AspectJ Spring支持

AspectJ

- @AspectJ的风格类似纯java注解的普通java类
- Spring可以使用AspectJ来做切入点解析
- AOP的运行时仍旧是纯的Spring AOP，对AspectJ的编译器或者织入无依赖性

对@AspectJ支持可以使用XML或Java风格的配置

确保AspectJ的aspectjweaver.jar库包含在应用程序（版本1.6.8或更高版本）的classpath中

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

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

@AspectJ切面使用@Aspect注解配置，拥有@Aspect的任何bean被Spring自动识别并应用

用**@Apsect注解的类可以有方法和字段**，他们也可能包括切入点（pointcut），通知（Advice）和引用（introduction）声明

@Apsect注解是**不能够通过类路径自动检测**发现的，**所以需要配合使用@Component注释或者在XML配置Bean**

一个类中的@Aspect注解标识它为一个切面，并且将自己从自动代理中排除

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of aspect here as normal -->
</bean>
```

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

因为@Aspect是可以代理的，所以说，标识为切面中之后，就把自己摘出来，不然就出现死循环

pointcut

一个切入点通过一个普通的方法定义来提供，并且切入点表达式使用@Pointcut注解，方法返回类型必须为void

定义一个名为'anyOldTransfer'，这个切入点将匹配任何名为“transfer”的方法的执行

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

execution	匹配方法执行的连接点
within	限定匹配特定类型的连接点
this	匹配特定连接点的bean引用是指定类型的实例的限制
target	限定匹配特定连接点的目标对象是指定类型的实例
args	限定匹配特定连接点的参数是给定类型的实例
@target	限定匹配特定连接点的类执行对象的具有给定类型的注解
@args	限定匹配特定连接点实际传入参数的类型具有给定类型的注解
@within	限定匹配到内具有给定的注释类型的连接点

组合pointcut

切入点表达式可以通过&&、||和!进行组合，也可以通过名字引用切入点表达式

通过组合，可以建立更加复杂的切入点表达式

```
@Pointcut("execution(public * (..))")
    private void anyPublicOperation() {}

    @Pointcut("within(com.xyz.someapp.trading..)")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

定义一个良好的切入点

- **AspectJ是编译期的AOP**
- **检查代码并匹配连接点与切入点的代价是昂贵的**
- **一个好的切入点应该包括一下几点**
  - **选择特定类型的连接点，如： execution, get, set, call, handler**
  - **确定连接点范围，如： within, withincode**
  - **匹配上下文信息，如： this, target, @annotation**

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class MoocAspect {

    @Before("execution(* com.imooc.aop.aspectj.biz.*Biz.*(..))")
    public void before() {
        //..
    }

}
```

这是**@Before advice**

下面是示例场景
代码如下：

```java
//这是切入点

@Pointcut("execution(* aop.aspectj.biz.*Biz.*(..))")
public void pointcut() {}

@Pointcut("within( aop.aspectj.biz.*)")
public void bizPointcut() {}

@Before("pointcut()")
public void before() {
    System.out.println("Before.");
}
```

```java
ApplicationContext context=new
ClassPathXmlApplicationContext("spring-aop-aspectj.xml");
MoocBiz moocBiz= (MoocBiz) context.getBean("moocBiz");
moocBiz.save("This is test");
```

这样好像是可以的，

这个是afterRunturing的使用，
如下，代码：

```java
@AfterReturning(pointcut ="execution(* aop.aspectj.biz.*Biz.*(..))()",returning = "r
eturnValue")
public  void afterReturning(Object returnValue){
    System.out.println("afterReturning MoocAspect : "+returnValue);
```

```
}
```

Before.
MoocBiz save : This is test
afterReturning MoocAspect : Save success!
这是输出结果

如果我们想知道异常是什么的时候，就可以用AfterThrowing绑定异常
下面贴代码：

```
public String save(String arg) {
      System.out.println("MoocBiz save : " + arg);
      throw  new RuntimeException("Save failed!");
//     return " Save success!";
}
```

```
@AfterThrowing(pointcut = "execution(* aop.aspectj.biz.*Biz.*(..))()",throwing = "e"
)
public  void afterthrowing(RuntimeException e){
   System.out.println("MoocAspect  afterthrowing :"+e);

