www.neuedu.com

Neuedu

Spring框架技术

——面向切面编程（AOP）

V1.1

本章内容

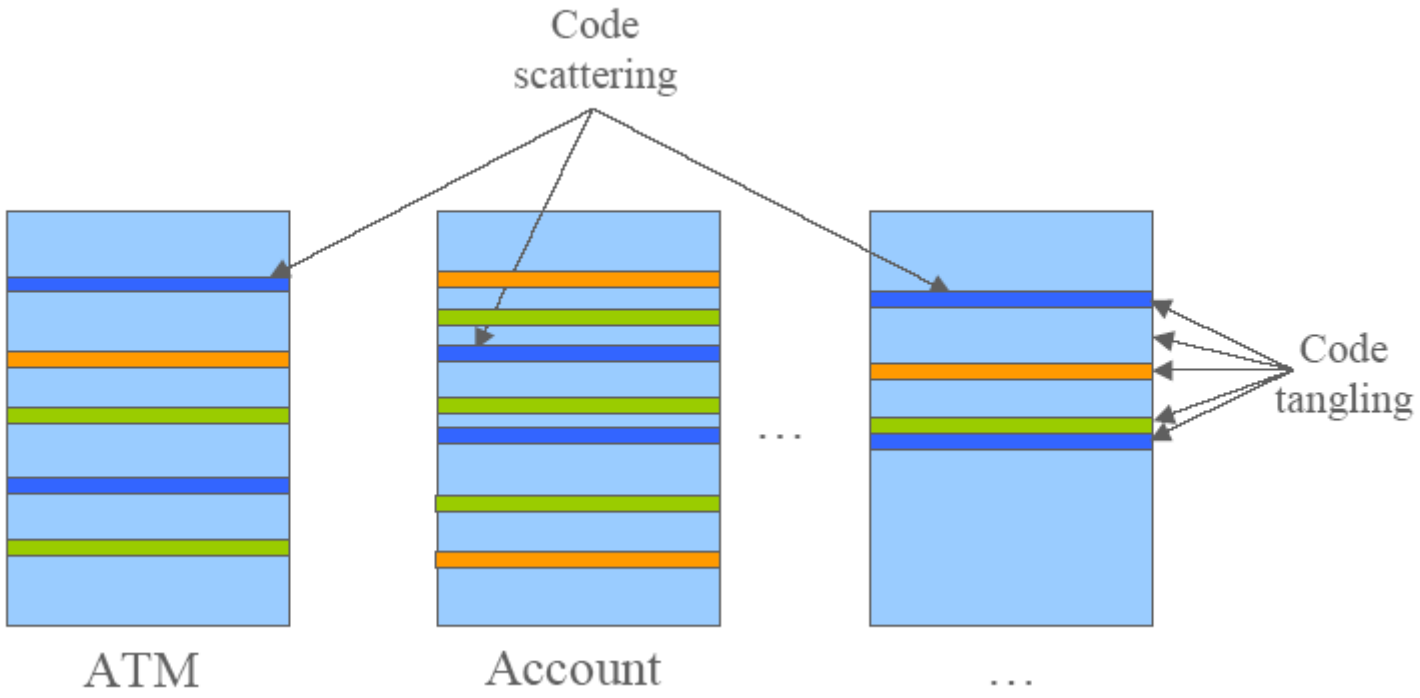
Neuedu

节	知识点	掌握程度	难易程度
AOP简介	AOP是什么	理解	难
	比较使用AOP和不使用AOP	理解	
	为何使用AOP	理解	
	AOP应用范围	了解	
核心概念	核心概念	理解	难
代理机制	静态代理	了解	
	动态代理	掌握	难
	基于注解方式的AOP编程	掌握	难
	基于配置方式的AOP编程	掌握	难

