Day07

通知获取数据

通过通知，可以获取方法中参数、返回值、异常的数据。

参数

方式 1

设定通知方法第一个参数为JoinPoint，通过该对象调用getArgs()方法，获取原始方法运行的参数数组
通知方法：

Java 复制代码

```java
public void before(JoinPoint joinPoint) throws Throwable {
    Object[] args = joinPoint.getArgs();
}
```

所有的通知均可以获取参数。

方式 2

通过切入点表达式可以为通知方法传递参数，然后在通知方法中获取。
方法：

Java 复制代码

```java
@Override
public void add(int p1, int p2) {
    System.out.println("p1=" + p1 + "p2=" + p2);
}
```

aop配置：

```xml
<aop:before method="testParam" pointcut="execution(* *(..)) &amp;&amp;
args(a, b)"/>
```

通知方法：

```java
public void testParam(int a, int b) {
    System.out.println("获取切点参数：a=" + a + ",  b=" + b);
}
```

返回值

第一种：

通过返回值变量名获取，适用于返回后通知（after-returning）

方法：

```java
public int save() {
    System.out.println("user service running...");
    return 100;
}
```

AOP 配置

```xml
<aop:aspect ref="myAdvice">
  <aop:pointcut id="pt" expression="execution(* *(..))  "/>
  <aop:after-returning method="afterReturning" pointcut-ref="pt"
returning="ret"/>
</aop:aspect>
```

通知方法

```java
public void afterReturning(Object ret) {
    System.out.println(ret);
}
```

第二种

通过在通知类的方法中调用原始方法获取，适用于环绕通知（around）

原始方法

```java
public int save() {
    System.out.println("user service running...");
    return 100;
}
```

AOP配置

```xml
<aop:aspect ref="myAdvice">
  <aop:pointcut id="pt" expression="execution(* *(..))  "/>
  <aop:around method="around" pointcut-ref="pt" />
</aop:aspect>
```

通知类

```java
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object ret = pjp.proceed();
    return ret;
}
```

异常数据

第一种

通知类的方法中调用原始方法捕获异常，适用于环绕通知（around）

原始方法

```java
public void save() {
    System.out.println("user service running...");
    int i = 1/0;
}
```

AOP配置

```xml
<aop:aspect ref="myAdvice">
  <aop:pointcut id="pt4" expression="execution(* *(..))  "/>
  <aop:around method="around" pointcut-ref="pt4" />
</aop:aspect>
```

通知方法

```java
public Object around(ProceedingJoinPoint pjp) throws Throwable {
  Object ret = pjp.proceed(); //对此处调用进行try……catch……捕获异常，或抛出异常
  return ret;
}
```

第二种

设定异常对象变量名，适用于返回后通知（after-throwing）

原始方法

```java
public void save() {
    System.out.println("user service running...");
    int i = 1/0;
}
```

AOP配置

```xml
<aop:aspect ref="myAdvice">
  <aop:pointcut id="pt4" expression="execution(* *(..))  "/>
  <aop:after-throwing method="afterThrowing" pointcut-ref="pt4"
throwing="t"/>
</aop:aspect>
```

通知方法

```java
public void afterThrowing(Throwable t){
    System.out.println(t.getMessage());
}
```

课堂作业

- 完成 AOP 的通知获取数据案例

AOP 配置注解

步骤

- 创建 maven 项目，添加依赖
- 开启AOP注解支持（<aop:aspectj-autoproxy />）
- 配置切面 @Aspect
- 定义专用的切入点方法，并配置切入点 @Pointcut
- 为通知方法配置通知类型及对应切入点，如 @Before

注意事项

1. 切入点最终体现为一个方法，可以无参无返回值，方法体无内容。
2. 切入点不能是抽象方法。
3. 引用切入点时，必须使用方法名，且后面的（）不能省略。
4. 切面类中定义的切入点只能在当前类中使用，如果想引用其他类中定义的切入点使用“类名.方法名()”引用
5. 可以在通知类型注解中添加参数（逗号分隔），实现 XML 配置中的属性，例如 after-returning 后的 returning 属性

AOP注解

@Aspect

在类定义的上方添加 @Aspect 注解，这将把该类设置为切面

```java
@Aspect
public class AopAdvice {
}
```

@Pointcut

在方法定义的上方添加 @Pointcut 注解，这将把该方法设置为切入点，切入点引用名称为方法名。

```java
@Pointcut("execution(* *(..))")
public void pt() {
}
```

@Before

在方法定义的上方添加 @Before 注解，这将把该方法设置为前置通知。

```java
@Before("pt()")
public void before(){
}
```

@After

在方法定义的上方添加 @After 注解，这将把该方法设置为后置通知。

```java
@After("pt()")
public void after(){
}
```

@AfterReturning

在方法定义的上方添加 @AfterReturning 注解，这将把该方法设置为返回后通知。

```java
@AfterReturning(value="pt()",returning = "ret")
public void afterReturning(Object ret) {
}
```

参数：returning ： 设定使用通知方法参数接收返回值的变量名

@AfterThrowing

在方法定义的上方添加 @AfterThrowing 注解，这将把该方法设置为异常后通知。

```java
@AfterThrowing(value="pt()",throwing = "t")
public void afterThrowing(Throwable t){
}
```

参数：throwing ： 设定使用通知方法参数接收原始方法中抛出的异常对象名

@Around

在方法定义的上方添加 @Around 注解，这将把该方法设置为环绕通知。

```java
@Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    Object ret = pjp.proceed();
    return ret;
}
```