}
```

最终通知：

After (finally) advice

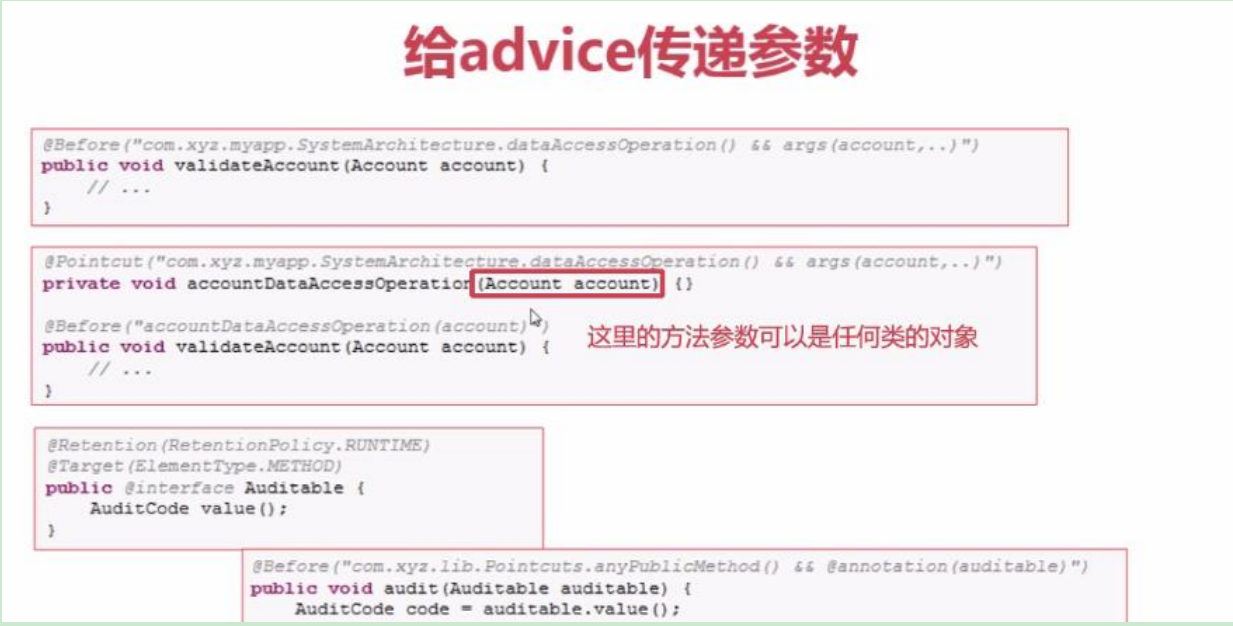
Around advice
这上面的proceed方法执行后的返回值问题
这个时候是真正我们执行方法的时候
无论这个方法的返回值是不是真的返回值
都是用这种方式来处理，在proceed这个地方它是不知道是不是有返回值 ，所以采用了一种通用的方式，是Objcet，把真正方法的返回值来return返回， 这是扩展**void不是没有返回值，只是是一种特殊类型的**

这个在proceed之前与之后，我们都可以做一些我们自己的操作， 写自己的代码，这就是around的特点
下面贴代码：

```
@Around("execution(* aop.aspectj.biz.*Biz.*(..))()")
public  Object around(ProceedingJoinPoint pjp) throws Throwable{
   System.out.println("Around 1");
Object obj=pjp.proceed();
System.out.println("Around 2");
System.out.println("Around "+obj);
   return obj;
}
```

下面是结果：
Before.
Around 1
MoocBiz save : This is test
after
Around 2
Around Save success!
afterReturning MoocAspect : Save success!
JoinPoint(连接点): 它定义在哪里(哪些点)加入你的逻辑功能，对于Spring AOP，Jointpoint指的就是Method.
PointCut(切入点的集合)：即一组Joinpoint，（通过正则表达式去匹配）就是说一个Advice可能在多个地方织入。
例如： @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
public void businessService() {}
返回值是任何类型，com.xyz.someapp.service下面的任何类的任何方法
Aspect(切面)： 实际是Advice和Pointcut的组合，但是Spring AOP 中的Advisor也是这样一个东西，但是Spring中为什么叫Advisor而不叫做Aspect。
Advice(通知)：所谓通知是指拦截到jointpoint之后所要做的事情就是通知即特定的Jointpoint处运行的代码。
对于Spring AOP 来讲，通知分为前置通知(Before advice)、后置通知(AfterreturningAdvice)、
异常通知(ThrowAdvice)、最终通知(AfterThrowing)、环绕通知(AroundAdvice)。

Target(目标对象):代理的目标对象即被通知的对象。
Weave(织入): 指将aspects应用到target对象并导致proxy对象创建的过程称为织入
Introducton(引入)：在不修改类代码的前提下，Introduction可以在运行期为类动态地添加一些方法或Field 。



这通常都是来获取相应的参数，做一些判断，或者做一些儿日志的记录

下面贴代码：

```
@Pointcut("execution(* com.imooc.aop.aspectj.biz.*Biz.*(..))")
public void pointcut() {}

@Pointcut("within(com.imooc.aop.aspectj.biz.*)")
public void bizPointcut() {}

@Before("pointcut()")
public void before() {
   System.out.println("Before.");
}


@Before("pointcut()&&args(arg)")
```

```
public void beforeWithParam(String arg) {
   System.out.println("beforeWithParam."+arg);
}
@Before("pointcut()&& @annotation(moocMethod)")
public void beforeWithAnnotation(MoocMethod moocMethod) {
   System.out.println("beforeWithAnnotation."+moocMethod.value());
}
```

这是俩种方式，一种是引用注解，一种是直接参数传值

