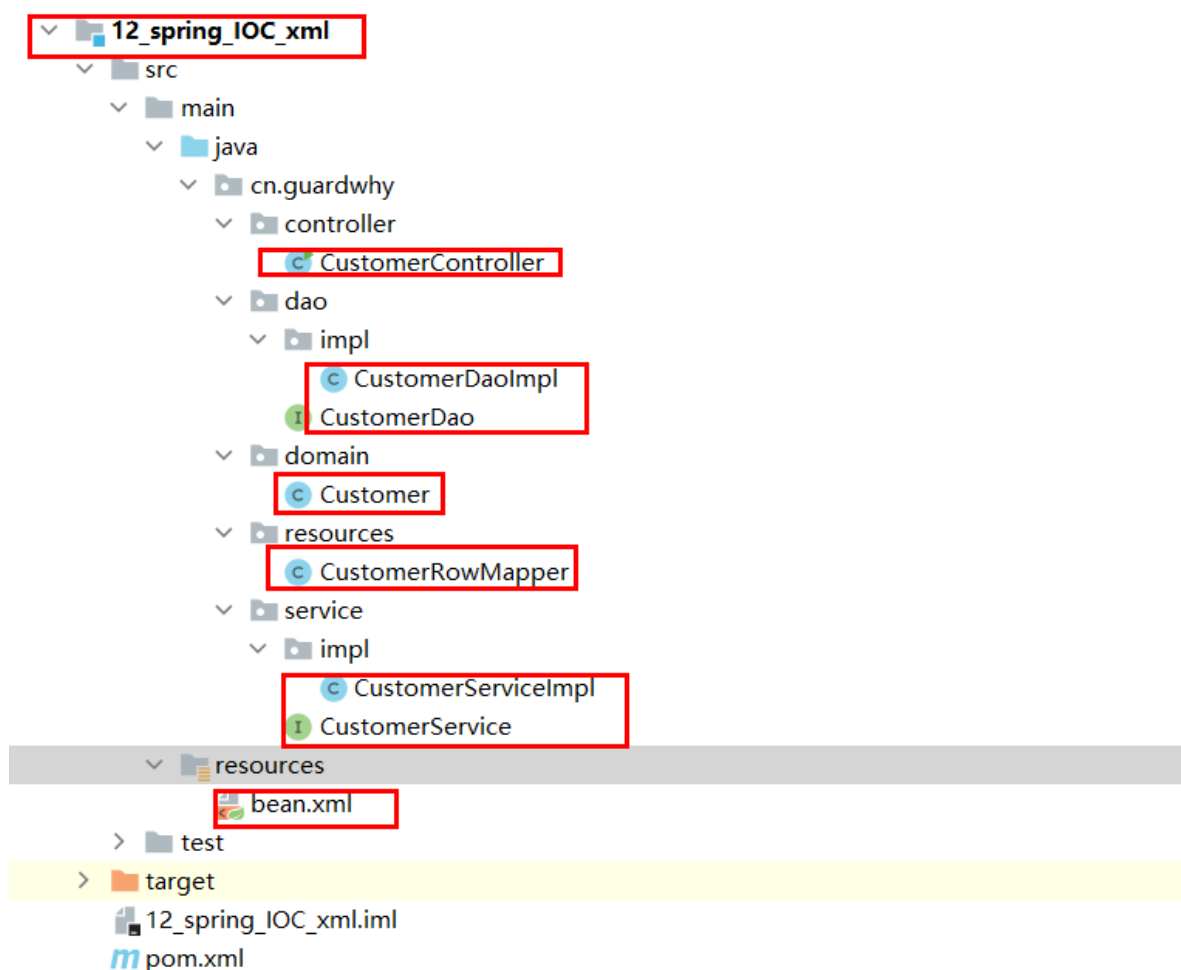
    <!--配置数据源对象(druid)-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!--注入连接数据库-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
</beans>
```


10.4.4 执行结果

```
十月 25, 2020 6:44:43 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} inited
Account{id=1, name='kebe', money=1000.0}
Account{id=2, name='curry', money=1500.0}
Account{id=3, name='james', money=2000.0}
Account{id=4, name='Duncan', money=2500.0}
Account{id=5, name='Gasol', money=3500.0}
Account{id=6, name='Gasol', money=3500.0}
查询一行一列-----
当前账户数量:6
```

11- Spring-IOC案例(xml实现)

11.1 项目目录



11.2 代码示例

客户实体类

```
package cn.guardwhy.domain;
/**
 * 客户实体类
 */
@Data
```

```

@NoArgsConstructor
@AllArgsConstructor
public class Customer {
    // 成员变量
    private Long custId;
    private String custName;
    private String custSource;
    private String custIndustry;
    private String custLevel;
    private String custAddress;
    private String custPhone;
}

```

结果集映射

```

package cn.guardwhy.resources;

import cn.guardwhy.domain.Customer;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public class CustomerRowMapper implements RowMapper<Customer> {
    /**
     * 结果集映射的方法：
     * 结果集中的每一行记录，都会调用一次该方法
     */
    public Customer mapRow(ResultSet rs, int index) throws SQLException{
        // 创建客户对象
        Customer customer = new Customer();
        customer.setCustId(rs.getLong("cust_id"));
        customer.setCustName(rs.getString("cust_name"));
        customer.setCustSource(rs.getString("cust_source"));
        customer.setCustIndustry(rs.getString("cust_industry"));
        customer.setCustLevel(rs.getString("cust_level"));
        customer.setCustAddress(rs.getString("cust_address"));
        customer.setCustPhone(rs.getString("cust_phone"));
        return customer;
    }
}

```

持久层dao

CustomerDao

```

package cn.guardwhy.dao;

import cn.guardwhy.domain.Customer;

import java.util.List;

public interface CustomerDao {
    /**
     * 查询全部客户
     */
    List<Customer> findAllCustomers();
}

```

CustomerDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.resources.CustomerRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import java.util.List;
/**
 * 客户dao实现类
 */
public class CustomerDaoImpl implements CustomerDao {
    // 定义JdbcTemplate
    private JdbcTemplate jdbcTemplate;

    // 设置方法
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    /**
     * 查询全部客户列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        // 定义sql
        String sql = "select t.cust_id, t.cust_name, t.cust_source, "+
            "t.cust_industry, t.cust_level, t.cust_address, t.cust_phone "+
            "from cst_customer t";

        // 执行查询操作
        List<Customer> list = jdbcTemplate.query(sql, new CustomerRowMapper());
        return list;
    }
}

```

业务层Service

CustomerService

```

package cn.guardwhy.service;

```

```

import cn.guardwhy.domain.Customer;

import java.util.List;
/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 查询全部客户列表
     */
    List<Customer> findAllCustomers();
}

```

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;

import java.util.List;
/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    private CustomerDao customerDao;

    // 设置方法
    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }

    /**
     * 查询全部客户的列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        return customerDao.findAllCustomers();
    }
}

```

配置bean.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">

    <!--配置客户dao-->
    <bean id="customerDao" class="cn.guardwhy.dao.impl.CustomerDaoImpl">
        <!--注入jdbcTemplate-->
    
```

```

        <property name="jdbcTemplate" ref="jdbcTemplate"></property>
    </bean>

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl">
        <!--注入客户dao-->
        <property name="customerDao" ref="customerDao"></property>
    </bean>

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--配置数据源对象(druid)-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!--注入连接数据库-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>

        <!--数据库连接池常用属性-->
        <!--初始化连接数量-->
        <property name="initialSize" value="6" />
        <!-- 最小空闲连接数 -->
        <property name="minIdle" value="3" />
        <!-- 最大并发连接数(最大连接池数量) -->
        <property name="maxActive" value="50" />
        <!-- 配置获取连接等待超时的时间 -->
        <property name="maxWait" value="60000" />
        <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
    </bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * 客户表现层
 */
public class CustomerController {

```

```

    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.获取客户service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.查询全部客户列表数据
        List<Customer> list = customerService.findAllCustomers();
        // 4.遍历操作
        for(Customer customer : list){
            System.out.println(customer);
        }
    }
}

```

11.3 执行结果

```

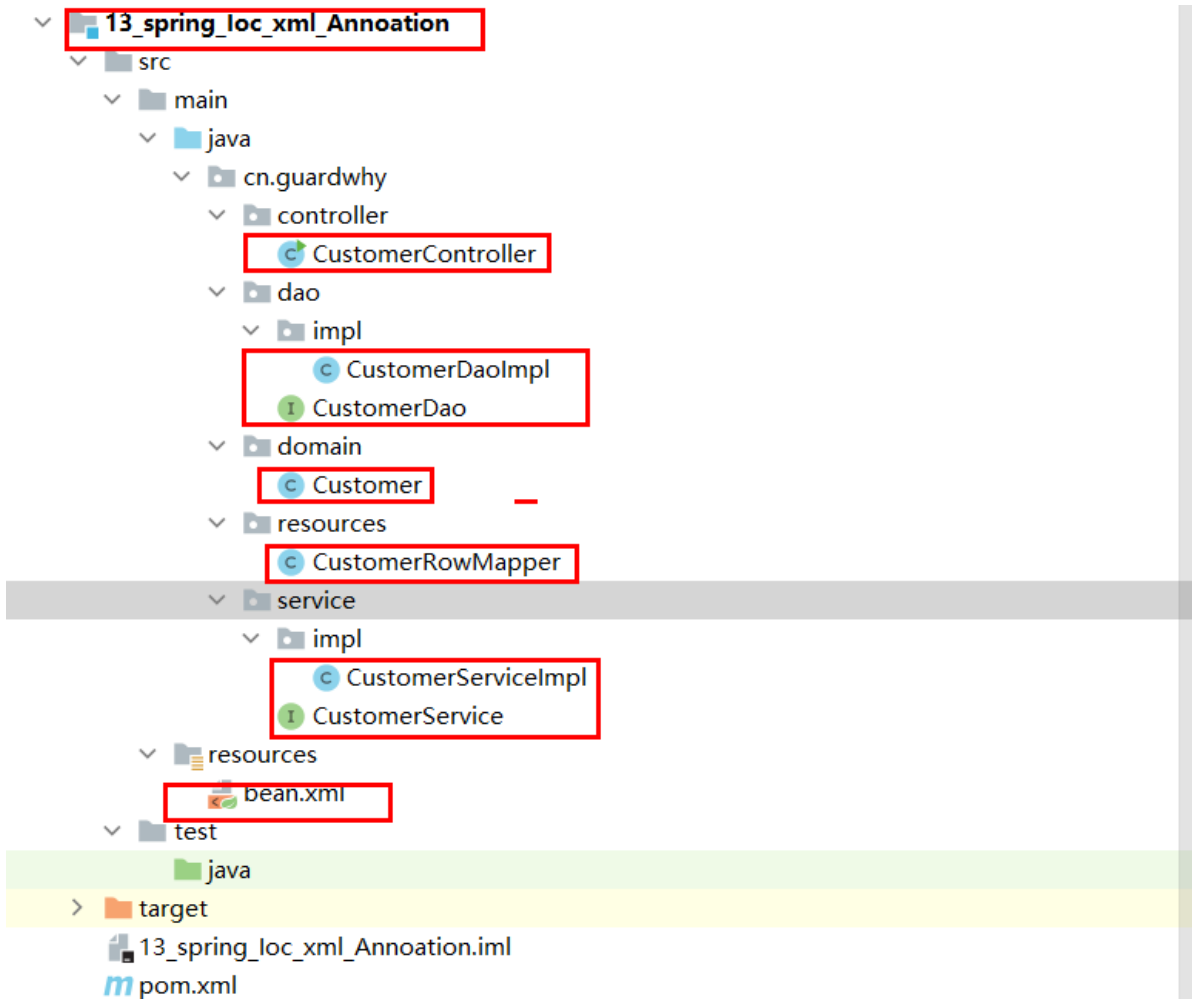
Run: CustomerController
三月 19, 2021 3:29:37 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} initied
Customer{custId=1, custName='美团', custSource='网络营销', custIndustry='互联网', custLevel='普通客户', custAddress='北京市海淀区', custPhone='7758258'}
Customer{custId=2, custName='360', custSource='网络安全', custIndustry='互联网', custLevel='普通客户', custAddress='北京市朝阳区', custPhone='0208888887'}
Customer{custId=3, custName='百度', custSource='网络搜索', custIndustry='互联网', custLevel='普通客户', custAddress='北京朝阳区', custPhone='389056'}
Customer{custId=4, custName='小米', custSource='手机制造', custIndustry='互联网+', custLevel='VIP', custAddress='武汉光谷', custPhone='2267890'}
Customer{custId=5, custName='腾讯', custSource='游戏社交', custIndustry='互联网', custLevel='VIP', custAddress='深圳市南山区', custPhone='123456'}
Customer{custId=6, custName='华为', custSource='通信手机', custIndustry='高科技制造业', custLevel='VIP', custAddress='深圳龙岗', custPhone='033567'}

Process finished with exit code 0

```

12- Spring-IOC案例(xml注解整合)

12.1 项目目录



12.2 代码示例

客户实体类

```
package cn.guardwhy.domain;
/**
 * 客户实体类
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer {
    // 成员变量
    private Long custId;
    private String custName;
    private String custSource;
    private String custIndustry;
    private String custLevel;
    private String custAddress;
    private String custPhone;
}
```

结果集映射

```
package cn.guardwhy.resources;

import cn.guardwhy.domain.Customer;
```

```

import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public class CustomerRowMapper implements RowMapper<Customer> {
    /**
     * 结果集映射的方法：
     *      结果集中的每一行记录，都会调用一次该方法
     */
    public Customer mapRow(ResultSet rs, int index) throws SQLException{
        // 创建客户对象
        Customer customer = new Customer();
        customer.setCustId(rs.getLong("cust_id"));
        customer.setCustName(rs.getString("cust_name"));
        customer.setCustSource(rs.getString("cust_source"));
        customer.setCustIndustry(rs.getString("cust_industry"));
        customer.setCustLevel(rs.getString("cust_level"));
        customer.setCustAddress(rs.getString("cust_address"));
        customer.setCustPhone(rs.getString("cust_phone"));
        return customer;
    }
}

