Spring学习笔记

资源

1. **下载地址：**https://repo.spring.io/release/org/springframework/spring/
2. docs**文档、**api**文档地址：**https://docs.spring.io/spring-framework/docs/
3. **中文文档地址：**https://www.docs4dev.com/docs/zh/spring-framework/5.1.3.RELEASE/reference
4. github**地址：**https://github.com/spring-projects/spring-framework
5. **笔记地址：**https://github.com/userwusir/study-notes

控制反转（IOC容器）重点

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

Spring**实现控制反转的是**IOC**容器，实现方法：依赖注入**

理解

程序从程序员控制（对象创建）——>用户控制（选择对象）

原理

从前

1. UserDao

```
public interface UserDao {
    /**
     * get user
     */
    void getUser();
}
```

2. UserDaoImpl

```
public class UserDaoImpl implements UserDao {

    @Override
    public void getUser() {
        System.out.println("获取用户信息");
    }
}
```

3. UserService

```
public interface UserService {
    /**
     * get user
     */
    void getUser();
}
```

4. UserServiceImpl

```
public class UserServiceImpl implements UserService{
    private UserDao userDao = new UserDaoImpl();
    @Override
    public void getUser() {
        userDao.getUser();
    }
}
```

5. Test

```
public void test(){
    UserService userService = new UserServiceImpl();
    userService.getUser();
}
```

6. **结果**

```
"C:\Program
获取用户信息
```

问题：此时，若是一个 UserDaoImpl2**继承**UserDao**，实现方法内容不同，若是想要在**Service**层调用**UserDaoImpl2**的方法，则需要修改**Service**层的代码，如下：**

UserDaoImpl2

```
public class UserDaoImpl2 implements UserDao{
    @Override
    public void getUser() {
        System.out.println("获取用户二");
    }
}
```

UserServiceImpl

```
public class UserServiceImpl implements UserService{
    //此处代码做了修改
    private UserDao userDao = new UserDaoImpl2();
    @Override
    public void getUser() {
        userDao.getUser();
    }
}
```

结果：

```
"C:\Progra
获取用户二
```

总结：这是由程序员控制对象的创建

现在

在原有UserDao、UserDaoImpl、UserService不变的情况下进行改进

1. UserServiceImpl

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;

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

    @Override
    public void getUser() {
        userDao.getUser();
    }
}
```

2. Test

```
public void test(){
    UserServiceImpl userService = new UserServiceImpl();
    userService.setUserDao(new UserDaoImpl());
    userService.getUser();
}
```

总结：UserServiceImpl**实现了**setter**方法来获取用户选择创建的**UserDao**对象调用对应**UserDaoImpl**的方法，实现了由程序员控制对象创建到用户控制对象创建。**

Spring配置

1. alias

   给bean**取别名**

   ```
   <alias name="hello" alias="hello2"/>
   ```

2. bean

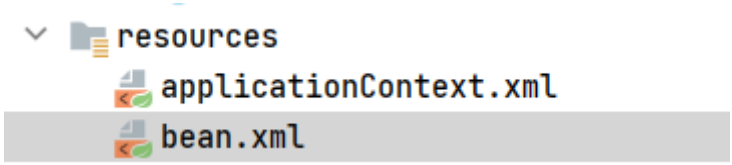
   对象，id**：对象唯一标识符，**class**：对象地址，包名+类型，**name**：取别名，可以取多个（各种分隔符）**

   ```
   <bean id="hello" class="com.wll.pojo.Hello" name="a b,c;d">
       <constructor-arg name="str" value="wuwuwu"/>
   </bean>
   ```

3. import

   团队多人开发，多个配置文件导入到总配置文件

   `example`

   applicationContext.xml

   ```
   <import resource="bean.xml"/>
   ```

HelloSpring

第一个Spring**程序**

```
public class HelloSpring {
    private String str;
```

```
    public String getStr() {
        return str;
    }

    //依赖注入的核心
    public void setStr(String str) {
        this.str = str;
    }

    @Override
    public String toString() {
        return "HelloSpring{" +
            "str='" + str + '\'' +
            '}';
    }
}
```

applicationContext.xml

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

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

    <!--bean相当于new类对象-->
    <bean id="hello" class="com.wll.pojo.HelloSpring">
        <!--property相当于对象的属性字段赋值-->
        <!--value: 外部对象      ref: 配置好的bean-->
        <property name="str" value="Hello Spring"/>
    </bean>

</beans>
```

Test

```
public void test(){
    //加载配置文件
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    //获取对象
    HelloSpring hello = (HelloSpring)
applicationContext.getBean("hello");
    System.out.println(hello.toString());
}
```

