1. AOP
2. 为什么使用AOP

SpringAOP是一种约定流程的编程，它可以将大量重复的流程通过约定的方式抽取出来，然后给予默认的实现，这样我们只需要完成部分的功能即可，那些恼人的重复流程可以留给SpringAOP完成。

1. AOP术语
2. 连接点（join point）

指的是具体被拦截的对象，因为Spring只能支持方法，所以被拦截的对象就是指特定的方法。

1. 切点（cut point）

有时不仅仅只有一个连接点，也可能是多个类的不同方法，我们可以通过正则表达式和指示器的规则来适配连接点，切点就是这种适配规则。

1. 通知（advice）

按照约定的流程中的方法，分为前置通知（before advice）,后置通知（after advice），环绕通知（around advice），事后返回通知（afterReturning advice），异常通知（afterThrowing advice）。

1. 目标对象（target）

被代理的对象。

1. 引入（introduction）

引入新的类和其方法，增强现有的类。

1. 织入（waving）

通过动态代理技术，为原有的服务对象生成代理对象，然后将于切点定义匹配的连接点拦截，并按照约定将各类通知织入约定流程的过程。

1. 切面（aspect）

一个可以定义切点，各类通知和引入的内容，SpringAOP将通过他来增强Bean的功能或者将对应的方法织入流程。

1. AOP实例解析
2. 环境介绍

这里，我们使用Spring Boot进行AOP实例测试。pom.xml如下所示：

<?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>com.spb</groupId>  
 <artifactId>springbootTest2</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <parent>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <groupId>org.springframework.boot</groupId>  
 <version>2.1.4.RELEASE</version>  
 </parent>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-aop</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.aspectj</groupId>  
 <artifactId>aspectjweaver</artifactId>  
 <version>1.9.3</version>  
 </dependency>  
 </dependencies>  
  
</project>

1. 确定连接点

连接点就是我们需要拦截的方法，可能是需要增强功能，可能是需要在该方法执行前后添加一些输出字段，也可能是要在该方法出现异常时做出特定的操作。

这里，我们设置一个UserService接口，一个UserServiceImpl实现类，PrintUser方法就是连接点。

UserService接口：

package com.springboot.chapter4.aspect.service;  
  
import com.springboot.chapter4.aspect.pojo.User;  
  
public interface UserService {  
 public void PrintUser(User user);  
}

UserServiceImpl实现类：

package com.springboot.chapter4.aspect.service.impl;  
  
import com.springboot.chapter4.aspect.pojo.User;  
import com.springboot.chapter4.aspect.service.UserService;  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserServiceImpl implements UserService {  
 @Override  
 public void PrintUser(User user) {  
 if(user == null){  
 throw new RuntimeException("检查用户参数是否为空...");  
 }  
 System.*out*.println("id= " + user.getId());  
 System.*out*.println("\tusername = " + user.getUsername());  
 System.*out*.println("\tpassword = " + user.getPassword());  
 }  
}

1. 创建切面

创建一个切面类，在切面中定义AOP的切点，通知，引入。注意，这里的注解@Aspect需要引入spring-boot-starter-aop的启动依赖，声明这是一个切面！！！

1. 定义切点

切点是一个匹配连接点的规则，使用的是正则表达式。切点也是一个方法，但是需要添加注解@Pointcut()，括号中是匹配规则，匹配的规则如下所示：

@Pointcut("execution(\* com.springboot.chapter4.aspect.service.UserService.PrintUser(..))")  
public void pointCut(){}

1. execution： 表示在执行的时候，拦截里面的正则匹配的方法
2. \*： execution的括号内就是一个类似函数签名的样式，\*在函数返回值的位置表示返回任意类型。
3. \*后面跟着的是连接点的全限定名
4. （..），表示着任意参数进行匹配
5. 定义通知

不同的通知有不同的注解标识，@Before, @After, @AfterReturning, @AfterThrowing， 注解中可以引用上面定义的切点函数，表示使用该切点的规则匹配连接点，如下所示：

@Before("pointCut()")  
public void before(){  
 System.*out*.println("before");  
}  
  