```

持久层dao

CustomerDao

```

package cn.guardwhy.dao;

import cn.guardwhy.domain.Customer;

import java.util.List;

public interface CustomerDao {
    /**
     * 查询全部客户
     */
    List<Customer> findAllCustomers();
}

```

CustomerDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.resources.CustomerRowMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;
/**
 * 客户dao实现类

```



```

*/
@Repository("customerDao")
public class CustomerDaoImpl implements CustomerDao {
    // 定义JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 查询全部客户列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        // 定义sql
        String sql = "select t.cust_id, t.cust_name, t.cust_source, "+
            "t.cust_industry, t.cust_level, t.cust_address, t.cust_phone "+
            "from cst_customer t";

        // 执行查询操作
        List<Customer> list = jdbcTemplate.query(sql, new CustomerRowMapper());
        return list;
    }
}

```

业务层Service

CustomerService

```

package cn.guardwhy.service;

import cn.guardwhy.domain.Customer;

import java.util.List;
/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 查询全部客户列表
     */
    List<Customer> findAllCustomers();
}

```

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
/**

```

```

    * 客户service实现类
    */
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;

    /**
     * 查询全部客户的列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        return customerDao.findAllCustomers();
    }
}

```

配置bean.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">

    <!--配置包扫描dao/service
        第一步：导入context名称空间和约束
        第二步：通过<context:component-scan>标签配置包扫描，spring框架在
        初始化IOC容器的时候，会扫描指定的包和它的子包
    -->
    <context:component-scan base-package="cn.guardwhy"></context:component-scan>

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--配置数据源对象(druid)-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!--注入连接数据库-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>

        <!--数据库连接池常用属性-->
        <!--初始化连接数量-->
        <property name="initialSize" value="6" />
        <!-- 最小空闲连接数 -->
        <property name="minIdle" value="3" />
        <!-- 最大并发连接数(最大连接池数量) -->

```

```

<property name="maxActive" value="50" />
<!-- 配置获取连接等待超时的时间 -->
<property name="maxWait" value="60000" />
<!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
<property name="timeBetweenEvictionRunsMillis" value="60000" />
</bean>
</beans>

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * 客户表现层
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.获取客户service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3.查询全部客户列表数据
        List<Customer> list = customerService.findAllCustomers();
        // 4.遍历操作
        for(Customer customer : list){
            System.out.println(customer);
        }
    }
}

```

12.3 执行结果

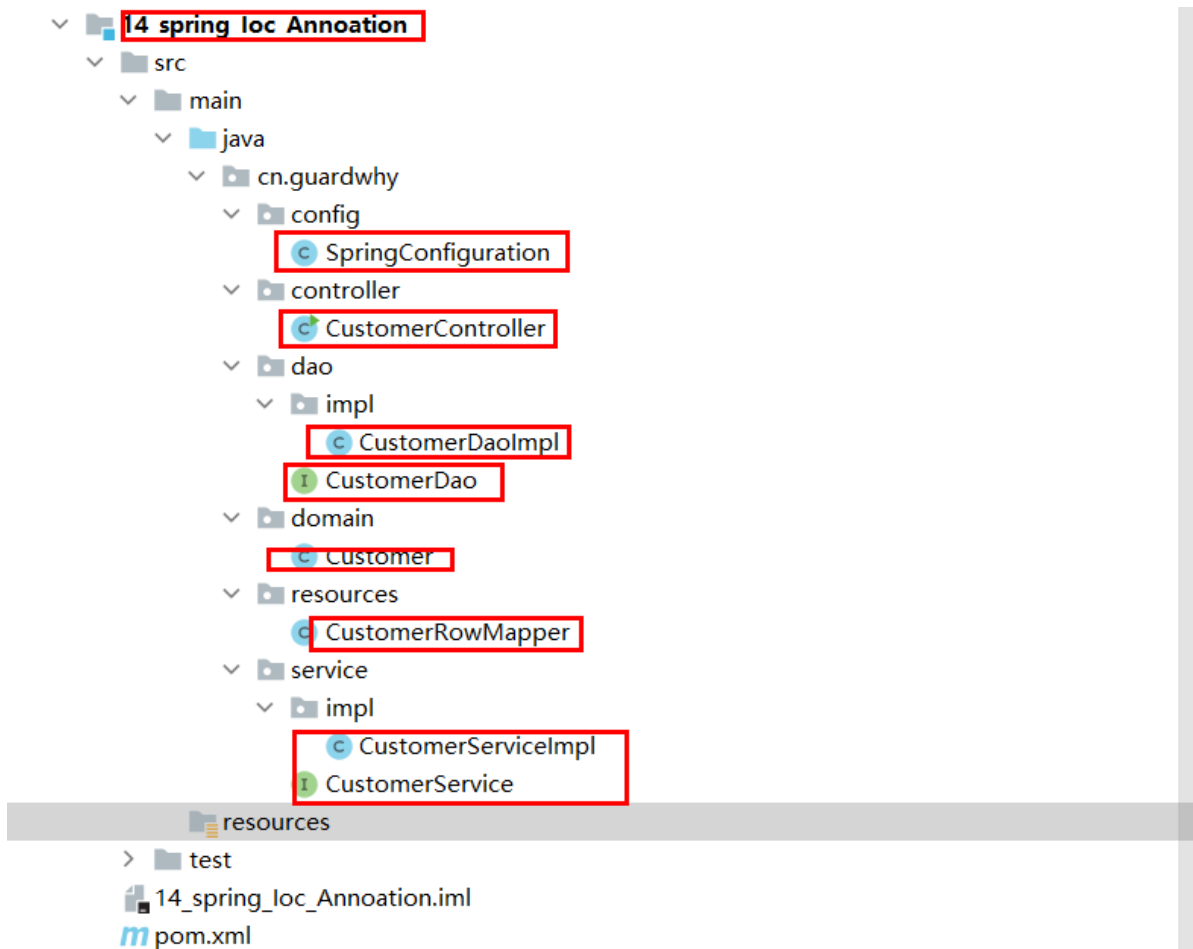
```

Run: CustomerController
三月 19, 2021 3:29:37 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} inited
Customer{custId=1, custName='美团', custSource='网络营销', custIndustry='互联网', custLevel='普通客户', custAddress='北京市海淀区', custPhone='7758258'}
Customer{custId=2, custName='360', custSource='网络安全', custIndustry='互联网', custLevel='普通客户', custAddress='北京市朝阳区', custPhone='0208888887'}
Customer{custId=3, custName='百度', custSource='网络搜索', custIndustry='互联网', custLevel='普通客户', custAddress='北京朝阳区', custPhone='389056'}
Customer{custId=4, custName='小米', custSource='手机制造', custIndustry='互联网+', custLevel='VIP', custAddress='武汉光谷', custPhone='2267890'}
Customer{custId=5, custName='腾讯', custSource='游戏社交', custIndustry='互联网', custLevel='VIP', custAddress='深圳市南山区', custPhone='123456'}
Customer{custId=6, custName='华为', custSource='通信手机', custIndustry='高新科技制造业', custLevel='VIP', custAddress='深圳龙岗', custPhone='033567'}
Process finished with exit code 0

```

13-Spring-IOC案例(注解实现)

13.1 项目目录



12.2 代码示例

客户实体类

```
package cn.guardwhy.domain;
/**
 * 客户实体类
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer {
    // 成员变量
    private Long custId;
    private String custName;
    private String custSource;
    private String custIndustry;
    private String custLevel;
    private String custAddress;
    private String custPhone;
}
```

结果集映射

```
package cn.guardwhy.resources;
```

```

import cn.guardwhy.domain.Customer;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public class CustomerRowMapper implements RowMapper<Customer> {
    /**
     * 结果集映射的方法:
     *      结果集中的每一行记录, 都会调用一次该方法
     */
    public Customer mapRow(ResultSet rs, int index) throws SQLException{
        // 创建客户对象
        Customer customer = new Customer();
        customer.setCustId(rs.getLong("cust_id"));
        customer.setCustName(rs.getString("cust_name"));
        customer.setCustSource(rs.getString("cust_source"));
        customer.setCustIndustry(rs.getString("cust_industry"));
        customer.setCustLevel(rs.getString("cust_level"));
        customer.setCustAddress(rs.getString("cust_address"));
        customer.setCustPhone(rs.getString("cust_phone"));
        return customer;
    }
}

```

持久层dao

CustomerDao

```

package cn.guardwhy.dao;

import cn.guardwhy.domain.Customer;

import java.util.List;

public interface CustomerDao {
    /**
     * 查询全部客户
     */
    List<Customer> findAllCustomers();
}

```

CustomerDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.resources.CustomerRowMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;
/**

```

```

    * 客户dao实现类
    */
@Repository("customerDao")
public class CustomerDaoImpl implements CustomerDao {
    // 定义JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 查询全部客户列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        // 定义sql
        String sql = "select t.cust_id, t.cust_name, t.cust_source, "+
            "t.cust_industry, t.cust_level, t.cust_address, t.cust_phone "+
            "from cst_customer t";

        // 执行查询操作
        List<Customer> list = jdbcTemplate.query(sql, new CustomerRowMapper());
        return list;
    }
}

```

业务层Service

CustomerService

```

package cn.guardwhy.service;

import cn.guardwhy.domain.Customer;

import java.util.List;
/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 查询全部客户列表
     */
    List<Customer> findAllCustomers();
}

```

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.CustomerDao;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

```

```

/**
 * 客户service实现类
 */
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
    // 定义客户dao
    @Autowired
    private CustomerDao customerDao;

    /**
     * 查询全部客户的列表
     */
    @Override
    public List<Customer> findAllCustomers() {
        return customerDao.findAllCustomers();
    }
}

```

注解配置类

SpringConfiguration

```

package cn.guardwhy.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;

/**
 * spring配置类:
 * 注解说明:
 *      1.@Configuration: 标记当前java类, 是一个spring的配置类
 *      2.@ComponentScan: 配置扫描包。相当于xml配置中<context:component-scan/>标签
 */
@Configuration
@ComponentScan(value = {"cn.guardwhy"})
public class SpringConfiguration {
    /**
     * <!--配置JdbcTemplate-->
     * <bean id="jdbcTemplate"
     class="org.springframework.jdbc.core.JdbcTemplate">
     * <!--注入数据源对象-->
     * <property name="dataSource" ref="dataSource"></property>
     * </bean>
     *
     * 说明:
     *      1.创建一个JdbcTemplate对象
     *      2.给该JdbcTemplate对象, 注入一个数据源对象DataSource
     *      3.将该JdbcTemplate对象, 以jdbcTemplate作为bean的名称, 放入spring容器中
     *
     * 注解:
     *      @Bean:
     作用: 把方法的返回值对象, 放入spring容器中
     */
}

```

```

    */
    @Bean(value = "jdbcTemplate")
    public JdbcTemplate createJdbcTemplate(DataSource dataSource){
        // 创建JdbcTemplate对象
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    /**
     * 配置数据源对象DataSource
     */
    @Bean(value = "dataSource")
    public DataSource createDataSource(){
        // 创建DataSource
        DruidDataSource dataSource = new DruidDataSource();
        // 注入属性
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/spring");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        dataSource.setInitialSize(6);
        dataSource.setMinIdle(3);
        dataSource.setMaxActive(50);
        dataSource.setMaxWait(60000);
        dataSource.setTimeBetweenEvictionRunsMillis(60000);
        return dataSource;
    }
}

```

表现层(Controller)

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.config.SpringConfiguration;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * 客户表现层
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
        AnnotationConfigApplicationContext(SpringConfiguration.class);
        // 2.获取客户service对象
    }
}

```



```

        CustomerService customerService = (CustomerService)
context.getBean("customerService");
        // 3.查询全部客户列表数据
        List<Customer> list = customerService.findAllCustomers();
        // 4.遍历操作
        for(Customer customer : list){
            System.out.println(customer);
        }
    }
}

