AOP 面向切面编程

AOP 术语：

* 切面(Aspect): **横切关注点(跨越应用程序多个模块的功能)被模块化的特殊对象**
* 通知(Advice): **切面必须要完成的工作**
* 目标(Target): **被通知的对象**
* 代理(Proxy): **向目标对象应用通知之后创建的对象**
* 连接点（Joinpoint）：**程序执行的某个特定位置**：如类某个方法调用前、调用后、方法抛出异常后等。**连接点由两个信息确定：方法表示的程序执行点；相对点表示的方位**。例如 ArithmethicCalculator#add() 方法执行前的连接点，执行点为 ArithmethicCalculator#add()； 方位为该方法执行前的位置
* 切点（pointcut）：**每个类都拥有多个连接点**：例如 ArithmethicCalculator 的所有方法实际上都是连接点，即**连接点是程序类中客观存在的事务**。**AOP 通过切点定位到特定的连接点。类比：连接点相当于数据库中的记录，切点相当于查询条件**。切点和连接点不是一对一的关系，一个切点匹配多个连接点，切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

**AspectJ：java社区里最完整最流行的AOP框架。**

可以使用基于AspectJ 注解或基于XML配置的AOP。

1. 使用注解的方式

SpringAOP

* 1. 加入jar 包



* 1. 在配置文件中加入aop 命名空间
  2. 基于注解的方式
     1. 在配置文件中加入如下配置：

<!-- 使AspjectJ 注解起作用：自动为匹配的类生成代理对象 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

* + 1. 把横切关注点的代码抽象到切面的类中

切面首先是一个IOC中的bean，即加入@Component注释

切面还需要加入@Aspect注释

* 1. 在类中声明各种通知：

声明一个方法

在方法前加入@Before 注解

* 1. 可以在通知方法中声明一个类型为JoinPoint 的参数，然后就能访问链接细节，如方法名称和参数值。

//把这个类声明为一个切面： 需要先把该类放入到IOC 容器中，再声明为一个切面

@Aspect

@Component

**public** **class** LogginAspect {

//声明该方法是一个前置通知：目标方法开始之前执行

@Before("execution(publicint spring.impl.ArithmeticCalculator.\*(int, int))")