结果：

```
"C:\Program Files\Java\jdk1.8.0_
HelloSpring{str='Hello Spring'}
```

总结：对象的创建、管理、装配统一由 Spring（`applicationContext.xml`）**进行管理**

通过Bean创建对象

在applicationContext.xml**中配置**Bean**（相当于**new**对象），默认**pojo**层使用无参构造方法，有参构造方法有三种配置**Bean**方式，在加载**applicationContext.xml**文件的时候对象就已经创建好了。**

1. **无参构造**

   Hello**的无参构造**

   ```
   public Hello() {
       System.out.println("Hello的无参构造");
   }
   ```

   applicationContext.xml

   ```
   <!--bean相当于new类对象-->
   <bean id="hello" class="com.wll.pojo.Hello">
   </bean>
   ```

   Test

   ```
   public void test(){
       ApplicationContext applicationContext = new
   ClassPathXmlApplicationContext("applicationContext.xml");
   }
   ```

   结果：

   ```
   "C:\Program Fil
   Hello的无参构造
   ```

2. **有参构造**

   Hello**的有参构造**

   ```
   public Hello(String str) {
       this.str = str;
       System.out.println("Hello的有参构造");
   }
   ```

1. **方式一，下标（index）**

   applicationContext.xml

```
<!--有参构造index-->
<bean id="hello" class="com.wll.pojo.Hello">
    <constructor-arg index="0" value="wll"/>
</bean>
```

   Test

```
public void test(){
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    Hello hello = (Hello)
applicationContext.getBean("hello");
    System.out.println(hello.toString());
}
```

   结果：

```
"C:\Program Files\Java\
Hello的有参构造
HelloSpring{str='wll'}
```

2. **方式二，类型（type），不建议使用**

```
<!--有参构造type，不建议使用-->
<bean id="hello" class="com.wll.pojo.Hello">
    <constructor-arg type="java.lang.String"
value="wusir"/>
</bean>
```

   结果：

```
"C:\Program Files\Java\jd
Hello的有参构造
HelloSpring{str='wusir'}
```

3. **方式三，参数名（name）**

```
<!--有参构造name-->
<bean id="hello" class="com.wll.pojo.Hello">
    <constructor-arg name="str" value="wuwuwu"/>
</bean>
```

   结果：

```
"C:\Program Files\Java\jdk
Hello的有参构造
HelloSpring{str='wuwuwu'}
```

依赖注入（DI）

程序运行过程中，如果需要调用另一个`对象`协助时，`无须`在代码中创建被调用者，而是依赖于`外部`的注入，对比着控制反转来理解。

1、构造器注入

`查看通过Bean创建对象`

2、Setter方法注入

1. **测试环境**

   pojo

```java
/**
 * @author wulele
 */
public class Email {
    private String email;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "Email{" +
            "email='" + email + '\'' +
            '}';
    }
}
```

```java
/**
 * @author wulele
 */
@Data
public class User {
    private String name;
    private Email email;
    private String[] products;
    private List<String> hobbies;
    private Map<String,String> games;
    private Properties info;
    private String none;
}
```

2. applicationContext.xml

```xml
<!--bean-->
<bean id="email" class="com.wll.pojo.Email">
    <property name="email" value="1415155099@qq.com"/>
</bean>
<bean id="user" class="com.wll.pojo.User">
    <property name="name" value="wll"/>
    <!--ref-->
    <property name="email" ref="email"/>
    <!--array-->
    <property name="products">
        <array>
            <value>手机</value>
            <value>电脑</value>
        </array>
    </property>
    <!--list-->
    <property name="hobbies">
        <list>
            <value>唱</value>
            <value>跳</value>
            <value>Rap</value>
        </list>
    </property>
    <!--map-->
    <property name="games">
        <map>
            <entry key="moba" value="英雄联盟"/>
        </map>
    </property>
    <!--properties-->
    <property name="info">
        <props>
```

```
            <prop key="性别">男</prop>
            <prop key="年龄">20</prop>
        </props>
    </property>
    <!--null-->
    <property name="none">
        <null/>
    </property>
</bean>
```

Test

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    User user = applicationContext.getBean("user",
User.class);
    System.out.println(user.toString());
}
```

结果：

```
User(name=wll,
    email=Email{email='1415155099@qq.com'},
    products=[手机, 电脑], hobbies=[唱, 跳, Rap],
    games={moba=英雄联盟},
    info={性别=男, 年龄=20},
    none=null)
```

3、拓展方式注入

p命名空间和c命名空间注入

pojo

```
/**
 * @author wulele
 */
@Data
public class Game {
    private String type;
    private String name;

    public Game() {
    }
```

```
    public Game(String type, String name) {
        this.type = type;
        this.name = name;
    }
}
```

applicationContext.xml