```

12.3 执行结果



```

Run: CustomerController
三月 19, 2021 3:29:37 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} initied
Customer{custId=1, custName='美团', custSource='网络营销', custIndustry='互联网', custLevel='普通客户', custAddress='北京市海淀区', custPhone='7758258'}
Customer{custId=2, custName='360', custSource='网络安全', custIndustry='互联网', custLevel='普通客户', custAddress='北京市朝阳区', custPhone='0208888887'}
Customer{custId=3, custName='百度', custSource='网络搜索', custIndustry='互联网', custLevel='普通客户', custAddress='北京朝阳区', custPhone='389056'}
Customer{custId=4, custName='小米', custSource='手机制造', custIndustry='互联网+', custLevel='VIP', custAddress='武汉光谷', custPhone='2267890'}
Customer{custId=5, custName='腾讯', custSource='游戏社交', custIndustry='互联网', custLevel='VIP', custAddress='深圳市南山区', custPhone='123456'}
Customer{custId=6, custName='华为', custSource='通信手机', custIndustry='高新科技制造业', custLevel='VIP', custAddress='深圳龙岗', custPhone='033567'}

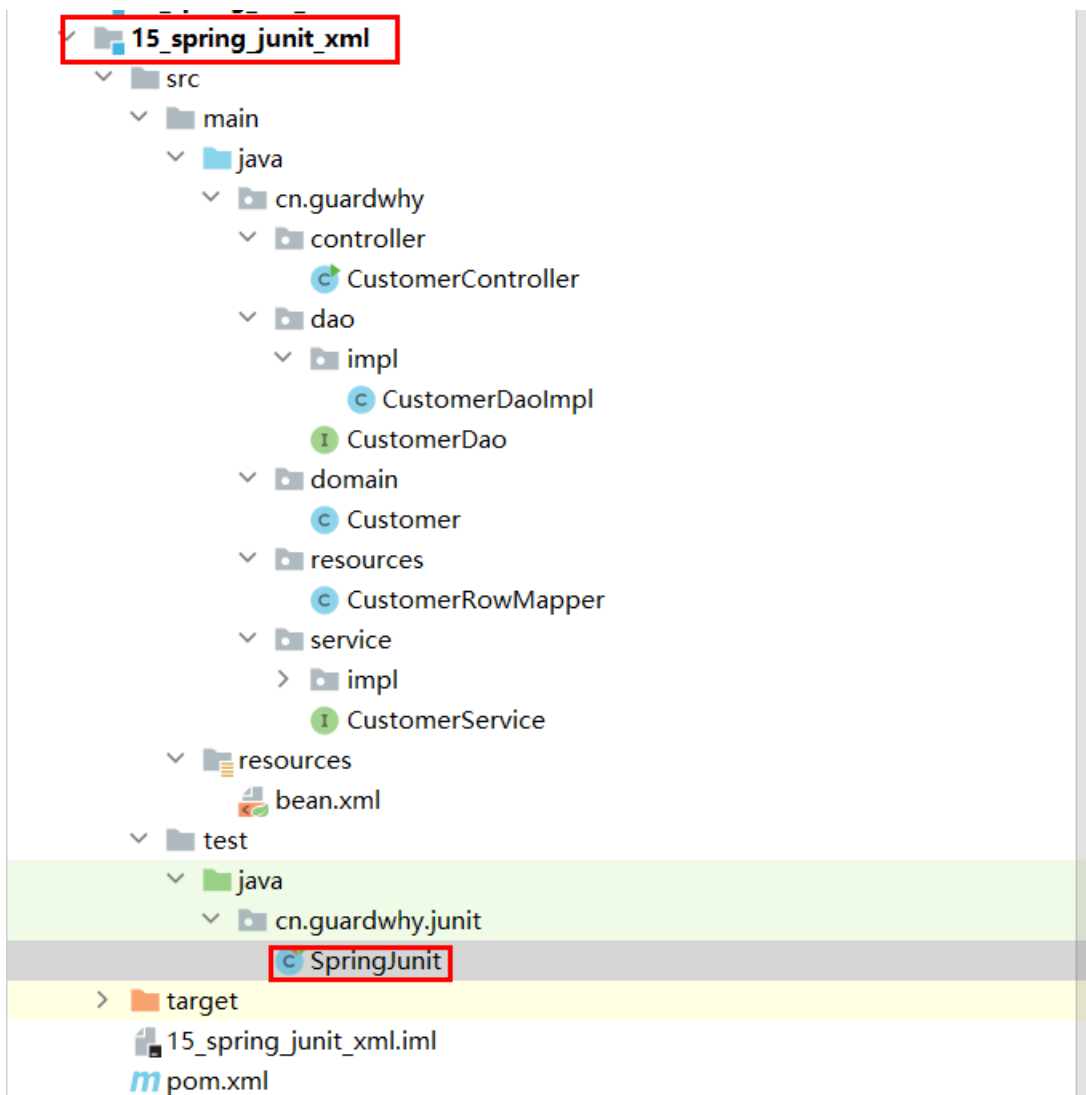
Process finished with exit code 0

```

14-Spring整合junit

14.1 基于xml版本

14.1.1 项目目录



14.1.2 配置pom.xml

```
<!--junit 版本-->
<junit.version>4.12</junit.version>

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

<!--junit依赖-->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
</dependency>
```

12.1.3 编写测试类

```
package cn.guardwhy.junit;

import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
```

```

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

import java.util.List;

/**
 * spring整合junit单元测试框架
 * 整合步骤:
 * 1. 导入junit单元测试框架包
 * 2. 导入spring提供的测试包(spring-test)
 * 3. 通过@RunWith注解, 把junit的测试类, 替换成spring提供的测试类
 *    (SpringJUnit4ClassRunner)
 * 4. 通过@ContextConfiguration注解, 加载spring的配置文件
 */
@RunWith(value = SpringJUnit4ClassRunner.class)
@ContextConfiguration(value = {"classpath:bean.xml"})

public class SpringJUnit {
    /**
     * 测试查询全部客户列表数据
     */
    @Test
    public void findAllCustomersTest1(){
        // 1. 加载spring配置文件, 创建spring容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2. 获取客户service对象
        CustomerService customerService = (CustomerService)
        context.getBean("customerService");
        // 3. 查询全部客户列表数据
        List<Customer> list = customerService.findAllCustomers();
        for(Customer c:list){
            System.out.println(c);
        }
    }

    @Autowired
    private CustomerService customerService;
    @Test
    public void findAllCustomersTest2(){
        // 查询客户列表
        List<Customer> list = customerService.findAllCustomers();
        for(Customer c:list){
            System.out.println(c);
        }
    }
}

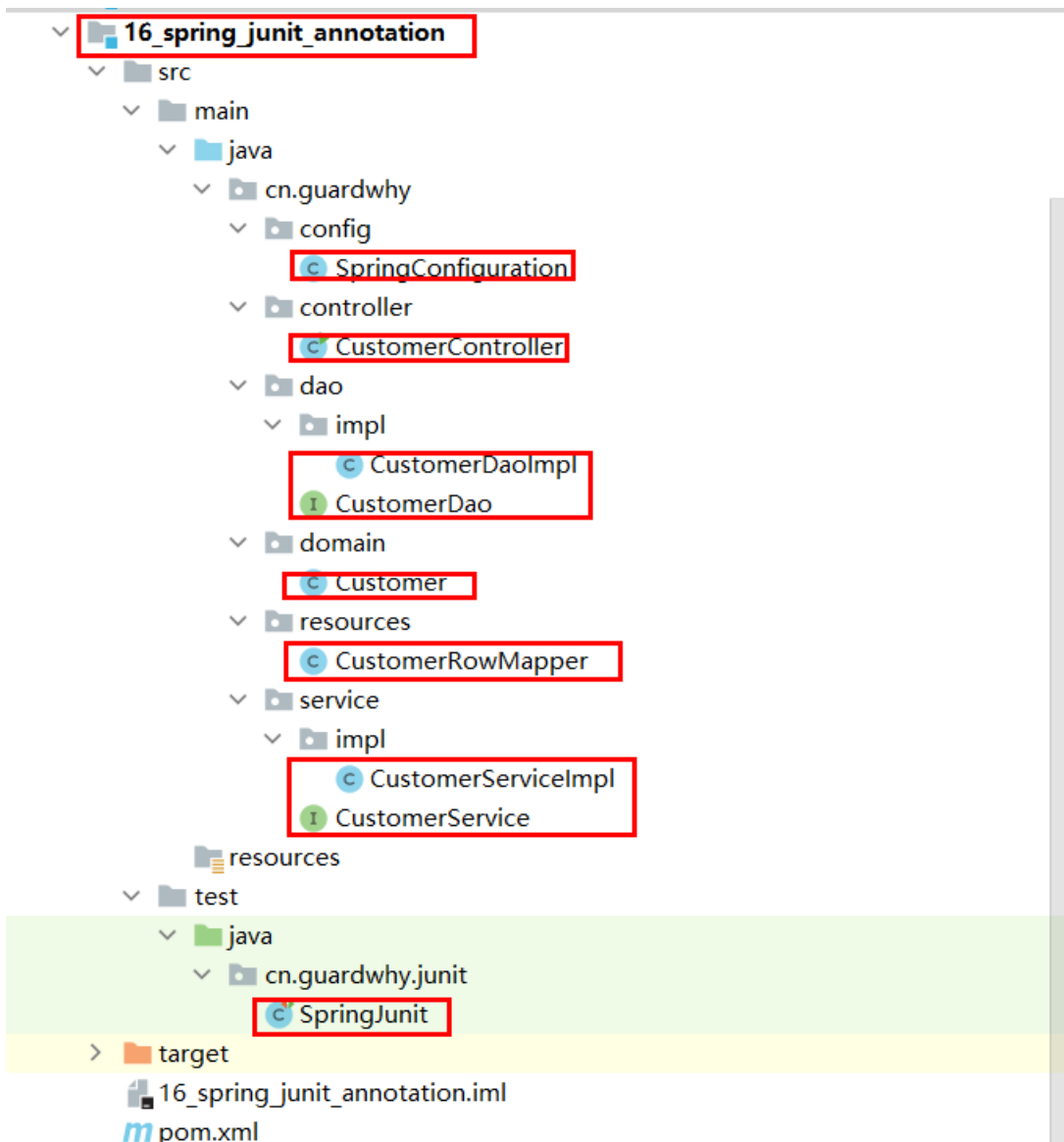
```

14.1.4 执行结果

```
十月 26, 2020 12:13:05 上午 com.alibaba.druid.support.logging.JakartaCommonsLoggingI
信息: {dataSource-1} initied
Customer{custId=1, custName='美团外卖', custSource='网络营销', custIndustry='互联网', c
Customer{custId=2, custName='360公司', custSource='网络安全', custIndustry='互联网', c
Customer{custId=3, custName='百度', custSource='网络搜索', custIndustry='互联网', cust
Customer{custId=4, custName='小米', custSource='手机制造', custIndustry='互联网+', cus
Customer{custId=5, custName='腾讯', custSource='网络广告', custIndustry='互联网', cust
Customer{custId=6, custName='华为', custSource='电视广告', custIndustry='高新科技制造业
```

14.2 基于注解版本

14.2.1 项目目录



14.2.2 编写测试类

```
package cn.guardwhy.junit;

import cn.guardwhy.config.SpringConfiguration;
import cn.guardwhy.domain.Customer;
import cn.guardwhy.service.CustomerService;
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;

import java.util.List;

/**
 * spring整合junit单元测试框架
 *
 * 整合步骤:
 *    1. 导入junit单元测试框架包
 *    2. 导入spring提供的测试包(spring-test)
 *    3. 通过@RunWith注解, 把junit的测试类, 替换成spring提供的测试类
 *       (SpringJUnit4ClassRunner)
 *    4. 通过@ContextConfiguration注解, 加载spring的配置类
 */
@RunWith(value = SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringConfiguration.class})

public class SpringJunit {

    // 定义客户service
    @Autowired
    private CustomerService customerService;

    /**
     * 测试查询全部客户列表数据
     */
    @Test
    public void findAllCustomersTest2(){
        // 查询客户列表
        List<Customer> list = customerService.findAllCustomers();
        for(Customer c:list){
            System.out.println(c);
        }
    }
}

```

执行结果

```

Run: SpringJunit.findAllCustomersTest2 (1)
Tests passed: 1 of 1 test - 260 ms
initiated
Customer{custId=1, custName='美团', custSource='网络营销', custIndustry='互联网', custLevel='普通客户', custAddress='北京市海淀区', custPhone='7758258'}
Customer{custId=2, custName='360', custSource='网络安全', custIndustry='互联网', custLevel='普通客户', custAddress='北京市朝阳区', custPhone='828888887'}
Customer{custId=3, custName='百度', custSource='网络搜索', custIndustry='互联网', custLevel='普通客户', custAddress='北京朝阳区', custPhone='389856'}
Customer{custId=4, custName='小米', custSource='手机制造', custIndustry='互联网+', custLevel='VIP', custAddress='武汉光谷', custPhone='2267898'}
Customer{custId=5, custName='腾讯', custSource='游戏社交', custIndustry='互联网', custLevel='VIP', custAddress='深圳南山区', custPhone='123456'}
Customer{custId=6, custName='华为', custSource='通信手机', custIndustry='高科技制造业', custLevel='VIP', custAddress='深圳龙岗', custPhone='833567'}
Process finished with exit code 0

```

15-Spring AOP

15.1 AOP基本概念

15.1.1 AOP概述

AOP (Aspect Oriented Programming) , 即面向切面编程。

- 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的延续, 是软件系统开发中的一个热点, 也是spring框架的一个重点。

- 利用AOP可以实现业务逻辑各个部分的隔离，从而使得业务逻辑各个部分的耦合性降低，提高程序的可重用性，同时提高开发效率。

简单理解：

aop是面向切面编程，使用动态代理技术，实现在不修改java源代码的情况下，运行时实现方法功能的增强。

15.1.2 AOP作用

#作用

使用动态代理技术，在程序运行期间，不修改java源代码对已有方法功能进行增强

#优势

- 1.减少重复代码，提高开发效率。
- 2.统一管理统一调用，方便维护。

15.1.3 AOP常用术语

#Joinpoint (连接点)

在spring中，连接点指的都是方法（指的是那些要被增强功能的候选方法）

#Pointcut (切入点)

在运行时已经被spring 的AOP实现了增强的方法。

#Advice (通知)

通知指的是拦截到joinpoint之后要做的事情。即增强的功能。

通知类型：前置通知、后置通知、异常通知、最终通知、环绕通知。

#Target(目标对象)

被代理的对象。比如动态代理案例中的演员。

#Proxy (代理)

一个类被AOP织入增强后，即产生一个结果代理类。

#Weaving (织入)