**public** **void** beforeMethod(JoinPoint joinPoint) {

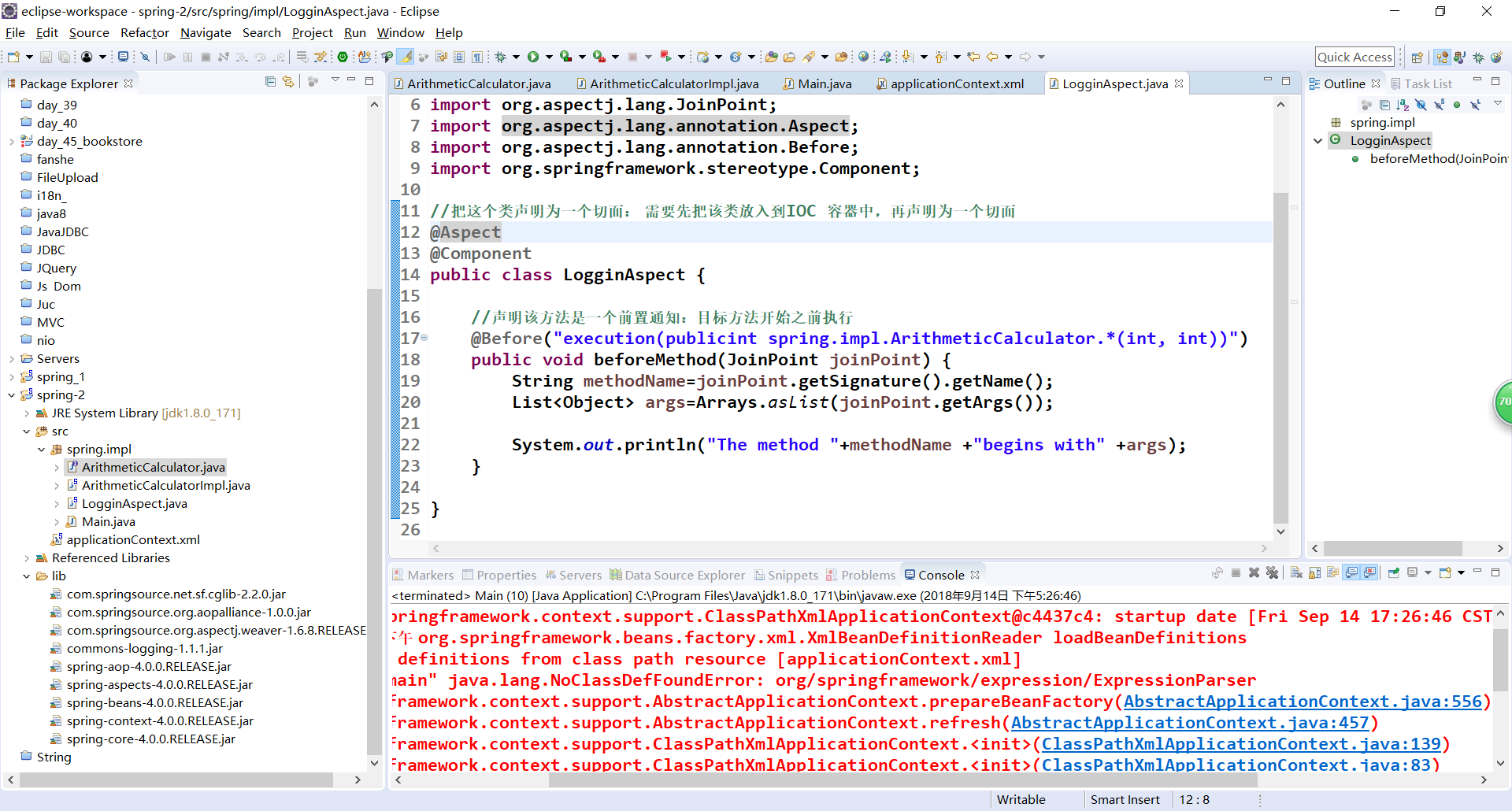
String methodName=joinPoint.getSignature().getName();

List<Object> args=Arrays.*asList*(joinPoint.getArgs());

System.***out***.println("The method "+methodName +"begins with" +args);

}

}



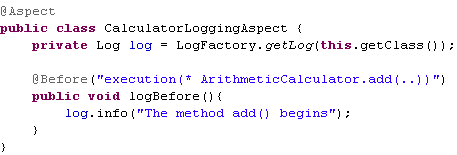
用 AspectJ 注解声明切面：

* **要在 Spring 中声明 AspectJ 切面, 只需要在 IOC 容器中将切面声明为 Bean 实例**. 当在 Spring IOC 容器中初始化 AspectJ 切面之后, Spring IOC 容器就会为那些与 AspectJ 切面相匹配的 Bean 创建代理.
* **在 AspectJ 注解中, 切面只是一个带有 @Aspect 注解的 Java 类**.
* **通知是标注有某种注解的简单的 Java 方法**.
* AspectJ 支持 5 种类型的通知注解:
  + **@Before:** 前置通知, 在方法执行之前执行
  + **@After:** 后置通知, 在方法执行之后执行
  + **@AfterRunning:** 返回通知, 在方法返回结果之后执行
  + **@AfterThrowing:** 异常通知, 在方法抛出异常之后
  + **@Around:** 环绕通知, 围绕着方法执行

前置通知：

* 前置通知:在方法执行之前执行的通知

前置通知使用 @Before 注解, 并将切入点表达式的值作为注解值.



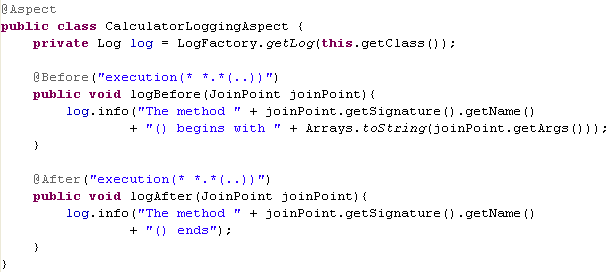
标识这个方法是个前置通知, 切点表达式表示**执行** ArithmeticCalculator

接口的 add() 方法. \* 代表匹配任意修饰符及任意返回值, 参数列表中的 ..