@Around("pointCut()")  
public void around(ProceedingJoinPoint jp) throws Throwable {  
 System.*out*.println("around before...");  
 //回调目标对象的原有方法  
 jp.proceed();  
 System.*out*.println("around after...");  
}  
  
@After("pointCut()")  
public void after(){  
 System.*out*.println("after");  
}  
  
@AfterReturning("pointCut()")  
public void afterReturning(){  
 System.*out*.println("afterReturning");  
}  
  
@AfterThrowing("pointCut()")  
public void afterrThrowing(){  
 System.*out*.println("afterThrowing");  
}

1. 环绕通知

环绕通知很强大，使用的场景是在需要大幅度地修改原有目标对象的服务逻辑时，它会取代原有目标对象方法，或者回调原有目标对象方法。它很强大，也很不稳定，能不用就不用。

在使用时与其他的通知一样，都需要设置在切面类中，不过它有一个ProceedingJoinPoint类型的参数，这个参数的对象有一个proceed方法，通过这个方法可以回调原有目标对象的方法。如下所示：

@Around("pointCut()")  
public void around(ProceedingJoinPoint jp) throws Throwable {  
 System.*out*.println("around before...");  
 //回调目标对象的原有方法  
 jp.proceed();  
 System.*out*.println("around after...");  
}

1. 引入

我们可以引入其他的接口来增强现有接口的功能，如通过检测用户信息是否为空，如果为空，则不需要打印操作，也就不会有异常发生。因此，我们新建一个验证接口和实现类，用于增强现有的UserServiceImpl类。这个引入操作需要定义在切面类中，使用注解@DeclareParents()，该注解有两个必填的属性value和defaultImpl。value表示要增强功能的目标对象的全限定名；defaultImpl表示用于增强功能的类。在使用该增强的类时，可以直接将原有的UserService类强制转换为增强的接口UserValidator，调用该接口中的增强方法。如下所示：

验证接口UserValidator:

package com.springboot.chapter4.aspect.validator;  
  
import com.springboot.chapter4.aspect.pojo.User;  
  
public interface UserValidator {  
 public boolean validate(User user);  
}

验证接口实现类UserValidatorImpl:

package com.springboot.chapter4.aspect.validator.impl;  
  
import com.springboot.chapter4.aspect.pojo.User;  
import com.springboot.chapter4.aspect.validator.UserValidator;  
  
public class UserValidatorImpl implements UserValidator {  
 @Override  
 public boolean validate(User user) {  
 System.*out*.println("引入新的接口： " + UserValidator.class.getSimpleName());  
 return user != null;  
 }  
}

切面中引入的定义：

@DeclareParents(value = "com.springboot.chapter4.aspect.service.impl.UserServiceImpl",  
 defaultImpl = UserValidatorImpl.class )  
public UserValidator userValidator;

测试引入的验证器：

//强制转换  
UserValidator userValidator = (UserValidator)userService;  
//验证用户是否为空  
if(userValidator.validate(user)){  
 userService.PrintUser(user);  
}

1. 引入的原理

JDK动态代理下会挂着两个接口，一个是原目标对象UserService，一个是引入的类UserValidator，我们可以将代理对象通过这两个接口相互转换，然后调度其对应的方法。

1. 通知获取参数

我们可以在通知中传递参数，只需要在切点处添加对应的正则式即可，对于非环绕通知，还可以使用一个连接点（JoinPoint）类型的参数，通过它也可以获取参数。如下所示：

@Befor(“pointCut() && args(user)”)

public void beforeParam(JoinPoint point, User user){

Object[] args = point.getArgs();

System.out.println(“before…”);

}

JoinPoint类型的参数对于非环绕通知而言，Spring AOP会自动将它传递到通知中。

1. 织入

Spring 推崇使用接口 + 实现类的方式进行织入，事实上，也可以只使用类的方式进行织入。动态代理有多种实现方式，我们使用的一般是JDK，它要求被代理的目标对象必须有接口，而对于CGLIB而言，是不需要接口的。

在默认的情况下，当你的代理对象有接口时，它会以JDK动态代理的方式运行，否则以CGLIB运行。