AOP是什么

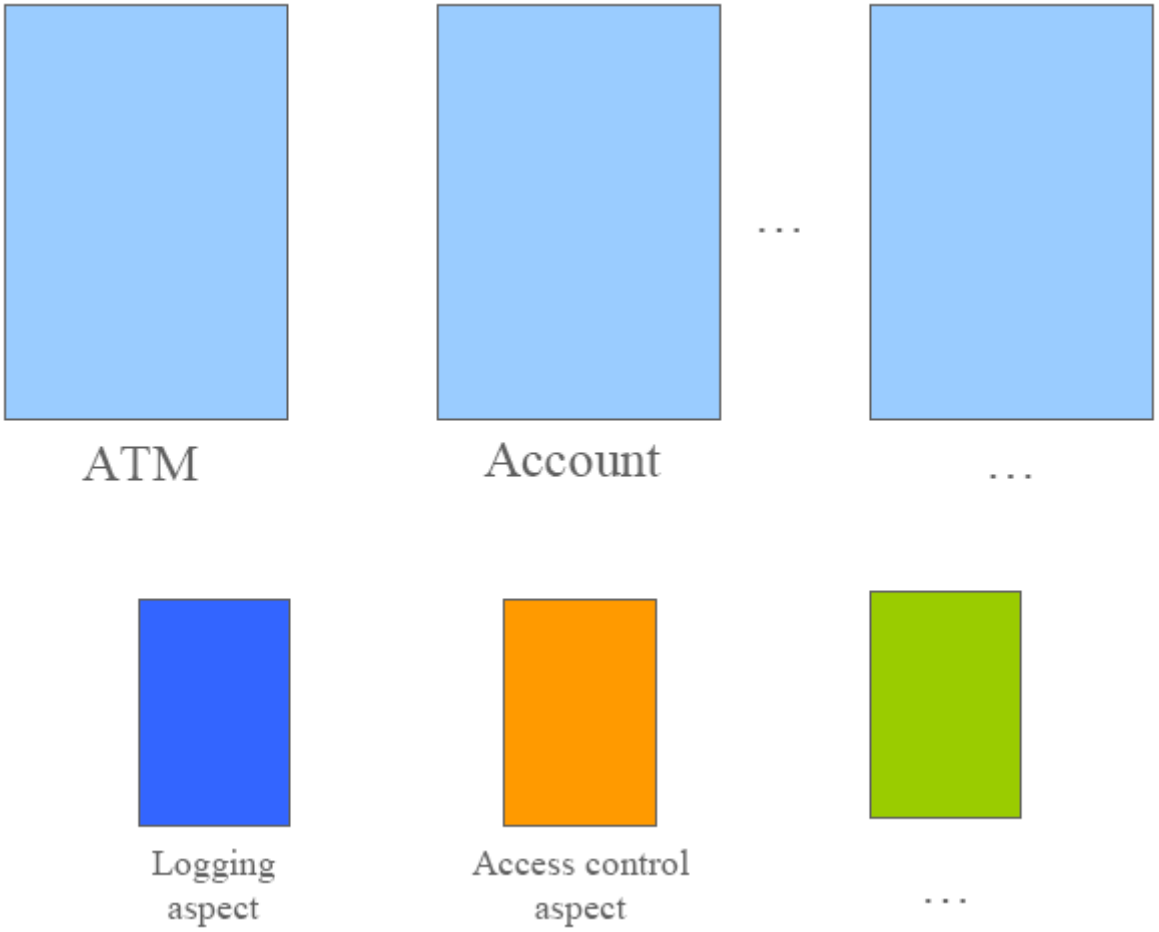
- 面向方面编程（Aspect Oriented Programming）
- Spring的一个关键的组件就是 *AOP框架*。 尽管如此，Spring IoC容器并不依赖于AOP，这意味着可以自由选择是否使用AOP，AOP提供强大的中间件解决方案，这使得Spring IoC容器更加完善。
- Spring AOP的出现是**为了取代 EJB中的事务机制，它有这种声明式的事务机制**，其实AOP的这种思想早就已经有了，并不是一种什么新的技术，也并不是说专门由java这里来实现的。
- 面向切面编程（AOP）提供另外一种角度来思考程序结构，通过这种方式弥补了面向对象编程（OOP）的不足。