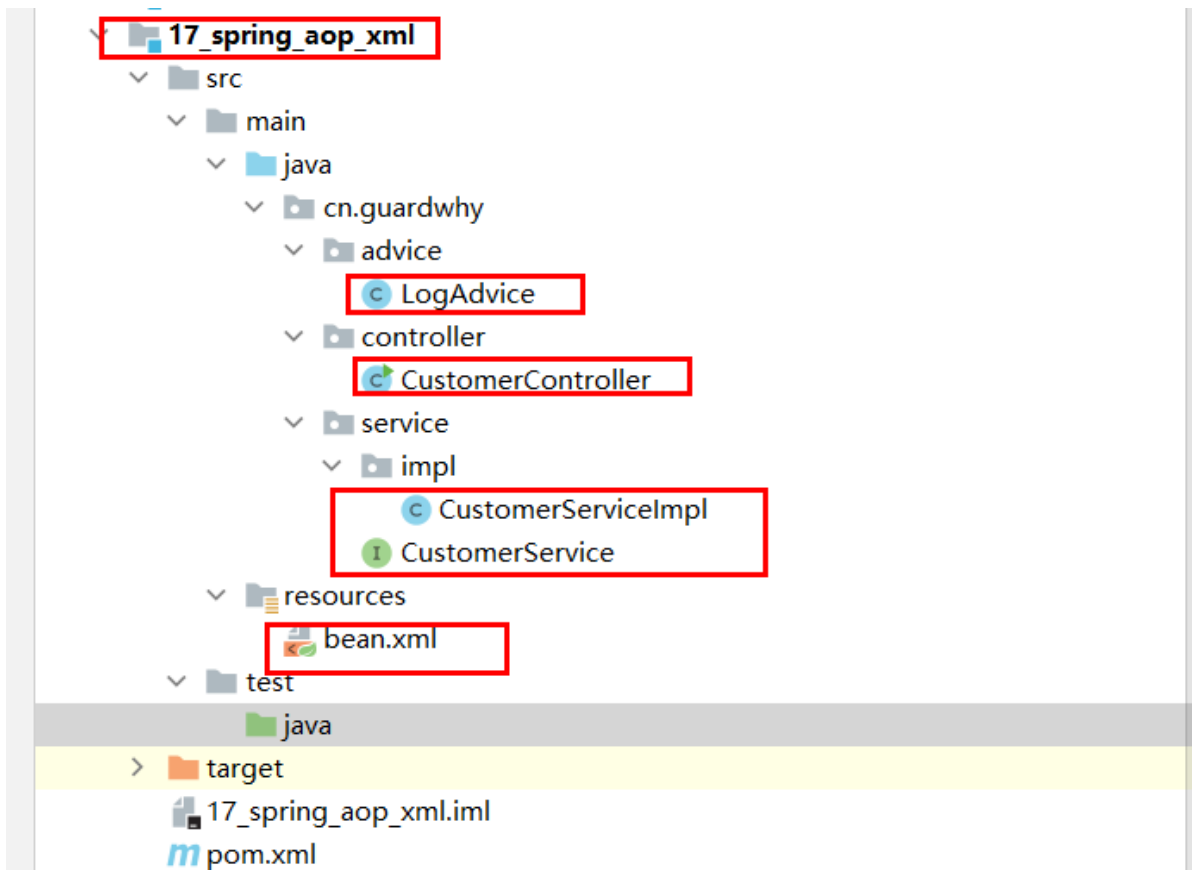
织入指的是把增强用于目标对象。创建代理对象的过程。

#Aspect(切面)

切面指的是切入点和通知的结合

15.2 完全xml配置AOP

15.2.1 项目目录



15.2.2 配置pom.xml

```
<properties>
  <!--spring版本-->
  <spring.version>5.0.2.RELEASE</spring.version>
  <!--aopalliance版本-->
  <aopalliance.version>1.0</aopalliance.version>
</properties>

<dependencies>
  <!-- spring aspects依赖 -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <!--aopalliance依赖-->
  <dependency>
    <groupId>aopalliance</groupId>
    <artifactId>aopalliance</artifactId>
    <version>${aopalliance.version}</version>
  </dependency>
</dependencies>
```

15.2.3 代码示例

CustomerService

```
package cn.guardwhy.service;
/**
 * 客户服务接口
```

```

    */
    public interface CustomerService {
        /**
         * 保存客户
         */
        void saveCustomer();

        /**
         * 根据客户id查询客户
         */
        void findCustomerById(Integer id);
    }

```

CustomerServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.service.CustomerService;
/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

    @Override
    public void findCustomerById(Integer id) {
        // 根据客户id查询客户
        System.out.println("根据客户id查询客户,客户id: " + id);
    }
}

```

LogAdvice

```

package cn.guardwhy.advice;
/**
 * 日志通知
 */
public class LogAdvice {
    /**
     * 记录用户操作日志
     */
    public void printLog(){
        System.out.println("记录用户操作日志");
    }
}

```

编写bean.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">
    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置日志通知-->
    <bean id="logAdvice" class="cn.guardwhy.advice.LogAdvice"></bean>

    <!--第一步:通过aop:config声明aop的配置-->
    <aop:config>
        <!--第二步: 通过aop:aspect配置切面,说明:
            id: 唯一标识名称
            ref: 指定通知bean对象的引用
        -->
        <aop:aspect id="logAspect" ref="logAdvice">
            <!--第三步: 通过aop:before配置前置通知,说明:
                method: 指定通知方法名称
                pointcut-ref: 指定切入点表达式
            -->
            <aop:before method="printLog" pointcut-ref="pt1"></aop:before>
            <!--第四步: 通过aop:pointcut配置切入点表达式,说明:
                id: 唯一标识名称
                expression: 指定切入点表达式
                切入点表达式组成:
                    访问修饰符 返回值 包名称 类名称 方法名称 (参数列表)
            -->
            <aop:pointcut id="pt1" expression="execution(public void
cn.guardwhy.service.impl.CustomerServiceImpl.saveCustomer())"/>
            </aop:aspect>
        </aop:config>
    </beans>

```

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户表现层
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.获取客户service
        CustomerService customerService = (CustomerService)
context.getBean("customerService");
        // 3.保存客户
        customerService.saveCustomer();
    }
}

```

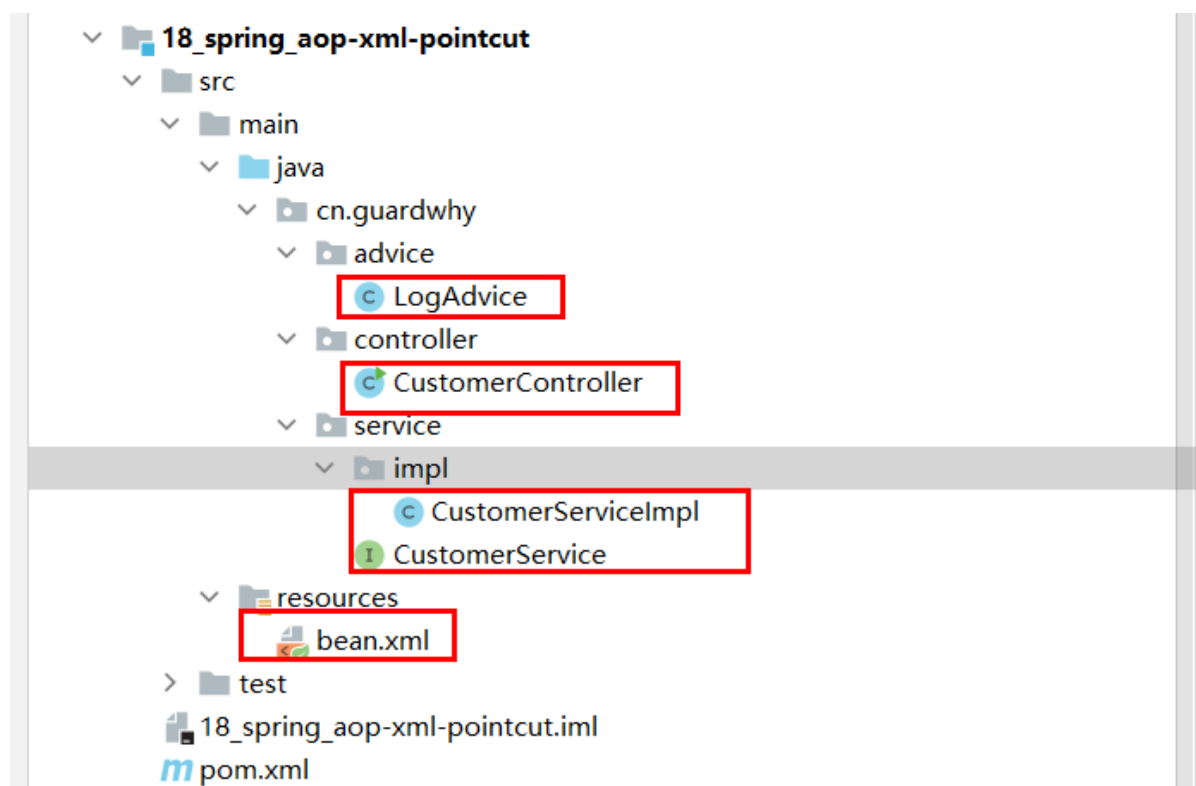
执行结果

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe
记录用户操作日志
保存客户操作

Process finished with exit code 0
```

15.3 切入点表达式

15.3.1 项目目录



13.3.2 常用标签说明

#<aop:config>

作用：声明aop配置

#<aop:aspect>:

作用：配置切面

属性：id：唯一标识切面的名称 ref：引用通知类bean的id

#<aop:pointcut>

作用：配置切入点表达式

属性：id：唯一标识切入点表达式名称 expression：定义切入点表达式

15.3.3 代码示例

CustomerService

```
package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {

    /**
     * 保存客户
     */
    void saveCustomer();

    /**
     * 根据客户id查询客户
     */
    void findCustomerById(Integer id);

}
```

CustomerServiceImpl

```
package cn.guardwhy.service.impl;

import cn.guardwhy.service.CustomerService;

/**
 * 客户service实现类
 */
public class CustomerServiceImpl implements CustomerService {

    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

    @Override
    public void findCustomerById(Integer id) {
        // 根据客户id查询客户
        System.out.println("根据客户id查询客户,客户id: " + id);
    }

}
```

LogAdvice

```
package cn.guardwhy.advice;

/**
 * 日志通知
 */
public class LogAdvice {

    /**
     * 记录用户操作日志
     */
    public void printLog(){
        System.out.println("记录用户操作日志");
    }

}
```

bean.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">

    <!--配置客户service-->
    <bean id="customerService"
class="cn.guardwhy.service.impl.CustomerServiceImpl"></bean>

    <!--配置日志通知-->
    <bean id="logAdvice" class="cn.guardwhy.advice.LogAdvice"></bean>

    <!--配置aop: 四个步骤-->
    <aop:config>
        <aop:aspect id="logAspect" ref="logAdvice">
            <aop:before method="printLog" pointcut-ref="pt1"></aop:before>

            <!--切入点表达式演化
            表达式组成:
                访问修饰符 返回值 包名称 类名称 方法名称 (参数列表)
            演化过程:
                全匹配模式:
                    public void
com.itheima.service.impl.CustomerServiceImpl.saveCustomer()
                访问修饰符可以省略:
                    void
com.itheima.service.impl.CustomerServiceImpl.saveCustomer()
                返回值可以使用通配符: *
                    *
com.itheima.service.impl.CustomerServiceImpl.saveCustomer()
                包名称可以使用通配符: * (有多少个包, 就需要多少个*)
                    * *. *. *. *. *. CustomerServiceImpl.saveCustomer()
                类名称可以使用通配符: *
                    * *. *. *. *. *. saveCustomer()
                方法名称可以使用通配符: *
                    * *. *. *. *. *. *()
                参数列表可以使用通配符: * (此时必须要有参数)
                    * *. *. *. *. *. *(*)
                参数列表可以使用: .. (有无参数都可以)
                    * *. *. *. *. *. *(..)
            -->
            <aop:pointcut id="pt1" expression="execution(*
cn.guardwhy.service..*.*(..))"></aop:pointcut>
        </aop:aspect>
    </aop:config>
</beans>
```

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
```

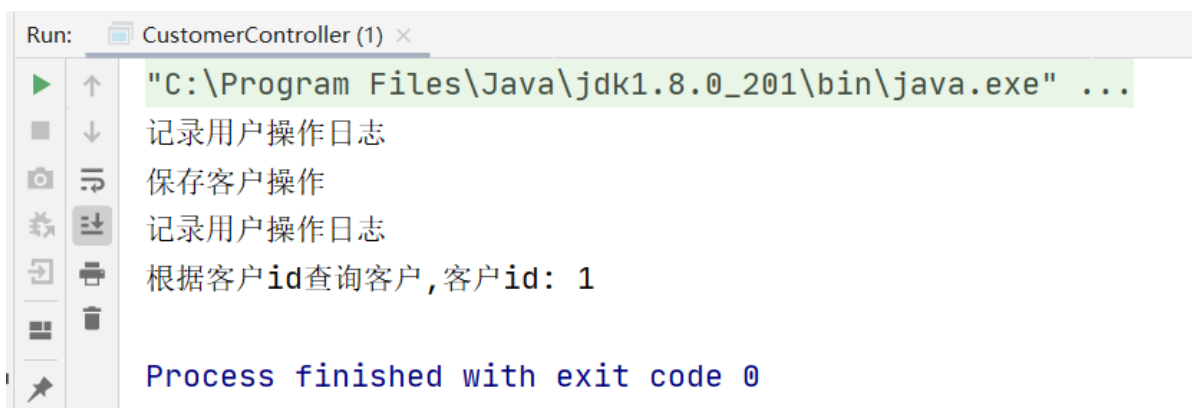
```

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

/**
 * 客户表现层
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.获取客户service
        CustomerService customerService = (CustomerService)
context.getBean("customerService");
        // 3.保存客户
        customerService.saveCustomer();
        // 4.根据客户id查询客户
        customerService.findCustomerById(1);
    }
}

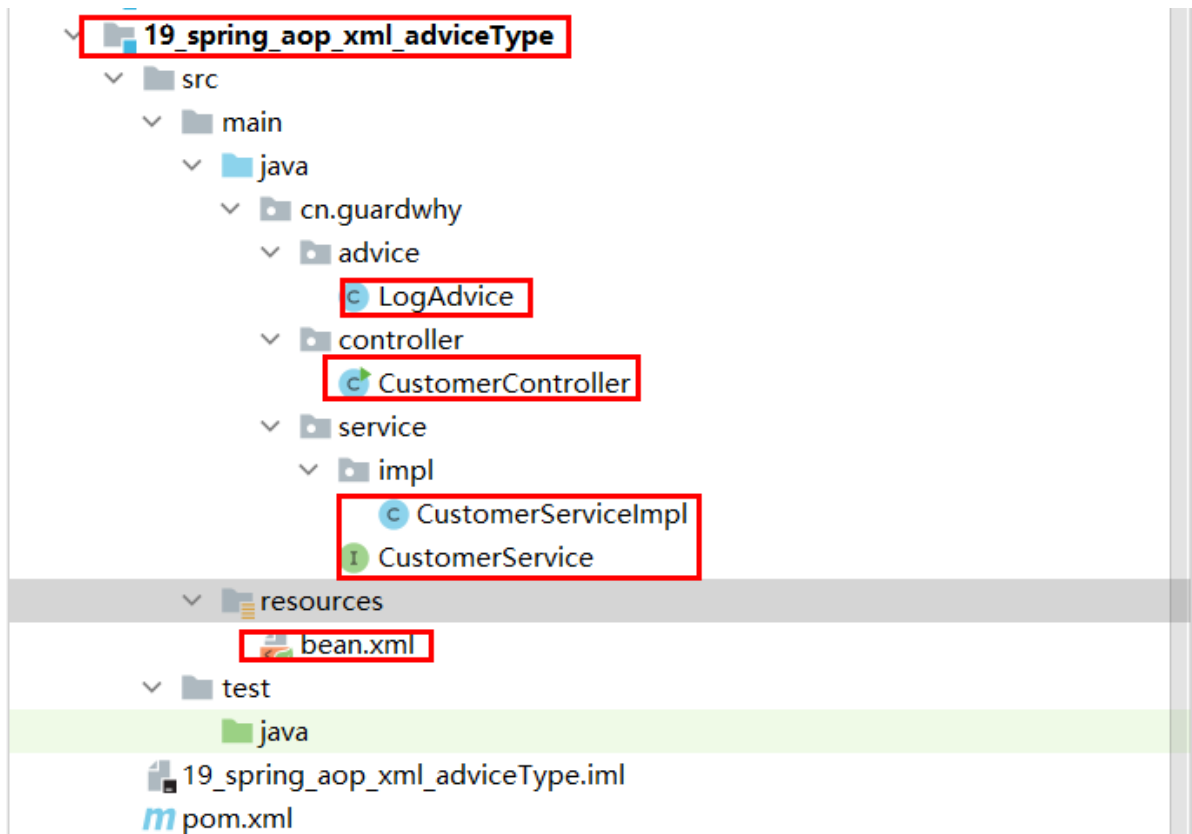
```