匹配任意数量的参数

后置通知：

* 后置通知是在连接点完成之后执行的, 即连接点返回结果或者抛出异常的时候, 下面的后置通知记录了方法的终止.
* 一个切面可以包括一个或者多个通知.

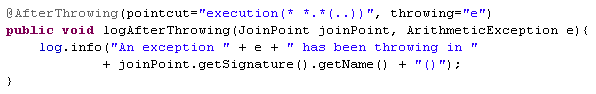


在返回通知中访问连接点的返回值：

* 在返回通知中, 只要将 **returning** 属性添加到 @AfterReturning 注解中, 就可以访问连接点的返回值. 该属性的值即为用来传入返回值的参数名称.
* 必须在通知方法的签名中添加一个**同名参数**. 在运行时, Spring AOP 会通过这个参数传递返回值.
* @AfterReturning(value="execution(\* spring.impl.\*.\*(..))",returning="result")
* **public** **void** afterReturning(JoinPoint joinPoint,Object result) {
* String methodName=joinPoint.getSignature().getName();
* System.***out***.println("The method "+methodName+" ends with "+result);
* }

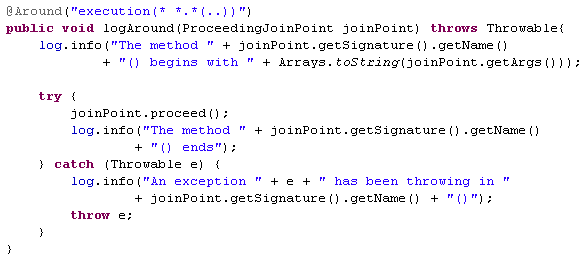
异常通知：

* 只在连接点抛出异常时才执行异常通知
* **将 throwing 属性添加到 @AfterThrowing 注解中**, 也可以访问连接点抛出的异常. Throwable 是所有错误和异常类的超类. 所以在异常通知方法可以捕获到任何错误和异常.
* **如果只对某种特殊的异常类型感兴趣**, 可以将参数声明为其他异常的参数类型. 然后通知就只在抛出这个类型及其子类的异常时才被执行.



环绕通知：

* 环绕通知是所有通知类型中功能最为强大的, 能够全面地控制连接点. 甚至**可以控制是否执行连接点**.
* **对于环绕通知来说, 连接点的参数类型必须是 ProceedingJoinPoint** . 它是 JoinPoint 的子接口, **允许控制何时执行, 是否执行连接点**.
* **在环绕通知中需要明确调用 ProceedingJoinPoint 的 proceed() 方法来执行被代理的方法. 如果忘记这样做就会导致通知被执行了, 但目标方法没有被执行**.
* **注意**: **环绕通知的方法需要返回目标方法执行之后的结果, 即调用 joinPoint.proceed(); 的返回值, 否则会出现空指针异常**



**指定切面的优先级：**

* 在同一个连接点上应用不止一个切面时, 除非明确指定, 否则它们的优先级是不确定的.
* 切面的优先级可以通过实现 Ordered 接口或利用 @Order 注解指定.
* 实现 Ordered 接口, getOrder() 方法的返回值越小, 优先级越高.
* 若使用 @Order 注解, 序号出现在注解中





使用JdbcTemplate:

需要先配置xml 文件

<!-- 自动扫描包-->

<context:component-scan base-package=*"spring"*/>

<!-- 导入资源文件 -->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 配置c3p0 数据源 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"user"* value=*"${jdbc.user}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

<property name=*"jdbcUrl"* value=*"${jdbc.jdbcUrl}"*></property>

<property name=*"driverClass"* value=*"${jdbc.driverClass}"*></property>

<property name=*"initialPoolSize"* value=*"${jdbc.initPoolSize}"*></property>

<property name=*"maxPoolSize"* value=*"${jdbc.maxPoolSize}"*></property>

</bean>

<!-- 配置 Spirng 的 JdbcTemplate -->

<bean class=*"org.springframework.jdbc.core.JdbcTemplate"* id=*"jdbcTemplate"*>

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

</bean>

1. 执行 INSERT, UPDATE, DELETE

jdbcTemplate.update(sql,"hyc","1");