比较：Without AOP



比较：With AOP

Neuedu



为何使用AOP

Neuedu

- 高度模块化，使得我们的系统更易实现和更易维护
- 使每个模块承担的责任更清晰，提高代码的可追踪性
- 解决设计时两难的局面，在不需改动原先代码的情况下推迟不必要的需求的实现
- 提高代码的重用性
- 加速系统的开发和部署，提高程序员的开发效率
- 降低系统开发的成本

AOP应用范围

Neuedu

- Persistence（持久化）
- Transaction management（事务管理）
- Security（安全）
- Logging, tracing, profiling and monitoring（日志，跟踪，优化，监控）

- Debugging（调试）
- Authentication（认证）
- Context passing（上下文传递）
- Error/Exception handling（错误/异常处理）

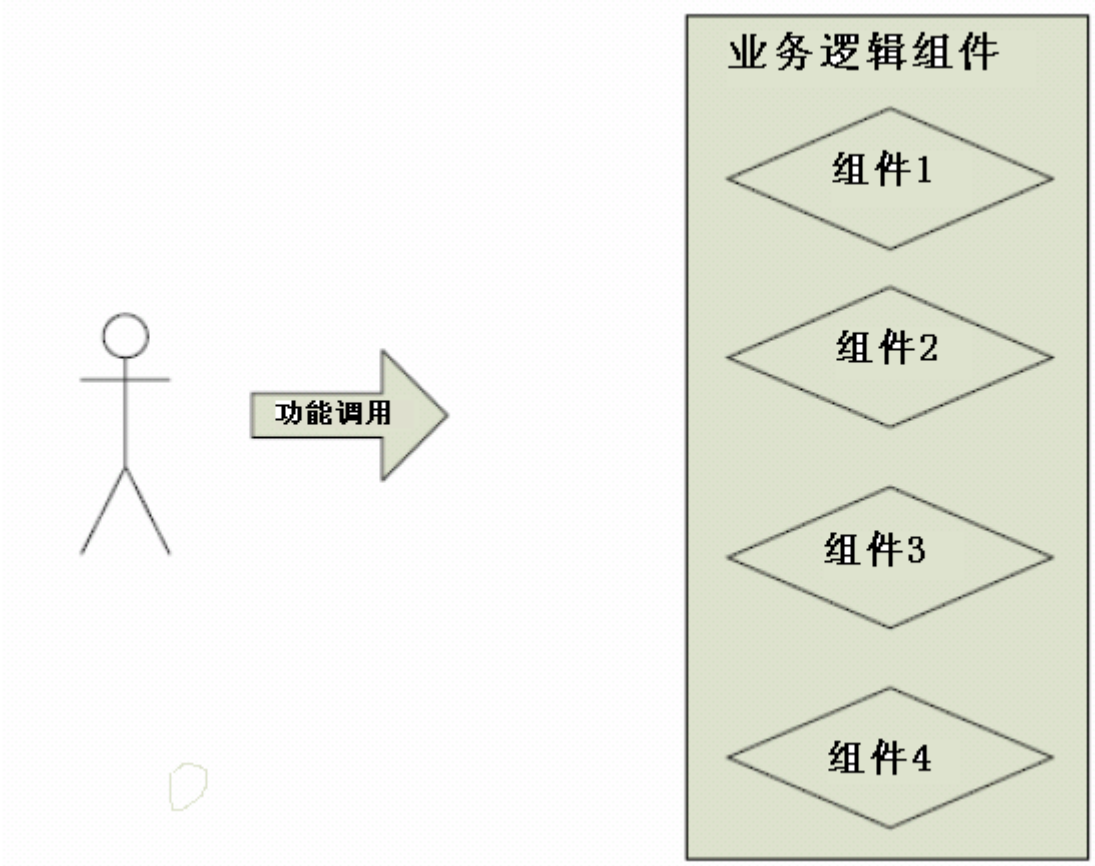
- Lazy loading（懒加载）
- Performance optimization（性能优化）
- Resource pooling（资源池）
- Synchronization（同步）