执行结果



13.4 通知类型

13.4.1 项目目录



13.4.2 常见通知基本概念

#<aop:before>

作用：配置前置通知,在目标方法执行前执行

属性：

method: 指定通知方法名称
pointcut: 定义切入点表达式
pointcut-ref: 引用切入点表达式的id

#<aop:after-returning>

作用：配置后置通知,无论目标方法正常返回，还是发生异常都会执行。

属性：

method: 指定通知方法名称
pointcut: 定义切入点表达式
pointcut-ref: 引用切入点表达式的id

#<aop:after-throwing>

作用：配置异常通知,在目标方法发生异常后执行。它和后置通知只能执行一个。

属性：

method: 指定通知方法名称
pointcut: 定义切入点表达式
pointcut-ref: 引用切入点表达式的id

#<aop:after>

作用：配置最终通知,在目标方法正常返回后执行。它和异常通知只能执行一个。

属性：

method: 指定通知方法名称。

pointcut: 定义切入点表达式。

pointcut-ref: 引用切入点表达式的id。

#<aop:around>

作用：配置环绕通知,综合了前面四类通知，可以手动控制通知的执行时间点和顺序

属性：

method: 指定通知方法名称

pointcut: 定义切入点表达式

pointcut-ref: 引用切入点表达式的id

13.4.3 代码示例

LogAdvice

```
package cn.guardwhy.advice;

import org.aspectj.lang.ProceedingJoinPoint;

/**
 * 日志通知
 */
public class LogAdvice {
    /**
     * 前置通知
     */
    public void beforeLog(){
        System.out.println("【前置通知】记录用户操作日志");
    }

    /**
     * 后置通知
     */
    public void afterReturningLog(){
        System.out.println("【后置通知】记录用户操作日志");
    }

    /**
     * 异常通知
     */
    public void afterThrowingLog(){
        System.out.println("【异常通知】记录用户操作日志");
    }

    /**
     * 最终通知
     */
    public void afterLog(){
        System.out.println("【最终通知】记录用户操作日志");
    }

    /**
     * 环绕通知:

```

```

*      1.它是spring框架为我们提供了手动控制通知执行时间点和顺序的一种特殊通知类型
*  原理分析:
*      2.spring框架提供了ProceedingJoinPoint接口,作为环绕通知的参数。在环绕通知
*  执行的时候,spring框架会提供实例化对象,我们直接使用即可。该接口中提供了
*  两个方法:
*      getArgs: 获取参数列表
*      proceed: 相当于反射中的invoke方法
*
*/
public void aroundLog(ProceedingJoinPoint pjp){
    // 前置通知
    System.out.println("[环绕通知-前置通知]记录用户操作日志");

    try {
        // 获取参数列表
        Object[] args = pjp.getArgs();
        // 反射调用目标方法
        Object retv = pjp.proceed(args);
        // 后置通知
        System.out.println("[环绕通知-后置通知]记录用户操作日志");
    } catch (Throwable throwable) {
        throwable.printStackTrace();
        // 异常通知
        System.out.println("[环绕通知-异常通知]记录用户操作日志");
    }
    // 最终通知
    System.out.println("[环绕通知-最终通知]记录用户操作日志");
}
}

```

配置bean.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">

    <!--配置客户service-->
    <bean id="customerService"
        class="cn.guardwhy.service.impl.CustomerServiceImpl"/>

    <!--配置日志通知-->
    <bean id="logAdvice" class="cn.guardwhy.advice.LogAdvice"/>

    <!--配置aop -->
    <aop:config>
        <aop:aspect id="logAspect" ref="logAdvice">
            <!--前置通知-->
            <aop:before method="beforeLog" pointcut-ref="pt1"/>
            <!--后置通知-->
            <aop:after-returning method="afterReturningLog" pointcut-ref="pt1"/>
            <!--异常通知-->
            <aop:after-throwing method="afterThrowingLog" pointcut-ref="pt1"/>
            <!--最终通知-->

```



```

        <aop:after method="afterLog" pointcut-ref="pt1"/>

        <!--环绕通知-->
        <aop:around method="aroundLog" pointcut-ref="pt1"/>
        <!--切入点表达式-->
        <aop:pointcut id="pt1" expression="execution(*
cn.guardwhy.service..*.*(..))"/>
    </aop:aspect>
</aop:config>
</beans>

```

CustomerController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户表现层
 */
public class CustomerController {
    public static void main(String[] args) {
        // 1.加载spring配置文件,创建spring容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:bean.xml");
        // 2.获取客户service
        CustomerService customerService = (CustomerService)
context.getBean("customerService");
        // 3.保存客户
        customerService.saveCustomer();
    }
}

```

执行结果

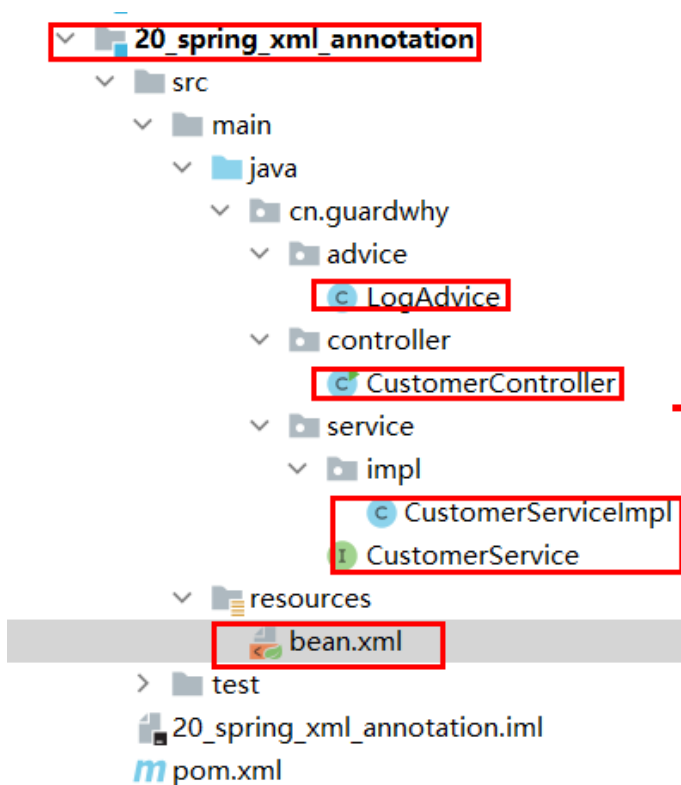
```

[前置通知]记录用户操作日志
[环绕通知-前置通知]记录用户操作日志
保存客户操作
[环绕通知-后置通知]记录用户操作日志
[环绕通知-最终通知]记录用户操作日志
【最终通知】记录用户操作日志
【后置通知】记录用户操作日志

```

13.5 注解配置AOP

13.5.1 项目目录



13.5.1 代码实现

CustomerService

```
package cn.guardwhy.service;

/**
 * 客户service接口
 */
public interface CustomerService {
    /**
     * 保存客户
     */
    void saveCustomer();

    /**
     * 根据客户id查询客户
     */
    void findCustomerById(Integer id);
}
```

CustomerServiceImpl

```
package cn.guardwhy.service.impl;

import cn.guardwhy.service.CustomerService;
import org.springframework.stereotype.Service;
```

```

/**
 * 客户service实现类
 */
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
    @Override
    public void saveCustomer() {
        System.out.println("保存客户操作");
    }

    @Override
    public void findCustomerById(Integer id) {
        // 根据客户id查询客户
        System.out.println("根据客户id查询客户,客户id: " + id);
    }
}

```

LogAdvice

```

package cn.guardwhy.advice;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

/**
 * 日志通知类
 * 注解说明:
 *      @Aspect:
 *          作用: 声明当前类是一个切面类, 相当于xml中aop:aspect标签
 *      @Before:
 *          作用: 配置前置通知
 *      @AfterReturning:
 *          作用: 配置后置通知
 *      @AfterThrowing:
 *          作用: 配置异常通知
 *      @After:
 *          作用: 配置最终通知
 *      @Pointcut:
 *          作用: 配置切入点表达式
 */
@Component("logAdvice")
@Aspect
public class LogAdvice {

    /**
     * 切入点表达式
     */
    @Pointcut("execution(* cn.guardwhy.service..*.*(..))")
    public void pt1(){}

    /**
     * 前置通知
     */
    @Before("pt1()")
    public void beforeLog(){

```

```

        System.out.println("[前置通知]记录用户操作日志");
    }

    /**
     * 后置通知
     */
    @AfterReturning("pt1()")
    public void afterReturningLog(){
        System.out.println("【后置通知】记录用户操作日志");
    }

    /**
     * 异常通知
     */
    @AfterThrowing("pt1()")
    public void afterThrowingLog(){
        System.out.println("【异常通知】记录用户操作日志");
    }

    /**
     * 最终通知
     */
    @After("pt1()")
    public void afterLog(){
        System.out.println("【最终通知】记录用户操作日志");
    }

    /**
     * 环绕通知:
     *     1.它是spring框架为我们提供了手动控制通知执行时间点和顺序的一种特殊通知类型
     * 原理分析:
     *     2.spring框架提供了ProceedingJoinPoint接口,作为环绕通知的参数。在环绕通知
     *     执行的时候,spring框架会提供实例化对象,我们直接使用即可。该接口中提供了
     *     两个方法:
     *         getArgs: 获取参数列表
     *         proceed: 相当于反射中的invoke方法
     */
    @Around("pt1()")
    public void aroundLog(ProceedingJoinPoint pjp){
        // 前置通知
        System.out.println("[环绕通知-前置通知]记录用户操作日志");

        try {
            // 获取参数列表
            Object[] args = pjp.getArgs();
            // 反射调用目标方法
            Object retv = pjp.proceed(args);
            // 后置通知
            System.out.println("[环绕通知-后置通知]记录用户操作日志");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
            // 异常通知
            System.out.println("[环绕通知-异常通知]记录用户操作日志");
        }
        // 最终通知
        System.out.println("[环绕通知-最终通知]记录用户操作日志");
    }
}

```

```
}
```

13.5.4 配置bean.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"
       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/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!--配置包扫描advice/service，说明：
        第一步：导入context名称空间和约束
        第二步：通过<context:component-scan>标签配置包扫描
    -->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--关键步骤：开启spring对注解aop支持-->
    <aop:aspectj-autoproxy/>
</beans>
```

CustomerController

```
package cn.guardwhy.controller;

import cn.guardwhy.service.CustomerService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 客户表现层
 */
public class CustomerController {

    public static void main(String[] args) {
        // 1.加载spring的配置文件，创建spring的ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");

        // 2.获取客户service
        CustomerService customerService =
        (CustomerService)context.getBean("customerService");

        // 3.保存客户
        customerService.saveCustomer();