```
<!—p命名空间，需要无参构造函数—>
<bean id="game" class="com.wll.pojo.Game" p:type="moba"
p:name="LOL"/>
<!—c命名空间，需要有参构造函数—>
<bean id="game2" class="com.wll.pojo.Game" c:type="fps"
c:name="CF"/>
```

Test

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    Game game = applicationContext.getBean("game", Game.class);
    Game game2 = applicationContext.getBean("game2", Game.class);
    System.out.println(game);
    System.out.println(game2);
}
```

结果：

```
"C:\Program Files\Java\jdk
Game(type=moba, name=LOL)
Game(type=fps, name=CF)
```

注意：

配置文件插入

```
xmlns:p="http://www.springframework.org/schema/p"
xmlns:c="http://www.springframework.org/schema/c"
```

Bean作用域

Scope	Description
singleton	(默认)将每个 Spring IoC 容器的单个 bean 定义范围限定为单个对象实例。
prototype	将单个 bean 定义的作用域限定为任意数量的对象实例。
request	将单个 bean 定义的范围限定为单个 HTTP 请求的生命周期。也就是说，每个 HTTP 请求都有一个在单个 bean 定义后面创建的 bean 实例。仅在可感知网络的 Spring `ApplicationContext` 中有效。
session	将单个 bean 定义的范围限定为 HTTP `Session` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。
application	将单个 bean 定义的范围限定为 `ServletContext` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。
websocket	将单个 bean 定义的范围限定为 `WebSocket` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。

`example`

```
<!--bean-->
<bean id="email" class="com.wll.pojo.Email" scope="">
    <property name="email" value="1415155099
</bean>
<bean id="user" class="com.wll.pojo.User"...
```

prototype
request
session
singleton

自动装配

xml实现自动装配

`example`

```
    <bean id="book" class="com.wll.pojo.Book"/>
    <bean id="bookSelf" class="com.wll.pojo.BookSelf" autowire="b"/>
</beans>
```

byType
byName

byName：在容器上下文查找和自己set方法对象一致的类型，注意保证全局id唯一

byType：在容器上下文查找和自己对象属性一致的类型，可以不写id

注解实现自动装配

注意xml文件，需要增加注解支持

```
<?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">
    <!--指定扫描包下的注解都会生效，以及注册javabean，使用这个，
annotation-config可以不用-->
    <context:component-scan base-package="com.wll.pojo"/>
    <!--<context:annotation-config/>-->

</beans>
```

1. Spring**提供的注解**

```
    <bean id="book" class="com.wll.pojo.Book"/>
    <bean id="books" class="com.wll.pojo.Book"/>
    <bean id="bookSelf" class="com.wll.pojo.BookSelf"/>
```

book**和**books**类型相同，**name**不同**

```
@Autowired
@Qualifier(value = "books")
private Book book;
```

Autowired**通过**byType**装配，如果有相同类型，则通过**byName**，可以用在属性字段上，也可用于方法上**

Qualifier**指定**name**去装配，两个搭配使用效果更佳**

2. Java**自带的注解**

```
@Resource(name = "books")
private Book book;
```

类似于Autowired + Qualifier**，开发常用**Autowired

使用注解开发

applicationContext.xml

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">
    <!--指定扫描包下的注解都会生效，可以注册javabean-->
    <context:component-scan base-package="com.wll.pojo"/>
    <!-- <bean id="user" class="com.wll.pojo.User"
scope="singleton">-->
    <!-- <property name="name" value="wll"/>-->
    <!-- </bean>-->
</beans>
```

pojo

```
/**
 * @author wulele
 */
// <bean id="user" class="com.wll.pojo.User"/>
@Component
@Scope("singleton")
public class User {
    // <property name="name" value="wll"/>
    @Value("wll")
    private String name;

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

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

```
}
```

Test

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    User user = applicationContext.getBean("user", User.class);
    System.out.println(user.toString());
}
```

结果：

```
"C:\Program Files\
User{name='wll'}
```

由@Component衍生出的几个注解，在MVC三层架构时使用

- Dao ⟶ @Repository
- Service ⟶ @Service
- Controller ⟶ @Controller

使用@Configuration配置开发

@Configuration**用于定义配置类，代替**xml**文件配置，该类内部包含一个或多个**@Bean**注解的方法，此类通过**AnnotationConfigApplicationContext**类进行扫描，构建**Bean**，初始化**Spring。

1. **使用**@Bean

    1. pojo

    ```
    /**
     * @author wulele
     */
    public class User {
        @Value("wll")
        private String name;
        //下面这两行可不写
        @Autowired
        private User user;

        public String getName() {
            return name;
        }
    }
    ```

    2. UserConfig