```
@MoocMethod("moocBiz save with MoocMethod")
public String save(String arg) {
      System.out.println("MoocBiz save : " + arg);
//    throw new RuntimeException(" Save failed!");
return " Save success!";
}
```

引用注解的这个需要自己在切入点的方法声明注解，还需要上面这样在方法上声明

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MoocMethod{
     String value();
}
```

因为我们使用的是自定义注解，所以在这里我们要自定义。
而直接声明的就是获取到我们需要传参的数，也就是我们执行的方法的参数

当然，我们也可以直接在切入点这里的方法使用添加这个注解，下面贴代码

```
@Pointcut("execution(* com.imooc.aop.aspectj.biz.*Biz.*(..))&&@annotation(moocMethod
)")
public void pointcut() {}
```

```
@Before("pointcut(moocMethod) ")
public void beforeWithAnnotation(MoocMethod moocMethod) {
   System.out.println("beforeWithAnnotation."+moocMethod.value());
}
```

这个就是 这个需要注意版本

Adivce的参数与泛型

- **Spring AOP可以处理泛型类的声明和使用方法的参数**

```
public interface Sample<T> {
    void sampleGenericMethod(T param);
    void sampleGenericCollectionMethod(Collection>T> param);
}
```

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
    // Advice implementation
}
```

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
    // Advice implementation
}
```

Advice参数名称

- 通知和切入点注解有一个额外的“argNames”属性，它可以用来指定所注解的方法的参数名

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
        argNames="bean,auditable")
public void audit(Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code and bean
}
```

- 如果第一参数是JoinPoint, ProceedingJoinPoint, JoinPoint.StaticPart，那么可以忽略它

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
        argNames="bean,auditable")
public void audit(JoinPoint jp, Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code, bean, and jp
}
```

Introductions

- 允许一个切面声明一个通知对象实现指定接口，并且提供了一个接口实现类来代表这些对象
- introduction使用@DeclareParents进行注解，这个注解用来定义匹配的类型拥有一个新的parent

这个在以前的笔记中实现过，是XML的方式

与在XML中实现方式有点区别

例如：给定一个接口UsageTracked，并且该接口拥有DefaultUsageTracked的实现，接下来的切面声明了所有的service接口的实现都实现了UsageTracked接口

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xzy.myapp.service.*+", defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("com.xyz.myapp.SystemArchitecture.businessService() && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }

}
```

这就是注解的方式形式

切面实例化模型

- 这是一个高级主题
- “perthis”切面通过指定@Aspect注解perthis子句实现
- 每个独立的service对象执行时都会创建一个切面实例
- service对象的每个方法在第一次执行的时候创建切面实例，切面在service对象失效的同时失效。

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

    private int someState;

    @Before(com.xyz.myapp.SystemArchitecture.businessService())
    public void recordServiceUsage() {
        // ...
    }
```

@Aspect就是声明一个切面类

重点如下：
声明是AOP及实现方式
AOP的基本概念
Spring中的AOP
Schema-based AOP
Spinrg AOP API
Aspectj
http://sishuok.com/forum/posts/list/281.html号称比较全的AspectJ使用说明

第一次复习

```
@Component
@Aspect
public class AspectJTest02 {


@Pointcut("execution(* com.Spring.Aspectj.*.*(..))")
public  void pointcut(){}



@Before("execution(* com.Spring.Aspectj.*.*(..))")
public  void before(){
System.out.println("before()...");
    }



@After(value = "pointcut()")
public  void  aVoid( ) {
System.out.println("after...");
  }


@AfterReturning( pointcut ="pointcut()",returning = "returnvalue")
public  void  afterReturning(Object returnvalue){
System.out.println("afterreturning是 :"+returnvalue);
    }


@AfterThrowing(pointcut = "pointcut()",throwing = "e")
public  void afterthowing (RuntimeException e){
System.out.println("afterthowing是: "+e);
```

```
    }



@Around("pointcut()")
public  Object around(ProceedingJoinPoint pjp) throws Throwable{
System.out.println("环绕开始 ");
Object o = pjp.proceed();
System.out.println("返回环绕的东西"+o);
System.out.println("环绕结束了");

return  o;
   }


@Before("within( com.Spring.Aspectj.Demo.*)")
public  void  before2(){
System.out.println("为Demo专门用的 before");
    }

@Before( "pointcut()&& args(arg)")
public  void before3(String arg){
System.out.println("beofe3的pointcut()&& args(arg)中的"+arg);
    }


@Before("pointcut()&& @annotation(moocMethod)")
public void beforeWithAnnotation(MoocMethod moocMethod) {
System.out.println("beofe3的pointcut()&& args(arg)中的"+moocMethod.value());
    }




}
```