        // 根据客户id查询客户
        // customerService.findCustomerById(1);
    }
}
```

```
[环绕通知-前置通知]记录用户操作日志  
[前置通知]记录用户操作日志
```

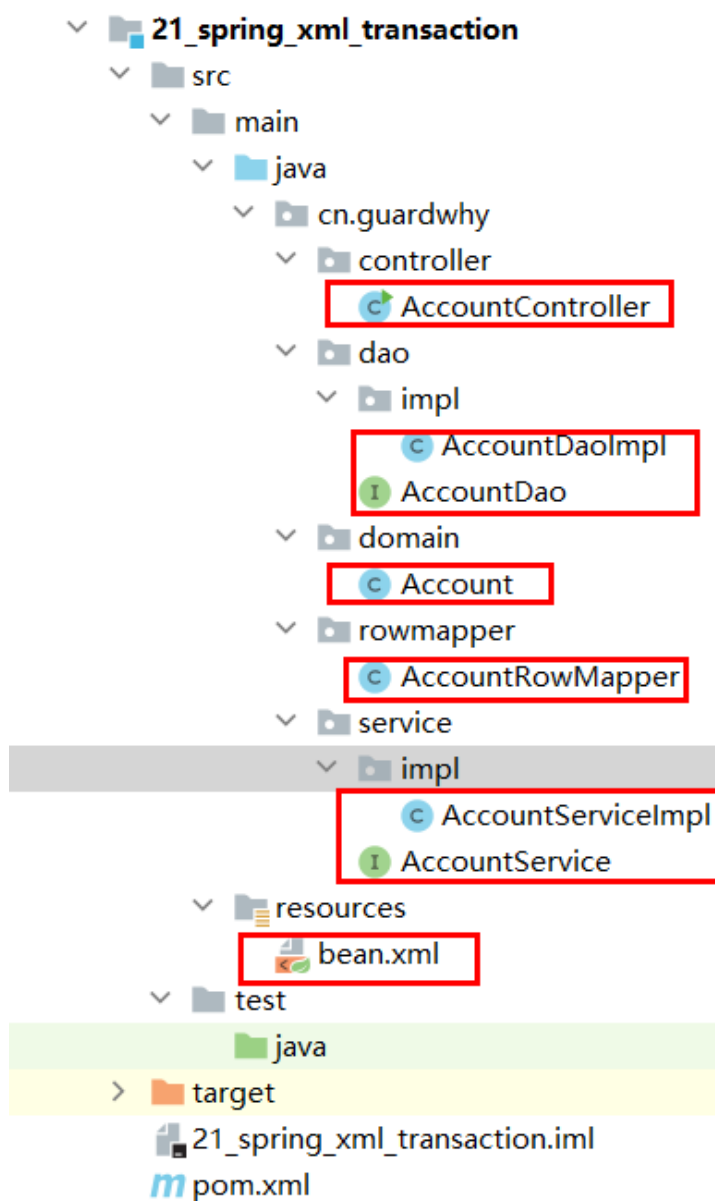
```
保存客户操作
```

```
【后置通知】记录用户操作日志  
【最终通知】记录用户操作日志  
[环绕通知-后置通知]记录用户操作日志  
[环绕通知-最终通知]记录用户操作日志
```

```
Process finished with exit code 0
```

15-声明式事务(xml)

15.1 项目目录



15.2 spring事务管理

事务定义

事务(transaction)一般是指要做的或者所做的事情。在程序中，尤其是在操作数据库的程序中，指的是访问并且可能更新数据库中数据项的一个执行单元。

这个执行单元由事务开始（begin transaction）和事务结束（end transaction）之间执行的全部操作组成。

事务ACID原则

事务具有4个基本特性：原子性、一致性、隔离性、持久性。也就是我们常说的ACID原则

原子性 (Atomicity)

- 一个事务已经是一个不可再分割的工作单位。事务中的全部操作要么都做；要么都不做

一致性 (Consistency)

- 事务必须是使得数据库状态从一个一致性状态，转变到另外一个一致性状态。也就是说在事务前，和事务后，被操作的目标资源状态一致。
比如银行转账案例中，转账前和转账后，总账不变。

隔离性 (Isolation)

- 一个事务的执行不能被其他事务的影响。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，多个并发事务之间不能相互干扰。

持久性 (Durability)

- 一个事务一旦提交，它对数据库中数据的改变会永久存储起来其他操作不会对它产生影响。

15.3 代码示例

数据表

```
-- 创建账户表
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money float
)ENGINE=InnoDB character set utf8 collate utf8_general_ci;

-- 初始化新增三个账户
insert into account(name,money) values('kebe',1000);
insert into account(name,money) values('curry',1500);
insert into account(name,money) values('james',2000);

select * from account;
```

编写实体类

```
package cn.guardwhy.domain;

/**
 * 账户实体类
 */
@Data
```

```

@NoArgsConstructor
@AllArgsConstructor
public class Account {
    // 成员变量
    private int id;
    private String name;
    private Float money;
}

```

结果集映射类

```

package cn.guardwhy.rowmapper;

import cn.guardwhy.domain.Account;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * 账户结果集映射类
 */
public class AccountRowMapper implements RowMapper<Account>{

    /**
     * 每一行记录，调用一次该方法
     */
    public Account mapRow(ResultSet rs, int index) throws SQLException {

        // 创建Account
        Account account = new Account();

        account.setId(rs.getInt("id"));
        account.setName(rs.getString("name"));
        account.setMoney(rs.getFloat("money"));

        return account;
    }
}

```

持久层(Dao)

dao接口

```

package cn.guardwhy.dao;

import cn.guardwhy.domain.Account;

/**
 * 账户dao接口
 */
public interface AccountDao {

    /**
     * 根据账户id查询账户
     */
    Account findAccountById(Integer accountId);
}

```



```

/**
 * 根据账户名称查询账户
 */
Account findAccountByName(String accountName);

/**
 * 更新账户
 */
void updateAccount(Account account);
}

```

AccountDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.AccountDao;
import cn.guardwhy.domain.Account;
import cn.guardwhy.rowmapper.AccountRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import java.util.List;

/**
 * 账户dao实现类
 */
public class AccountDaoImpl implements AccountDao{

    // 定义JdbcTemplate
    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    /**
     * 根据账户id查询账户
     *
     * @param accountId
     */
    public Account findAccountById(Integer accountId) {
        List<Account> list = jdbcTemplate.query("select * from account where id=?", new AccountRowMapper(), accountId);

        return list.isEmpty()?null:list.get(0);
    }

    /**
     * 根据账户名称查询账户
     *
     * @param accountName
     */
    public Account findAccountByName(String accountName) {
        List<Account> list = jdbcTemplate.query("select * from account where name=?", new AccountRowMapper(), accountName);

        // 判断不为空

```

```

        if(list.isEmpty()){
            return null;
        }

        // 判断账户是否唯一
        if(list.size()>1){
            throw new RuntimeException("账户不唯一，请检查!");
        }

        return list.get(0);
    }

    /**
     * 更新账户
     *
     * @param account
     */
    public void updateAccount(Account account) {
        jdbcTemplate.update("update account set money=? where
id=?",account.getMoney(),account.getId());
    }
}

```

业务层(Service)

service接口

```

package cn.guardwhy.service;

import cn.guardwhy.domain.Account;

/**
 * 账户service接口
 */
public interface AccountService {

    /**
     * 根据账户id查询账户
     */
    Account findAccountById(Integer accountId);

    /**
     * 转账操作：
     * 参数说明：
     *     sourceName: 转出账户
     *     destName: 转入账户
     *     money: 转账金额
     */
    void transfer(String sourceName,String destName,Float money);
}

```

AccountServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.AccountDao;

```

```

import cn.guardwhy.domain.Account;
import cn.guardwhy.service.AccountService;

/**
 * 账户service实现类
 */
public class AccountServiceImpl implements AccountService{

    // 定义账户dao
    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    /**
     * 根据账户id查询账户
     *
     * @param accountId
     */
    public Account findAccountById(Integer accountId) {
        return accountDao.findAccountById(accountId);
    }

    /**
     * 转账操作：
     * 参数说明：
     * sourceName: 转出账户
     * destName: 转入账户
     * money: 转账金额
     *
     * @param sourceName
     * @param destName
     * @param money
     */
    public void transfer(String sourceName, String destName, Float money) {

        // 1.查询转出账户
        Account sourceAccount = accountDao.findAccountByName(sourceName);

        // 2.查询转入账户
        Account destAccount = accountDao.findAccountByName(destName);

        // 3.转出账户-money
        sourceAccount.setMoney(sourceAccount.getMoney()-money);

        // 4.转入账户+money
        destAccount.setMoney(destAccount.getMoney()+money);

        // 5.更新转出账户
        accountDao.updateAccount(sourceAccount);

        // 6.更新转入账户
        accountDao.updateAccount(destAccount);

    }
}

```

15.4 声明式事务配置

配置事务管理器

```
<!--第一步：配置事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源对象-->
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

配置aop和切入点表达式

```
<!--第二步：配置aop与切入点表达式-->
<aop:config>
    <aop:pointcut id="pt1" expression="execution(* cn.guardwhy.service...*(..))"> </aop:pointcut>
</aop:config>
```

建立通知与切入点表达式关系

```
<!--第二步：配置aop与切入点表达式-->
<aop:config>
    <aop:pointcut id="pt1" expression="execution(* cn.guardwhy.service...*(..))"></aop:pointcut>

    <!--第三步：建立事务管理器（通知）与切入点表达式关系-->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
</aop:config>
```

配置通知

```
<!--第四步：配置通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">

</tx:advice>
```

配置事务属性

```
<!--第四步：配置通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--第五步：配置事务属性-->
    <tx:attributes>
        <!--tx:method标签：配置业务方法的名称规则，说明：
        name：方法名称规则，可以使用通配符*
        isolation：事务隔离级别，使用默认值即可
        propagation：事务传播行为，增/删/改方法使用REQUIRED。查询方法使用SUPPORTS。
        read-only：是否只读事务。增/删/改方法使用false。查询方法使用true。
        timeout：配置事务超时。使用默认值-1即可。永不超时。
        rollback-for：发生某种异常时，回滚；发生其它异常不回滚。没有默认值，任何异常都回滚。
        no-rollback-for：发生某种异常时不回滚；发生其它异常时回滚。没有默认值，任何异常都回滚。
        -->
```

```

        <tx:method name="transfer*" propagation="REQUIRED" read-only="false"/>
    </tx:attributes>
</tx:advice>

```

配置bean.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"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!--配置账户service-->
    <bean id="accountService"
class="cn.guardwhy.service.impl.AccountServiceImpl">
        <!--注入账户dao-->
        <property name="accountDao" ref="accountDao"/>
    </bean>

    <!--配置账户dao-->
    <bean id="accountDao" class="cn.guardwhy.dao.impl.AccountDaoImpl">
        <!--注入JdbcTemplate-->
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置数据源对象-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!--注入连接数据库的四个基本要素-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>

        <!--注入连接池公共属性-->
        <!-- 初始化连接数量 -->
        <property name="initialSize" value="6" />
        <!-- 最小空闲连接数 -->
        <property name="minIdle" value="3" />
        <!-- 最大并发连接数(最大连接池数量) -->
        <property name="maxActive" value="50" />
        <!-- 配置获取连接等待超时的时间 -->
        <property name="maxWait" value="60000" />
        <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
    </bean>

```

```

<!--配置声明式事务-->
<!--第一步：配置事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源对象-->
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!--第二步：配置aop和切入点表达式-->
<aop:config>
    <aop:pointcut id="pt1" expression="execution(* cn.guardwhy.service...*(..))"></aop:pointcut>

    <!--第三步：建立通知和切入点表达式的关系-->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
</aop:config>

<!--第四步：配置通知-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--第五步：配置事务属性-->
    <tx:attributes>
        <tx:method name="transfer*" propagation="REQUIRED" read-
only="false" />
    </tx:attributes>
</tx:advice>

</beans>

```

表现层(Controller)

AccountController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.AccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 账户表现层
 */
public class AccountController {

    public static void main(String[] args) {
        // 1.加载spring配置文件，创建spring ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");

        // 2.获取账户service
        AccountService accountService =
        (AccountService)context.getBean("accountService");

        // 3.转账操作
        accountService.transfer("curry","james",200f);
    }
}

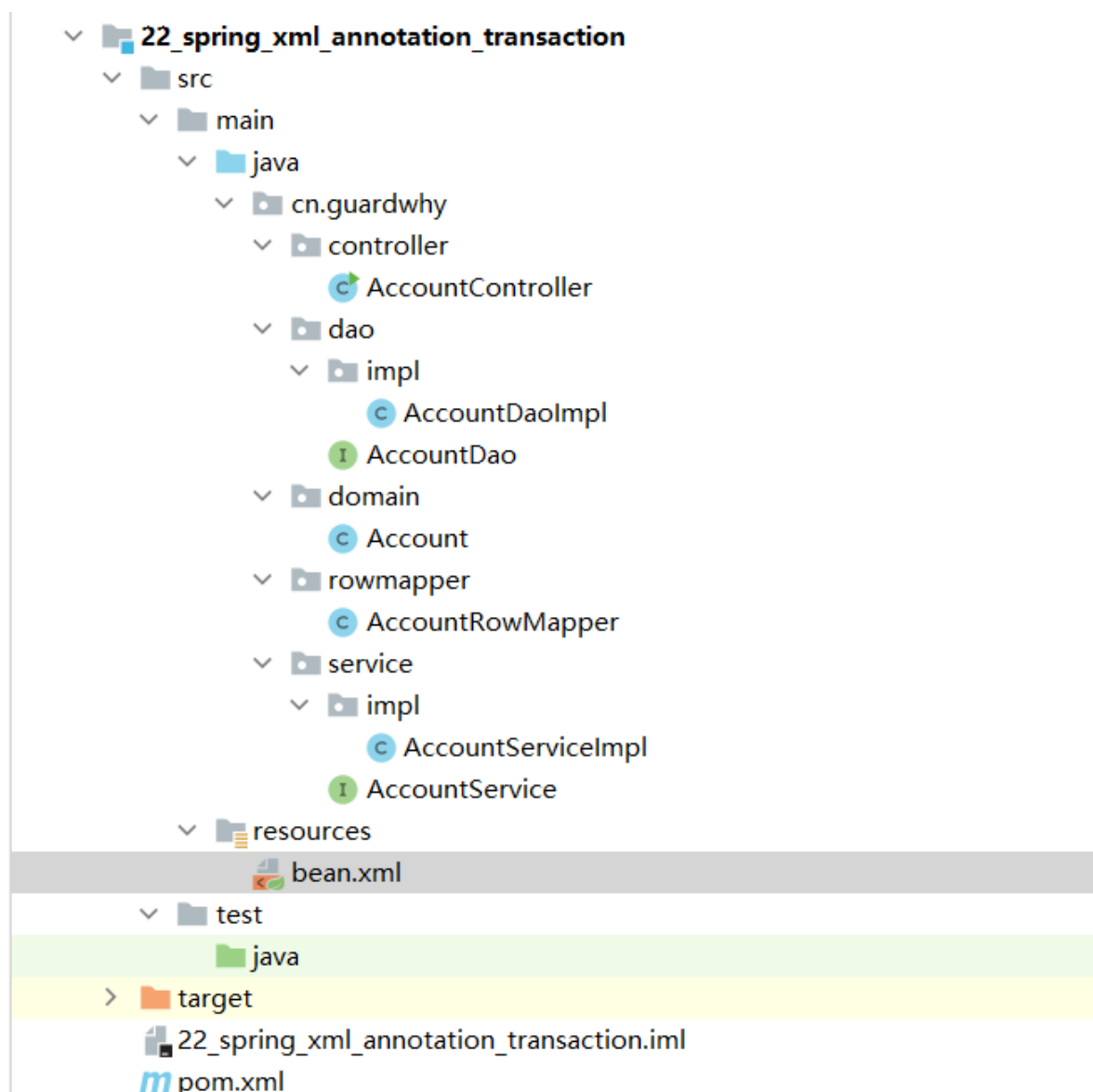
```