```java
/**
 * @author wulele
 */
@Configuration
public class UserConfig {
    //取别名，默认为getUser
    @Bean(name = "user")
    //设置作用域
    @Scope("singleton")
    //描述
    @Description("return user")
    public User getUser() {
        return new User();
    }
}
```

相当于

```xml
<bean id="user" class="com.wll.pojo.User"/>
```

3. Test

```java
public static void main(String[] args) {
    ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(UserConfig.class);
    User user = applicationContext.getBean("user",
User.class);
    System.out.println(user.getName());
}
```

4. **结果**

```
"C:\Pro
wll
```

2. **使用**@Component**和**@ComponentScan**（包扫描）**

1. pojo

```java
/**
 * @author wulele
 */
//取别名，默认user
@Component("getUser")
public class User {
    @Value("wll")
    private String name;

    public String getName() {
        return name;
    }
}
```

2. UserConfig

```java
/**
 * @author wulele
 */
@Configuration
//包扫描，等价于 <context:component-scan base-
package="com.wll.pojo"/>
@ComponentScan("com.wll.pojo")
//导入配置文件
//@Import(UserConfig2.class)
public class UserConfig {
    public User getUser() {
        return new User();
    }
}
```

3. **总结**
    - @Component**注解表明一个类会作为组件类，并告知**Spring**要为这个类创建**bean**，搭配**@ComponentScan**注解使用**
    - @Bean**注解告诉**Spring**这个方法将会返回一个对象，这个对象要注册为**Spring**应用上下文中的**bean**。相较于**@Component**更加灵活**

代理

静态代理

本人（RealSubject）和代理人（ProxySubject）都是对象，实现了同一个主题（Subject）。如果本人太忙，有些工作无法自己亲自完成，就将其交给代理人负责。

1. Subject

```java
/**
 * 主题
 * @author wulele
 */
public interface Subject {
    /**
     * 业务操作
     * @param str
     */
    public void doSomething(String str);
}
```

2. RealSubject

```java
/**
 * 真实对象（本人）
 * @author wulele
 */
public class RealSubject implements Subject {
    @Override
    public void doSomething(String str) {
        System.out.println("something ---> " + str);
    }
}
```

3. ProxySubject

```java
/**
 * 代理对象（代理人）
 * @author wulele
 */
public class ProxySubject implements Subject {
    private RealSubject realSubject = null;

    public ProxySubject() {
        this.realSubject = new RealSubject();
    }

    @Override
    public void doSomething(String str) {
        this.before("ProxySubject");
        realSubject.doSomething(str);
    }

    public void before(String str) {
        System.out.println("something ---> " + str);
    }
}
```

4. Client

```
public class Client {
    public static void main(String[] args) {
        Subject subject = new ProxySubject();
        subject.doSomething("RealSubject");
    }
}
```

5. **结果**

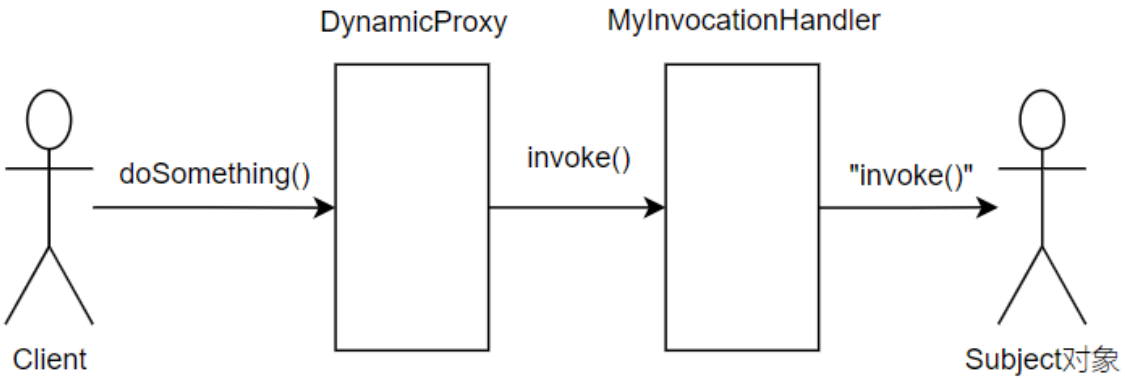
```
"C:\Program Files\Java\jdk1.
something ---> ProxySubject
something ---> RealSubject
```

动态代理

参考《设计模式之禅》，资源地址：链接：https://pan.baidu.com/s/1Pot02W-5DpKZ5zeaHXS-Rg **提取码：**86dz

反射学习地址：注解和反射学习笔记 - 芜湖男酮 - 博客园 (cnblogs.com)

动态代理调用过程示意图：



实现动态代理有两个重要的类和接口 `InvocationHandler` （接口）和 `Proxy` （类），当我们通过动态代理对象调用一个方法时候，这个方法的调用就会被转发到实现 `InvocationHandler` 接口类的invoke()**方法来调用**

1. Subject