传参

```java
@Pointcut("execution(* *(..)) && args(a, b)")
public void pt(int a, int b) {

}

@Before("pt(a, b)")
public void testAdd1(int a, int b) {
    System.out.println("通知获取参数1：a= " + a + ", b= " + b);
}
@Before(value = "execution(* *(..)) && args(a, b)")
public void testAdd2(int a, int b) {
    System.out.println("通知获取参数2：a= " + a + ", b= " + b);
}
```

AOP注解开发通知执行顺序控制

- AOP使用XML配置情况下，通知的执行顺序由配置顺序决定。但是在注解情况下由于不存在配置顺序的概念的概念，参照通知所配置的方法名字符串对应的编码值顺序，可以简单理解为字母排序。
- 同一个通知类中，相同通知类型以方法名排序为准。
- 不同通知类中，以类名排序为准。
- 使用@Order注解通过变更bean的加载顺序改变通知的加载顺序

企业开发经验

- 通知方法名由3部分组成，分别是前缀、顺序编码、功能描述
- 前缀为固定字符串，如 seehope，baidu 等，无实际意义
- 顺序编码为 6 位以内的整数，通常 3位即可，不足位补 0
- 功能描述为该方法对应的实际通知功能，例如 exception、strLenCheck
- 通知执行顺序使用顺序编码控制，使用时做一定空间预留
- 003使用，006使用，预留001、002、004、005、007、008
  - 使用时从中段开始使用，方便后期做前置追加或后置追加
  - 最终顺序以运行顺序为准，以测试结果为准，不以设定规则为准

开启 AOP 自动代理注解

Spring注解配置类定义上方使用 @EnableAspectJAutoProxy 可以开启 AOP 自动注解驱动的支持，实现自动加载AOP注解。

Java 复制代码

```java
@Configuration
@ComponentScan("com.rushuni")
@EnableAspectJAutoProxy
public class SpringConfig {
}
```

课堂作业

- 完成 AOP配置注解案例，熟悉 AOP 配置的注解。

AOP 实战案例一：方法耗时统计

需求

对项目进行业务层接口执行监控，测量如下业务层接口的执行效率：

Java 复制代码

```java
public interface AccountService {

    void save(Account account);

    void delete(Integer id);

    void update(Account account);

    List<Account> listAll();

    Account getById(Integer id);
}
```

分析

- 测量接口执行效率：接口方法执行前后获取执行时间，求出执行时长 – System.currentTimeMillis( )
- 对项目进行监控：项目中所有接口方法，AOP思想，执行期动态织入代码
  - 环绕通知
  - proceed()方法执行前后获取系统时间

总结就是使用 Around advice, 然后在方法调用前, 记录一下开始时间, 然后在方法调用结束后, 记录结束时间, 它们的时间差就是方法的调用耗时.

开发步骤

- 定义切入点
- 编写 AOP 环绕通知，完成测量功能
- 注解配置AOP
- 开启注解驱动支持

核心代码

Java 复制代码

```java
package com.rushuni.sm_annotation.aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

/**
 * @author rushuni
 * @date 2021年07月20日 4:07 下午
 */

@Component
@Aspect
public class RunTimeMonitorAdvice {
    //切入点，监控业务层接口
    @Pointcut("execution(* com.rushuni.sm_annotation.service.*Service.list*(..))")
    public void pt(){}

    @Around("pt()")
    public Object runtimeAround(ProceedingJoinPoint pjp) throws Throwable {
        //获取执行签名信息
        Signature signature = pjp.getSignature();
        //通过签名获取执行类型（接口名）
        String targetClass = signature.getDeclaringTypeName();
        //通过签名获取执行操作名称（方法名）
        String targetMethod = signature.getName();
        //获取操作前系统时间beginTime
        long beginTime = System.currentTimeMillis();
        Object ret = pjp.proceed(pjp.getArgs());
        //获取操作后系统时间endTime
        long endTime = System.currentTimeMillis();
        System.out.println(targetClass+" 中 "+targetMethod+" 运行时长 "+(endTime-beginTime)+"ms");
        return ret;
    }
}
```

AOP 实战案例二：方法调用日志

记录一个方法调用的 log 也是一个很常见的功能。

需求:

某个服务下的方法调用需要有 log，记录调用的参数以及返回结果。

当方法调用出异常时, 有特殊处理, 例如打印异常 log, 报警等。

根据上面的需求, 我们可以:

- 使用 before advice 来在调用方法前打印调用的参数。
- 使用 after returning advice 在方法返回打印返回的结果。
- 而当方法调用失败后, 可以使用 after throwing advice 来做相应的处理。

核心

Java 复制代码

```java
@Pointcut("execution(* *(..))")
public void pointcut() {
}

@Before("pointcut()")
public void logMethodInvokeParam(JoinPoint joinPoint) {
    System.out.println("---Before method {"
                       + joinPoint.getSignature().toShortString()
                       + "} invoke, param: {"
                       + joinPoint.getArgs()
                       + "}---");
}

@AfterReturning(pointcut = "pointcut()", returning = "retVal")
public void logMethodInvokeResult(JoinPoint joinPoint, Object retVal) {
    System.out.println("---After method {"
                       + joinPoint.getSignature().toShortString()
                       + "} invoke, result: {"
                       + retVal
                       + "}---");
}

@AfterThrowing(pointcut = "pointcut()", throwing = "exception")
public void logMethodInvokeException(JoinPoint joinPoint, Exception
exception) {
    System.out.println("---method {"
                       + joinPoint.getSignature().toShortString()
                       + "} invoke, exception: {"
                       + exception.getMessage()
                       + "}---");
}
```

课后作业

- 使用 AOP 技术，监测 DAO 层接口各方法的运行时长。
- 使用 AOP 技术，记录 Service 层接口各方法的调用日志。