15.5 执行结果

	id	name	money		id	name	money
1	1	curry	1500		1	curry	1300
2	2	james	2000	执行前	2	james	2200
3	3	小王	2200		3	小王	2200
4	4	候大利	3500		4	候大利	3500

16-声明式事务(注解xml)

16.1项目目录



16.2 代码实现

持久层(Dao)

AccountDao

```
package cn.guardwhy.dao;
```

```

import cn.guardwhy.domain.Account;

/**
 * 账户dao接口
 */
public interface AccountDao {

    /**
     * 根据账户id查询账户
     */
    Account findAccountById(Integer accountId);

    /**
     * 根据账户名称查询账户
     */
    Account findAccountByName(String accountName);

    /**
     * 更新账户
     */
    void updateAccount(Account account);
}

```

AccountDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.AccountDao;
import cn.guardwhy.domain.Account;
import cn.guardwhy.rowmapper.AccountRowMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

/**
 * 账户dao实现类
 */
@Repository("accountDao")
public class AccountDaoImpl implements AccountDao{

    // 定义JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    /**
     * 根据账户id查询账户
     *
     * @param accountId
     */
}

```



```

    public Account findAccountById(Integer accountId) {
        List<Account> list = jdbcTemplate.query("select * from account where
id=?", new AccountRowMapper(), accountId);

        return list.isEmpty()?null:list.get(0);
    }

    /**
     * 根据账户名称查询账户
     *
     * @param accountName
     */
    public Account findAccountByName(String accountName) {
        List<Account> list = jdbcTemplate.query("select * from account where
name=?", new AccountRowMapper(), accountName);

        // 判断不为空
        if(list.isEmpty()){
            return null;
        }

        // 判断账户是否唯一
        if(list.size()>1){
            throw new RuntimeException("账户不唯一，请检查!");
        }

        return list.get(0);
    }

    /**
     * 更新账户
     *
     * @param account
     */
    public void updateAccount(Account account) {
        jdbcTemplate.update("update account set money=? where
id=?",account.getMoney(),account.getId());
    }
}

```

业务层(Service)

AccountService

```

package cn.guardwhy.service;

import cn.guardwhy.domain.Account;

/**
 * 账户service接口
 */
public interface AccountService {

    /**
     * 根据账户id查询账户
     */
    Account findAccountById(Integer accountId);
}

```

```

/**
 * 转账操作：
 * 参数说明：
 *     sourceName: 转出账户
 *     destName: 转入账户
 *     money: 转账金额
 */
void transfer(String sourceName,String destName,Float money);
}

```

AccountServiceImpl

```

package cn.guardwhy.service.impl;

import cn.guardwhy.dao.AccountDao;
import cn.guardwhy.domain.Account;
import cn.guardwhy.service.AccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * 账户service实现类
 */
@Service("accountService")
public class AccountServiceImpl implements AccountService{

    // 定义账户dao
    @Autowired
    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    /**
     * 根据账户id查询账户
     *
     * @param accountId
     */
    public Account findAccountById(Integer accountId) {
        return accountDao.findAccountById(accountId);
    }

    /**
     * 转账操作：
     * 参数说明：
     * sourceName: 转出账户
     * destName: 转入账户
     * money: 转账金额
     * @param sourceName
     * @param destName
     * @param money
     */
    public void transfer(String sourceName, String destName, Float money) {

```

```

// 1.查询转出账户
Account sourceAccount = accountDao.findAccountByName(sourceName);

// 2.查询转入账户
Account destAccount = accountDao.findAccountByName(destName);

// 3.转出账户-money
sourceAccount.setMoney(sourceAccount.getMoney()-money);

// 4.转入账户+money
destAccount.setMoney(destAccount.getMoney()+money);

// 5.更新转出账户
accountDao.updateAccount(sourceAccount);

// 6.更新转入账户
accountDao.updateAccount(destAccount);

}
}

```

配置bean.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:tx="http://www.springframework.org/schema/tx"
    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/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!--配置包扫描dao/service-->
    <context:component-scan base-package="cn.guardwhy"/>

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入数据源对象-->
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置数据源对象-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!--注入连接数据库的四个基本要素-->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/spring"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>

        <!--数据库连接池常用属性-->
        <!-- 初始化连接数量 -->
        <property name="initialSize" value="6" />
        <!-- 最小空闲连接数 -->
        <property name="minIdle" value="3" />
    </bean>

```

```

<!-- 最大并发连接数(最大连接池数量) -->
<property name="maxActive" value="50" />
<!-- 配置获取连接等待超时的时间 -->
<property name="maxWait" value="60000" />
<!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
<property name="timeBetweenEvictionRunsMillis" value="60000" />
</bean>

<!--spring事务管理配置步骤-->
<!--第一步：配置事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--注入数据源对象-->
    <property name="dataSource" ref="dataSource"/>
</bean>
<!--第二步：开启spring对注解事务的支持 -->
<tx:annotation-driven/>
</beans>

```

表现层(Controller)

AccountController

```

package cn.guardwhy.controller;

import cn.guardwhy.service.AccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 账户表现层
 */
public class AccountController {

    public static void main(String[] args) {
        // 1.加载spring配置文件，创建spring ioc容器
        ApplicationContext context = new
        ClassPathXmlApplicationContext("classpath:bean.xml");

        // 2.获取账户service
        AccountService accountService =
        (AccountService)context.getBean("accountService");

        // 3.转账操作
        accountService.transfer("curry","james",200f);
    }
}

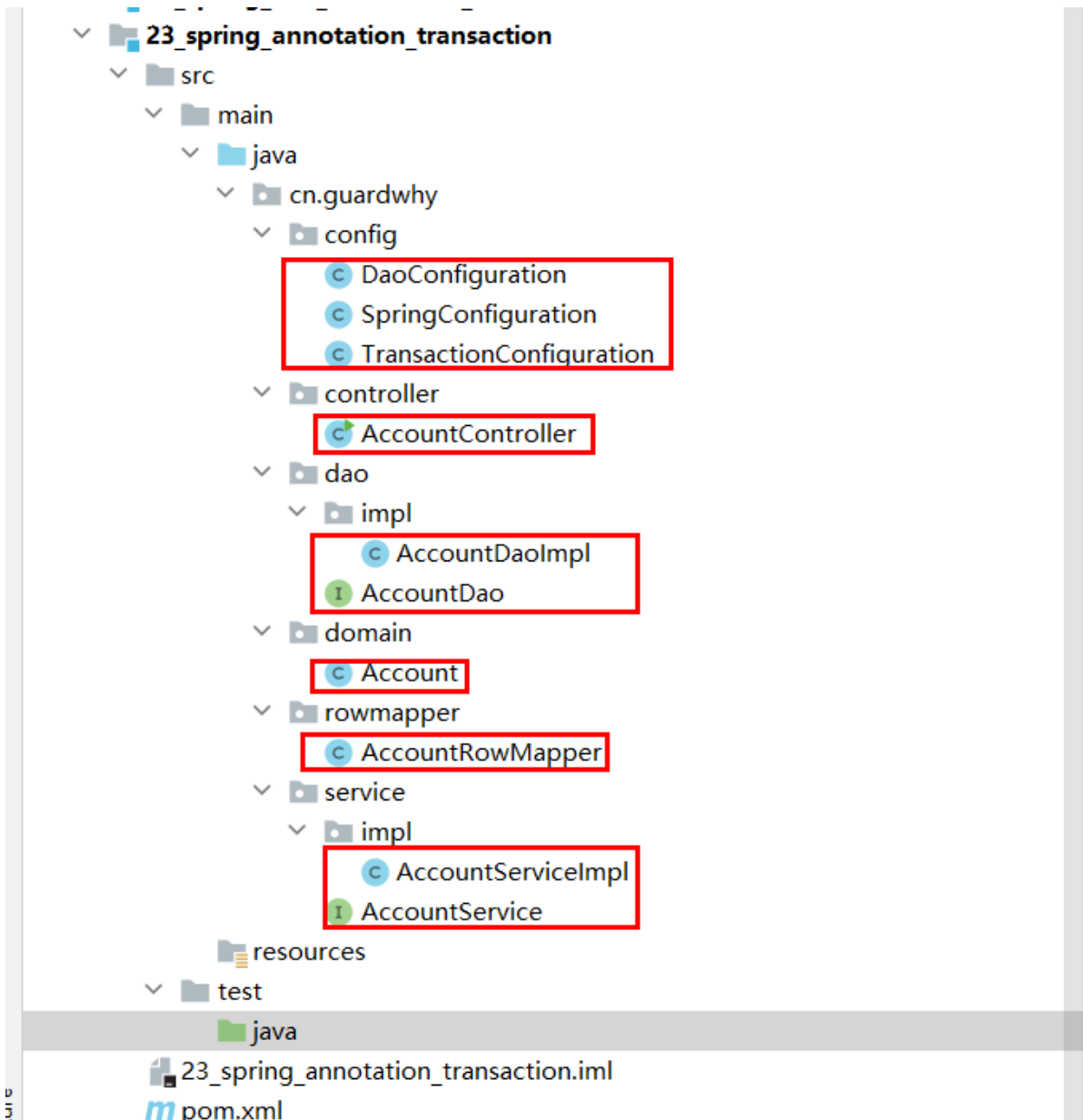
```

16.3 执行结果

	id	name	money		id	name	money
1	1	curry	1100	1	1	curry	900
2	2	james	2400	2	2	james	2600
3	3	小王	2200	3	3	小王	2200
4	4	候大利	3500	4	4	候大利	3500

17- 声明式事务(注解)

17.1 项目目录



17.2 持久层配置类

```
package cn.guardwhy.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.JdbcTemplate;
```

```

import javax.sql.DataSource;

/**
 * 持久层配置类
 */
public class DaoConfiguration {
    /**
     * 创建JdbcTemplate对象
     */
    @Bean("jdbcTemplate")
    public JdbcTemplate getJdbcTemplate(DataSource dataSource){
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    /**
     * 创建数据源对象
     */
    @Bean("dataSource")
    public DataSource createDataSource(){
        // 创建DataSource
        DruidDataSource dataSource = new DruidDataSource();

        // 注入属性
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/spring");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        dataSource.setInitialSize(6);
        dataSource.setMinIdle(3);
        dataSource.setMaxActive(50);
        dataSource.setMaxWait(60000);
        dataSource.setTimeBetweenEvictionRunsMillis(60000);
        return dataSource;
    }
}

```

17.3 事务配置类

```

package cn.guardwhy.config;

import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import javax.sql.DataSource;

/**
 * 事务配置类
 */
public class TransactionConfiguration {
    @Bean("transactionManager")
    public DataSourceTransactionManager getTransactionManager(DataSource
dataSource){

        // 创建事务管理器对象
    }
}

```

```

        DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();

        // 设置数据源
        transactionManager.setDataSource(dataSource);

        return transactionManager;
    }
}

```

17.4 主配置类

```

package cn.guardwhy.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.transaction.annotation.EnableTransactionManagement;

/**
 * 主配置类步骤:
 *      1.使用@Configuration注解, 声明主配置类
 *      2.使用@ComponentScan注解, 扫描注解配置
 *      3.使用@Import注解, 导入其它模块配置类
 *      4.使用@EnableTransactionManagement注解, 启用spring对注解事务的配置
 */
@Configuration
@ComponentScan(value="cn.guardwhy")
@Import(value={DaoConfiguration.class,TransactionConfiguration.class})
@EnableTransactionManagement
public class SpringConfiguration {
}

```

17.5 表现层

AccountController

```

package cn.guardwhy.controller;

import cn.guardwhy.config.SpringConfiguration;
import cn.guardwhy.service.AccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * 账户表现层
 */
public class AccountController {

    public static void main(String[] args) {
        // 1.加载spring配置文件, 创建spring ioc容器
        ApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfiguration.class);

        // 2.获取账户service
    }
}

```

```

    AccountService accountService =
        (AccountService)context.getBean("accountService");

    // 3.转账操作
    accountService.transfer("curry","james",200f);
}
}

```

17.6 执行结果

	id	name	money		id	name	money
1	1	curry	900	1	1	curry	700
2	2	james	2600	2	2	james	2800
3	3	小王	2200	3	3	小王	2200
4	4	侯大利	3500	4	4	侯大利	3500

18-Spring 整合Mybatis

MyBatis-Spring学习

引入Spring之前需要了解mybatis-spring包中的一些重要类

官方网站: <http://mybatis.org/spring/zh/index.html>

mybatis-spring


最近更新: 14 十一月 2020 | 版本: 2.0.6

阅读全文

简介

入门

SqlSessionFactoryBean

事务

简介

什么是 MyBatis-Spring?