```java
/**
 * 主题
 *
 * @author wulele
 */
public interface Subject {
    /**
     * 业务操作
     *
     * @param str
     */
    public void doSomething(String str);
}
```

2. RealSubject

```java
/**
 * 真实对象（本人）
 *
 * @author wulele
 */
public class RealSubject implements Subject {
    @Override
    public void doSomething(String str) {
        System.out.println("real subject ---> " + str);
    }
}
```

3. MyInvocationHandler

```java
/**
 * 动态代理的Handler类
 *
 * @author wulele
 */
public class MyInvocationHandler implements InvocationHandler
{
    /**
     * 接收任意对象
     */
    private Object target;

    public MyInvocationHandler(Object target) {
        this.target = target;
    }

    /**
     * 不是我们显式的调用这个方法
```

```
     */
    @Override
    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
        return method.invoke(this.target, args);
    }
}
```

4. DynamicProxy

```
/**
 * 动态代理类
 * 泛型<T>传入什么类型参数，就返回什么类型的参数
 *
 * @author wulele
 */
public class DynamicProxy<T> {
    public static <T> T newProxy(Subject subject) {
        //获取ClassLoader
        ClassLoader loader =
subject.getClass().getClassLoader();
        //获取接口数组
        Class<?>[] interfaces =
subject.getClass().getInterfaces();
        //获取handler
        MyInvocationHandler handler = new
MyInvocationHandler(subject);
        before("返回代理对象");
        return (T) Proxy.newProxyInstance(loader, interfaces,
handler);
    }

    public static void before(String msg) {
        System.out.println("dynamic proxy ---> " + msg);
    }
}
```

5. Client

```
/**
 * 场景设置
 *
 * @author wulele
 */
public class Client {
    public static void main(String[] args) {
        //定义一个主题
        Subject subject = new RealSubject();
```

```
        //定义主题的代理
        Subject proxy = DynamicProxy.newProxy(subject);
        //代理的行为
        proxy.doSomething("doSomething");
    }
}
```

面向切面编程（AOP）重点

AOP实现机制的核心就是动态代理，将核心类与非核心类进行分离，找到切入点横向扩展代码，不侵入原有代码。

方式一：Spring的API接口

1. UserService

```
/**
 * @author wulele
 */
public interface UserService {
    /**
     * add
     */
    public void add();

    /**
     * delete
     */
    public void delete();
}
```

2. UserServiceImpl

```
/**
 * @author wulele
 */
public class UserServiceImpl implements UserService{
    @Override
    public void add() {
        System.out.println("增加用户");
    }

    @Override
    public void delete() {
        System.out.println("删除用户");
    }
```

```
}
```

3. BeforeLog

```
/**
 * @author wulele
 */
public class BeforeLog implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] objects,
Object o) throws Throwable {
        System.out.println(o.getClass().getName() + "的" +
method.getName()+"()");
    }
}
```

4. AfterLog

```
/**
 * @author wulele
 */
public class AfterLog implements AfterReturningAdvice {
    @Override
    public void afterReturning(Object o, Method method,
Object[] objects, Object o1) throws Throwable {
        System.out.println("返回值" + o + " " +
o1.getClass().getName() + " " + method.getName() + "()");
    }
}
```

5. applicationContext.xml

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

xsi:schemaLocation="http://www.springframework.org/schema/bea
ns
       http://www.springframework.org/schema/beans/spring-
beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-
aop.xsd">
    <bean id="userService"
class="com.wll.service.UserServiceImpl"/>
    <bean id="beforeLog" class="com.wll.log.BeforeLog"/>
```

```
    <bean id="afterLog" class="com.wll.log.AfterLog"/>
    <aop:config>
        <!--切入点 expression:表达式 execution(要执行的位置)-->
        <aop:pointcut id="pointcut" expression="execution(*
com.wll.service.UserServiceImpl.*(..))"/>
        <!--环绕增强-->
        <aop:advisor advice-ref="afterLog" pointcut-
ref="pointcut"/>
        <aop:advisor advice-ref="beforeLog" pointcut-
ref="pointcut"/>
    </aop:config>
</beans>
```

6. Test

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //代理的是接口
        UserService userService =
applicationContext.getBean("userService", UserService.class);
        userService.add();
    }
}
```

7. **结果**

```
"C:\Program Files\Java\jdk1.8.0_281\bin\java.ex
com.wll.service.UserServiceImpl的add()
增加用户
返回值null com.wll.service.UserServiceImpl add()
```

方式二：自定义类

1. MyLog