1. jdbcTemplate.batchUpdate(sql, batchArgs);

执行批量更新: 批量的 INSERT, UPDATE, DELETE

最后一个参数是 Object[] 的 List 类型: 因为修改一条记录需要一个 Object 的数组,

那么多条不就需要多个 Object 的数组吗

1. 从数据库中获取一条记录, 实际得到对应的一个对象

注意不是调用 queryForObject(String sql, Class<Employee> requiredType, Object... args) 方法!

而需要调用 queryForObject(String sql, RowMapper<Employee> rowMapper, Object... args)

其中的 RowMapper 指定如何去映射结果集的行, 常用的实现类为 BeanPropertyRowMapper

使用 SQL 中列的别名完成列名和类的属性名的映射. 例如 last\_name lastName

不支持级联属性. JdbcTemplate 到底是一个 JDBC 的小工具, 而不是 ORM 框架

1. \* jdbcTemplate.query(sql, rowMapper,5);

查到实体类的集合

注意调用的不是 queryForList 方法

获取单个列的值, 或做统计查询

\* 使用 queryForObject(String sql, Class<Long> requiredType)

使用NamedParameterJdbcTemplate：

<!-- 配置NamedParameterJdbcTemplate, 该对象可以使用具名参数.

其没有无参数的构造器，所以必须为其构造器指定参数 -->

<bean id=*"namedParameterJdbcTemplate"*

class=*"org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"*>

<constructor-arg ref=*"dataSource"*></constructor-arg>

</bean>

使用具名参数时, 可以使用 update(String sql, SqlParameterSource paramSource) 方法进行更新操作

\* 1. SQL 语句中的参数名和类的属性一致!

\* 2. 使用 SqlParameterSource 的 BeanPropertySqlParameterSource 实现类作为参数.

事务：

在配置文件中：

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

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

</bean>

<!-- 启用事务 --> <!-- 在需要添加事务注解的函数上面加@Transacational -->

<tx:annotation-driven transaction-manager=*"transactionManager"*/>

在需要事务的函数前面添加事务注解：

//添加事务注解

@Transactional

//添加事务注解

//1.使用propagation 指定事务的传播行为，即当前的事务方法被另外一个事务方法调用时。

// 如何使用事务，默认取值为 REQUIRED ，即使用调用方法的事务。

//REQUIRES\_NEW: 事务自己的事务, 调用的事务方法的事务被挂起.

@Transactional(propagation=Propagation.REQUIRES\_NEW)。

* 当 bookService 的 purchase() 方法被另一个事务方法 checkout() 调用时, 它默认会在现有的事务内运行. 这个默认的传播行为就是 REQUIRED. 因此在 checkout() 方法的开始和终止边界内只有一个事务. 这个事务只在 checkout() 方法结束的时候被提交, 结果用户一本书都买不了
* 事务传播属性可以在 @Transactional 注解的 propagation 属性中定义。

* //2.使用 isolation 指定事务的隔离级别, 最常用的取值为 READ\_COMMITTED
* //3.默认情况下 Spring 的声明式事务对所有的运行时异常进行回滚.
* //也可以通过对应的属性进行设置. 通常情况下去默认值即可.
* //4.使用 readOnly 指定事务是否为只读. 表示这个事务只读取数据但不更新数据,
* //这样可以帮助数据库引擎优化事务. 若真的事一个只读取数据库值的方法, 应设置 readOnly=true
* //5.使用 timeout 指定强制回滚之前事务可以占用的时间.

* // @Transactional(propagation=Propagation.REQUIRES\_NEW,
* // isolation=Isolation.READ\_COMMITTED,
* // noRollbackFor={UserAccountException.class})
* //@Transactional(propagation=Propagation.REQUIRES\_NEW,
* // isolation=Isolation.READ\_COMMITTED,
* // readOnly=false,
* // timeout=3)