MyBatis-Spring 会帮助你 MyBatis 代码无缝地整合到 Spring 中。它将允许 MyBatis 参与到 Spring 的事务管理之中，创建映射器 mapper 和 `SqlSession` 并注入到 bean 中，以及将 MyBatis 的异常转换为 Spring 的 `DataAccessException`。最终，可以做到应用代码不依赖于 MyBatis，Spring 或 MyBatis-Spring。

所需条件

在开始使用 MyBatis-Spring 之前，你需要先熟悉 Spring 和 MyBatis 这两个框架和有关它们的术语。这很重要——因为本手册中不会提供二者的基本内容，安装和配置教程。

MyBatis-Spring 需要以下版本：

MyBatis-Spring	MyBatis	Spring Framework	Spring Batch	Java
2.0	3.5+	5.0+	4.0+	Java 8+
1.3	3.4+	3.2.2+	2.1+	Java 6+

如果使用 Maven 作为构建工具，仅需要在 pom.xml 中加入以下代码即可：

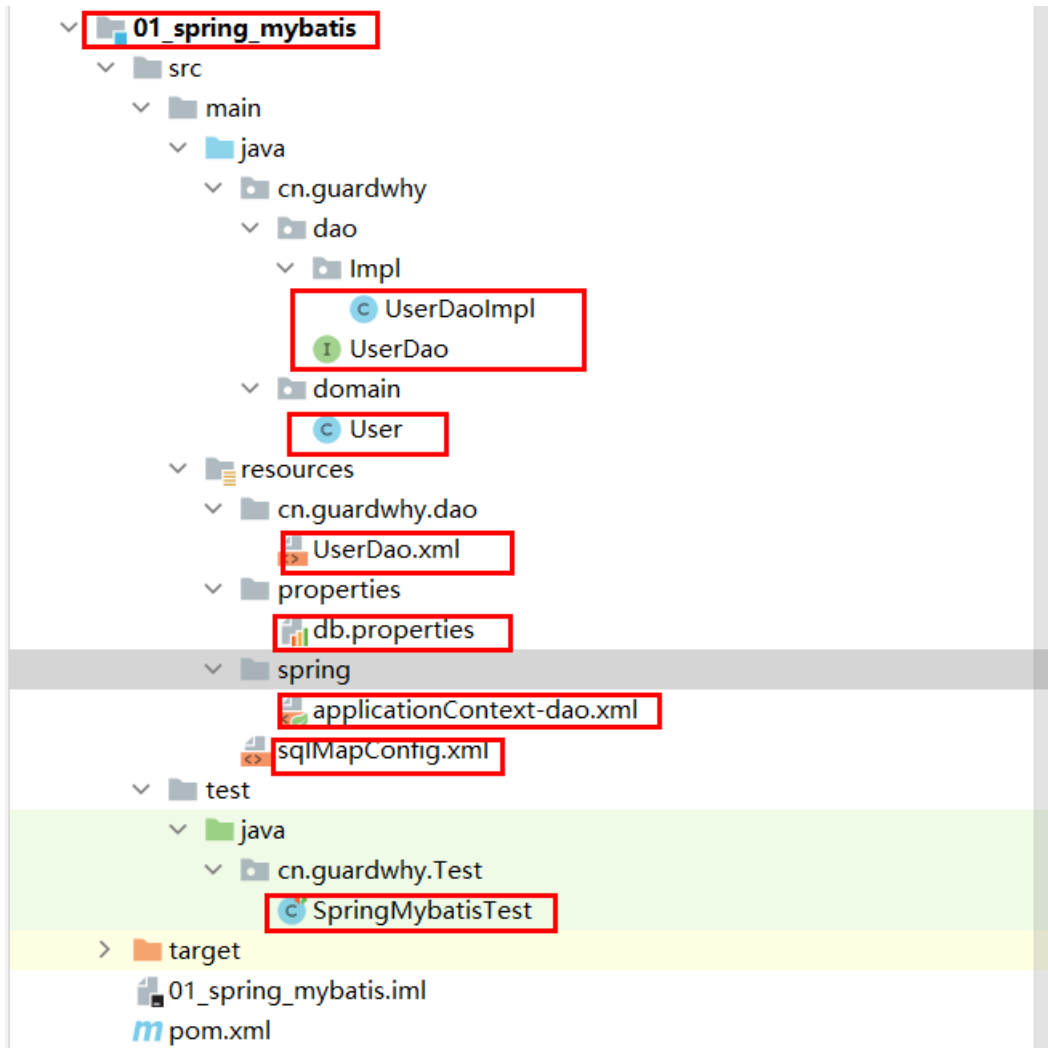
```

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>2.0.6</version>
</dependency>

```


18.1 整合方式一

18.1.1 项目目录



18.1.2 创建数据库

```
-- 创建数据库
create database db_mybatis;

-- 创建数据表
create table user (
  id int primary key auto_increment,
  user_name varchar(20) not null,
  birthday date,
  sex char(1) default '男',
  address varchar(50)
);

-- 插入数据
insert into user values (null, '侯大利', '1980-10-24', '男', '江州');
insert into user values (null, '田甜', '1992-11-12', '女', '扬州');
insert into user values (null, '王永强', '1983-05-20', '男', '扬州');
insert into user values (null, '杨红', '1995-03-22', '女', '秦阳');

select * from user;
```

```
<?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>cn.guardwhy</groupId>
    <artifactId>Spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <!-- spring版本 -->
        <spring.version>5.2.9.RELEASE</spring.version>
        <!-- mysql版本 -->
        <mysql.version>5.1.30</mysql.version>
        <!--druid版本-->
        <druid.version>1.0.29</druid.version>
        <!--junit 版本-->
        <junit.version>4.12</junit.version>
        <!--aopalliance版本-->
        <aopalliance.version>1.0</aopalliance.version>
    </properties>

    <dependencies>
        <!--spring 版本-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- mysql数据库依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>
        <!--spring jdbc依赖-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!--druid依赖-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>${druid.version}</version>
        </dependency>
        <!--spring test依赖-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-Test</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!--junit依赖-->
        <dependency>
```

```

        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
    </dependency>
    <!-- spring aspects依赖 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!--aopalliance依赖-->
    <dependency>
        <groupId>aopalliance</groupId>
        <artifactId>aopalliance</artifactId>
        <version>${aopalliance.version}</version>
    </dependency>
    <!--lombok插件-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.16</version>
    </dependency>
    <!--mybatis-spring整合-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.6</version>
    </dependency>
    <!-- mybatis相关依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>
</dependencies>
<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>/**/*.properties</include>
                <include>/**/*.xml</include>
            </includes>
            <filtering>true</filtering>
        </resource>
    </resources>
</build>
</project>

```

db.properties

```

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

```

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>

    <!--定义实体类别名-->
    <typeAliases>
        <package name="cn.guardwhy.domain"/>
    </typeAliases>
</configuration>
```

18.1.3 代码示例

编写User实体类

```
package cn.guardwhy.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;
}
```

编写持久层

UserDao

```
package cn.guardwhy.dao;

import cn.guardwhy.domain.User;

import java.util.List;

public interface UserDao {
    // 查找所有用户
    List<User> findAllUsers();
}
```

UserDao.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="cn.guardwhy.dao.UserDao">
    <!--结果集映射-->
    <resultMap id="userMap" type="user">
        <id property="id" column="id"/>
        <result property="username" column="user_name"/>
        <result property="birthday" column="birthday"/>
        <result property="sex" column="sex"/>
        <result property="address" column="address"/>
    </resultMap>
    <!-- 1.查询表中所有的用户-->
    <select id="findAllUsers" resultMap="userMap" parameterType="int">
        select * from user
    </select>
</mapper>

```

UserDaoImpl

```

package cn.guardwhy.dao.impl;

import cn.guardwhy.dao.UserDao;
import cn.guardwhy.domain.User;
import org.mybatis.spring.SqlSessionTemplate;

import java.util.List;

public class UserDaoImpl implements UserDao {
    // 注入sqlSession
    private SqlSessionTemplate sqlSession;

    public void setSqlSession(SqlSessionTemplate sqlSession) {
        this.sqlSession = sqlSession;
    }

    @Override
    public List<User> findAllUsers() {
        UserDao mapper = sqlSession.getMapper(UserDao.class);
        return mapper.findAllUsers();
    }
}

```

applicationContext-dao.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">

    <!--1.读取db.properties文件-->
    <context:property-placeholder
location="classpath:properties/db.properties"/>

```

```

<!--2.创建数据源-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<!--3.创建SqlSessionFactory（使用SqlSessionFactoryBean）-->
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <!--注入数据源-->
    <property name="dataSource" ref="dataSource"/>
    <!--绑定Mybatis配置文件-->
    <property name="configLocation" value="classpath:sqlMapConfig.xml"/>
    <property name="mapperLocations"
value="classpath:cn/guardwhy/dao/UserDao.xml"/>
</bean>

<!--4.SqlSessionTemplate: 就是使用的sqlSession-->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <!--构造器注入,没有set方法-->
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>

<!--配置dao-->
<bean id="userDao" class="cn.guardwhy.dao.impl.UserDaoImpl">
    <property name="sqlSession" ref="sqlSession"/>
</bean>
</beans>

```

测试代码

```

package cn.guardwhy.Test;

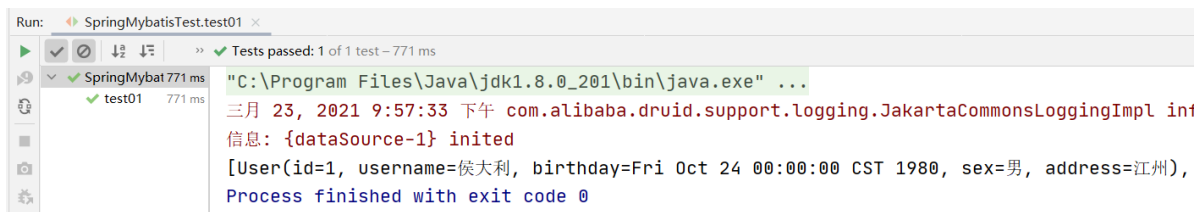
import cn.guardwhy.dao.UserDao;
import cn.guardwhy.domain.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

public class SpringMybatisTest {
    @Test
    public void test01(){
        ApplicationContext context = new
ClassPathXmlApplicationContext("spring/applicationContext-dao.xml");
        UserDao userDao = (UserDao) context.getBean("userDao");
        List<User> user = userDao.findAllUsers();
        System.out.print(user);
    }
}

```

执行结果



18.2 整合方式二

mybatis-spring1.2.3版以上的才有这个。dao实现类继承Support类，直接利用 getSqlSession() 获得，然后直接注入SqlSessionFactory。

比起方式1，不需要管理SqlSessionTemplate，而且对事务的支持更加友好。

SqlSessionDaoSupport

SqlSessionDaoSupport 是一个抽象的支持类，用来为你提供 SqlSession 调用，getSqlSession() 方法你会得到一个 SqlSessionTemplate，之后可以用于执行 SQL 方法，就像下面这样：

```
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {  
    public User getUser(String userId) {  
        return getSqlSession().selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);  
    }  
}
```

在这个类里面，通常更倾向于使用 MapperFactoryBean，因为它不需要额外的代码。但是，如果你需要在 DAO 中做其它非 MyBatis 的工作或需要一个非抽象的实现类，那么这个类就很有用了。

SqlSessionDaoSupport 需要通过属性设置一个 sqlSessionFactory 或 SqlSessionTemplate。如果两个属性都被设置了，那么 SqlSessionFactory 将被忽略。

假设类 UserMapperImpl 是 SqlSessionDaoSupport 的子类，可以编写如下的 Spring 配置来执行设置：

```
<bean id="userDao" class="org.mybatis.spring.sample.dao.UserDaoImpl">  
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />  
</bean>
```

18.2.1 代码示例

UserDaoImpl

```
package cn.guardwhy.dao.impl;  
  
import cn.guardwhy.dao.UserDao;  
import cn.guardwhy.domain.User;  
import org.mybatis.spring.SqlSessionTemplate;  
import org.mybatis.spring.support.SqlSessionDaoSupport;  
  
import java.util.List;  
  
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {  
  
    @Override  
    public List<User> findAllUsers() {  
        UserDao userDao = getSqlSession().getMapper(UserDao.class);  
        return userDao.findAllUsers();  
    }  
}
```

applicationContext-dao.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">
```

```

<!--1. 读取db.properties文件-->
<context:property-placeholder
location="classpath:properties/db.properties"/>

<!--2. 创建数据源-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<!--3. 创建SqlSessionFactory (使用SqlSessionFactoryBean) -->
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <!--注入数据源-->
    <property name="dataSource" ref="dataSource"/>
    <!--绑定Mybatis配置文件-->
    <property name="configLocation" value="classpath:sqlMapConfig.xml"/>
    <property name="mapperLocations"
value="classpath:cn/guardwhy/dao/UserDao.xml"/>
</bean>
<!--4. bean id配置-->
<bean id="userDao" class="cn.guardwhy.dao.impl.UserDaoImpl">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
</beans>

```

测试代码

```

package cn.guardwhy.Test;

import cn.guardwhy.dao.UserDao;
import cn.guardwhy.domain.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

public class SpringMybatisTest {
    @Test
    public void test02(){
        ApplicationContext context = new
ClassPathXmlApplicationContext("spring/applicationContext-dao.xml");
        UserDao userDao = (UserDao) context.getBean("userDao");
        List<User> user = userDao.findAllUsers();
        System.out.print(user);
    }
}

```

执行结果

Run: SpringMybatisTest.test02

Tests passed: 1 of 1 test - 723 ms

SpringMybat 723 ms

test02 723 ms

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...  
三月 23, 2021 10:51:18 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl in  
信息: {dataSource-1} inited  
[User(id=1, username=侯大利, birthday=Fri Oct 24 00:00:00 CST 1980, sex=男, address=江州),  
Process finished with exit code 0
```