```java
/**
 * @author wulele
 */
public class MyLog {
    public void before(){
        System.out.println("method before");
    }
    public void after(){
        System.out.println("method after");
    }
}
```

2. applicationContext.xml

```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/bea
ns

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="userService"
class="com.wll.service.UserServiceImpl"/>
    <bean id="myLog" class="com.wll.diylog.MyLog"/>
    <aop:config>
        <!--自定义切面-->
        <aop:aspect ref="myLog">
            <!--切入点-->
            <aop:pointcut id="pointcut"
expression="execution(* com.wll.service.UserServiceImpl.*
(..))"/>
            <!--切入方法前-->
            <aop:before method="before" pointcut-
ref="pointcut"/>
            <!--切入方法后-->
            <aop:after method="after" pointcut-
ref="pointcut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

方式三：注解

1. AnnotationLog

```
/**
 * Aspect 标记这个类是一个切面
 * @author wulele
 */
@Aspect
public class AnnotationLog {
    @Before("execution(* com.wll.service.UserServiceImpl.*
(..))")
    public void before() {
        System.out.println("method before");
    }

    @After("execution(* com.wll.service.UserServiceImpl.*
(..))")
    public void after() {
        System.out.println("method after");
    }

    @Around("execution(* com.wll.service.UserServiceImpl.*
(..))")
    public void around(ProceedingJoinPoint pj) throws
Throwable {
        //pj参数代表我们要切入的点
        System.out.println("around before");
        //打印签名
        System.out.println(pj.getSignature());
        //执行方法
        pj.proceed();
        System.out.println("around after");
    }
}
```

2. applicationContext.xml

```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/bea
ns

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="userService"
class="com.wll.service.UserServiceImpl"/>
    <bean id="annotationLog"
class="com.wll.auto.AnnotationLog"/>
    <!--开启注解支持   false(JDK默认)  true(cglib)-->
    <aop:aspectj-autoproxy proxy-target-class="true"/>
</beans>
```

3. **结果**

```
"C:\Program Files\Java\jdk1.8.0_281\bin\jav
around before
void com.wll.service.UserServiceImpl.add()
method before
增加用户
method after
around after
```

整合Mybatis

官网： mybatis-spring –

pom.xml**导入下列包**

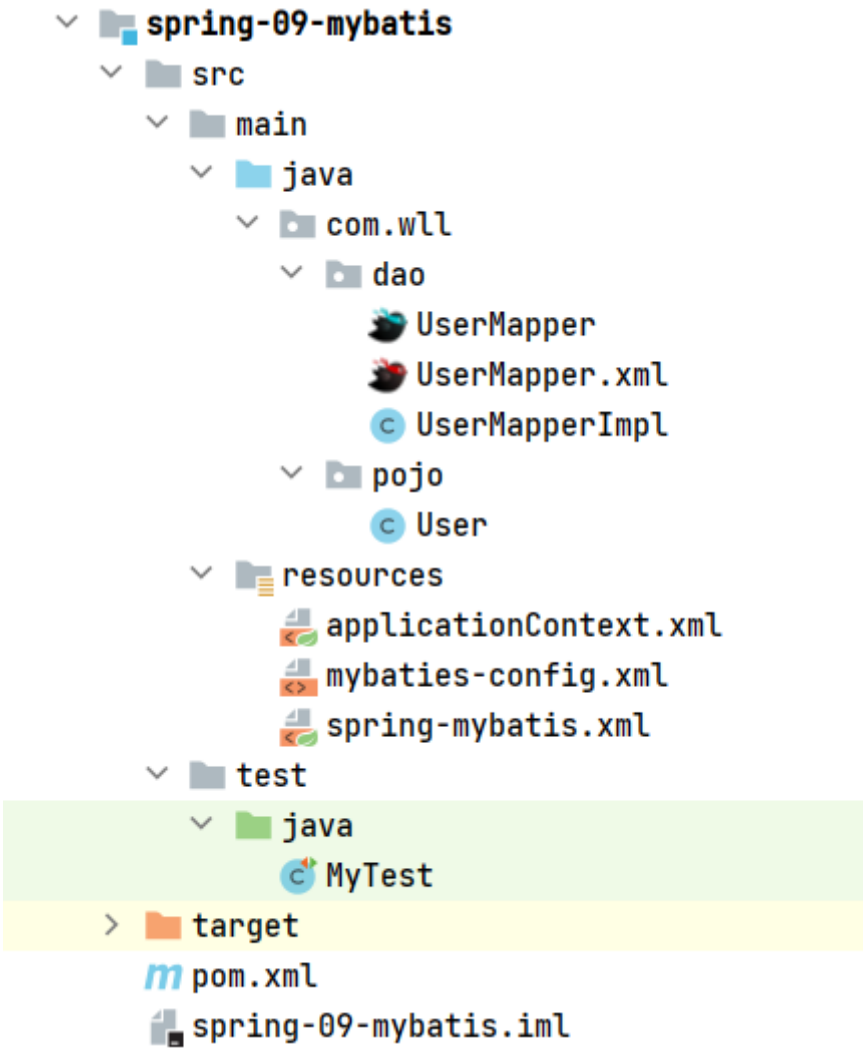
`mybatis、mysql-connector-java、aspectjweaver、mybatis-spring、spring-webmvc、spring-jdbc、junit`

编写 `User、UserMapper、UserMapper.xml、mybatis-config.xml`

参考文章： Mybatis学习笔记 - 芜湖男酮 - 博客园 (cnblogs.com)

方式一

1. **文件结构**



2. User

```java
/**
 * @author wulele
 */
@Data
public class User {
    private int id;
    private String name;
    private String pwd;
}
```

3. UserMapper

```java
/**
 * @author wulele
 */
public interface UserMapper {
    /**
     * get user list
     *
     * @return
     */
    List<User> getUser();
}
```

4. UserMapper.xml

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

5. UserMapperImpl

```java
/**
 * @author wulele
 */
public class UserMapperImpl implements UserMapper{
    /**
     * 以前使用sqlSession,现在使用SqlSessionTemplate
     */
    private SqlSessionTemplate sqlSession = null;

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

    @Override
    public List<User> getUser() {
        UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
        return mapper.getUser();
    }
}
```

6. mybitas-config.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>
    <typeAliases>
        <package name="com.wll.pojo"/>
    </typeAliases>
</configuration>
```

7. spring-mybatis.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/bea
ns

http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <!--只为简化这几句
        String source = "mybaties-config.xml";
        InputStream inputStream =
Resources.getResourceAsStream(source);
        SqlSessionFactory sessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        SqlSession sqlSession =
sessionFactory.openSession(true);
    -->
    <!--配置数据源-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataS
ource">
        <property name="driverClassName"
value="com.mysql.jdbc.Driver"/>
        <property name="url"
                  value="jdbc:mysql://localhost/mybatis?
useSSL=true&amp;useUnicode=true&amp;characterEncoding=utf-
8"/>
        <property name="username" value="root"/>
        <property name="password" value="123123"/>
    </bean>

    <!--sqlSessionFactory-->
```

```
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--绑定数据源-->
        <property name="dataSource" ref="dataSource" />
        <!--绑定Mybatis配置-->
        <property name="configLocation" value="mybaties-
config.xml" />
        <!--mapper文件注册-->
        <property name="mapperLocations"
value="com/wll/dao/*.xml" />
    </bean>

    <!--SqlSessionTemplate就是我们使用的sqlSession-->
    <bean id="sqlSession"
class="org.mybatis.spring.SqlSessionTemplate">
        <!--构造器注入sqlSessionFactory-->
        <constructor-arg index="0" ref="sqlSessionFactory" />
    </bean>

</beans>
```

8. 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/bea
ns
       http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <!--导入spring与mybatis整合的部分-->
    <import resource="spring-mybatis.xml" />
    <!--注册我们需要使用的Bean-->
    <bean id="userMapperImpl"
class="com.wll.dao.UserMapperImpl">
        <property name="sqlSession" ref="sqlSession" />
    </bean>
</beans>
```

9. Test

```
public class MyTest {
    @Test
    public void test() throws IOException {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserMapper mapper =
applicationContext.getBean("userMapperImpl",
UserMapperImpl.class);
        List<User> userList = mapper.getUser();
        for (User user : userList) {
            System.out.println(user);
        }
    }
}
```

10. **结果**

```
Loading class `com.mysql.jdbc.Drive
User(id=1, name=jack, pwd=123123)
User(id=2, name=tom, pwd=123123)
User(id=3, name=pick, pwd=123123)
User(id=4, name=bob, pwd=1234)
```

11. **总结**

    将mybatis**的配置简化，最后注册成为一个**bean**来让我们获取所需的**sqlsession**，不再用为了**sqlsession**而写一个工具类，但是多了一个**UserMapperImpl**的实现类来返回查询结果（加了一层）。**

方式二

在方式一的基础上选择使用SqlSessionDaoSupport**而不是**SqlSessionTemplate

1. **修改**UserMapperImpl

```
/**
 * @author wulele
 */
public class UserMapperImpl extends SqlSessionDaoSupport
implements UserMapper{
    @Override
    public List<User> getUser() {
        return
getSqlSession().getMapper(UserMapper.class).getUser();
    }
}
```

2. **修改**spring-mybatis.xml，**删除以下部分**

```
<!—SqlSessionTemplate就是我们使用的sqlSession—>
<bean id="sqlSession"
class="org.mybatis.spring.SqlSessionTemplate">
    <!—构造器注入sqlSessionFactory—>
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>
```

3. **修改**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/bea
ns
       http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <!—导入spring与mybatis整合的部分—>
    <import resource="spring-mybatis.xml"/>

    <bean id="userMapperImpl"
class="com.wll.dao.UserMapperImpl">
        <!—注意是工厂—>
        <property name="sqlSessionFactory"
ref="sqlSessionFactory"/>
    </bean>
</beans>
```

4. **总结**

   将我们原本需要去获取SqlSessionTemplate**的过程封装在了**SqlSessionDaoSupport**中，简化了操作。**

事务管理

- **声明式事务：**AOP**切入，不影响原有代码**
- **编程式事务：在代码中进行事务管理**

声明式事务

1. UserMapper

```
/**
 * @author wulele
 */
```

```
public interface UserMapper {
    /**
     * get user list
     *
     * @return
     */
    List<User> getUser();

    /**
     * add user
     *
     * @param user
     * @return
     */
    int addUser(User user);

    /**
     * delete user
     *
     * @param id
     * @return
     */
    int deleteUser(int id);
}
```

2. UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.wll.mapper.UserMapper">
    <insert id="addUser">
        insert into user(id,name,pwd) values(#{id},#{name},#
{pwd})
    </insert>
    <!--制造错误，体现事务作用-->
    <delete id="deleteUser">
        deletes from user where id = #{id}
    </delete>
    <select id="getUser" resultType="user">
        select * from mybatis.user
    </select>
</mapper>
```

3. UserMapperImpl

```
/**
```

```
 * @author wulele
 */
public class UserMapperImpl extends SqlSessionDaoSupport
implements UserMapper {

    @Override
    public List<User> getUser() {
        User user = new User(5, "xxx", "qwe");
        UserMapper mapper =
getSqlSession().getMapper(UserMapper.class);
        //先执行添加用户
        mapper.addUser(user);
        //再执行删除用户，如果删除失败，按照事务原子性应该回滚
        mapper.deleteUser(5);
        return mapper.getUser();
    }

    @Override
    public int addUser(User user) {
        return
getSqlSession().getMapper(UserMapper.class).addUser(user);
    }

    @Override
    public int deleteUser(int id) {
        return
getSqlSession().getMapper(UserMapper.class).deleteUser(id);
    }
}
```

4. spring-mybatis.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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/bea
ns
       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/aop
       http://www.springframework.org/schema/aop/spring-
aop.xsd">
    <!--配置数据源-->
```

```
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataS
ource">
        <property name="driverClassName"
value="com.mysql.jdbc.Driver"/>
        <property name="url"
                  value="jdbc:mysql://localhost/mybatis?
useSSL=true&amp;useUnicode=true&amp;characterEncoding=utf-
8"/>
        <property name="username" value="root"/>
        <property name="password" value="123123"/>
    </bean>
    <!--sqlSessionFactory-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--绑定数据源-->
        <property name="dataSource" ref="dataSource"/>
        <!--绑定Mybatis配置-->
        <property name="configLocation" value="mybaties-
config.xml"/>
        <!--mapper文件注册-->
        <property name="mapperLocations"
value="com/wll/mapper/*.xml"/>
    </bean>

    <!--以下为Spring事务配置，基本就是固定格式，如果不进行这些配置，那
么错误的delete语句不会被执行，但是add语句会被执行，违背了事务原子性，而
开启事务后不会执行add语句-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransact
ionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--配置事务-->
    <tx:advice id="txAdvice" transaction-
manager="transactionManager">
        <tx:attributes>
            <!--add开启事务，propagation为REQUIRED-->
            <tx:method name="add" propagation="REQUIRED"/>
            <tx:method name="delete" propagation="REQUIRED"/>
            <!--所有开启事务-->
            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
    <!--aop事务切入-->
    <aop:config>
        <aop:pointcut id="txPointCut" expression="execution(*
com.wll.mapper.*.*(..))"/>
```

```
        <aop:advisor advice-ref="txAdvice" pointcut-
ref="txPointCut"/>
    </aop:config>
</beans>
```

5. 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/bea
ns
       http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <!--导入spring与mybatis整合的部分-->
    <import resource="spring-mybatis.xml"/>
    <!--注册我们需要使用的Bean-->
<!--    <bean id="userMapperImpl"
class="com.wll.dao.UserMapperImpl">-->
<!--        <property name="sqlSession" ref="sqlSession"/>-->
<!--    </bean>-->

    <bean id="userMapperImpl"
class="com.wll.mapper.UserMapperImpl">
        <property name="sqlSessionFactory"
ref="sqlSessionFactory"/>
    </bean>
</beans>
```