AOP涉及到的概念

Neuedu

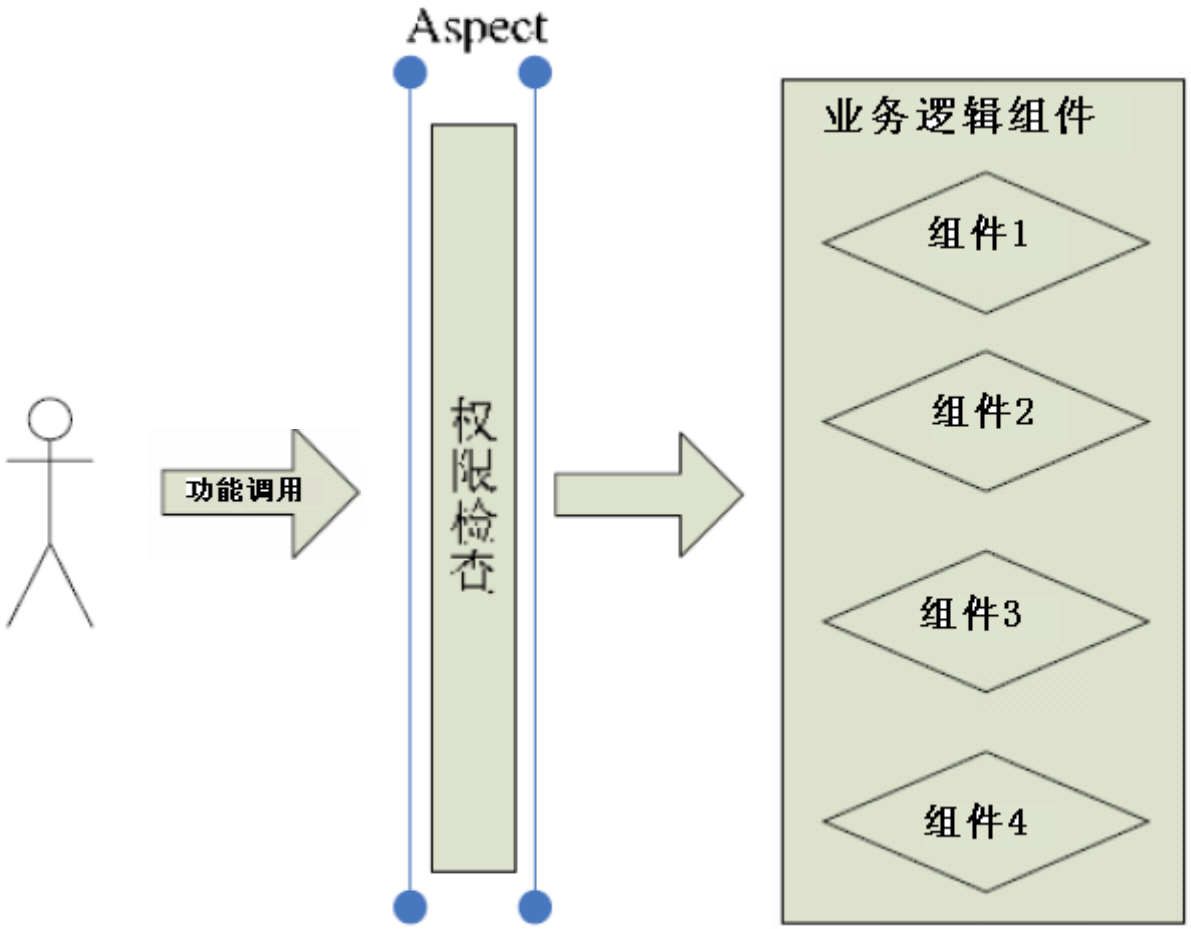
- Aspect(切面):指横切性关注点的抽象即为切面,它与类相似,只是两者的关注点不一样,类是对物体特征的抽象,而切面是对横切性关注点的抽象.
- joinpoint(连接点):所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点,实际上joinpoint还可以是field或类构造器)
- Pointcut(切入点):所谓切入点是指我们要对哪些joinpoint进行拦截的定义.
- Advice(通知):所谓通知是指拦截到joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知
- Target(目标对象):代理的目标对象
- 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.
- Introduction(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.

切面（Aspect）

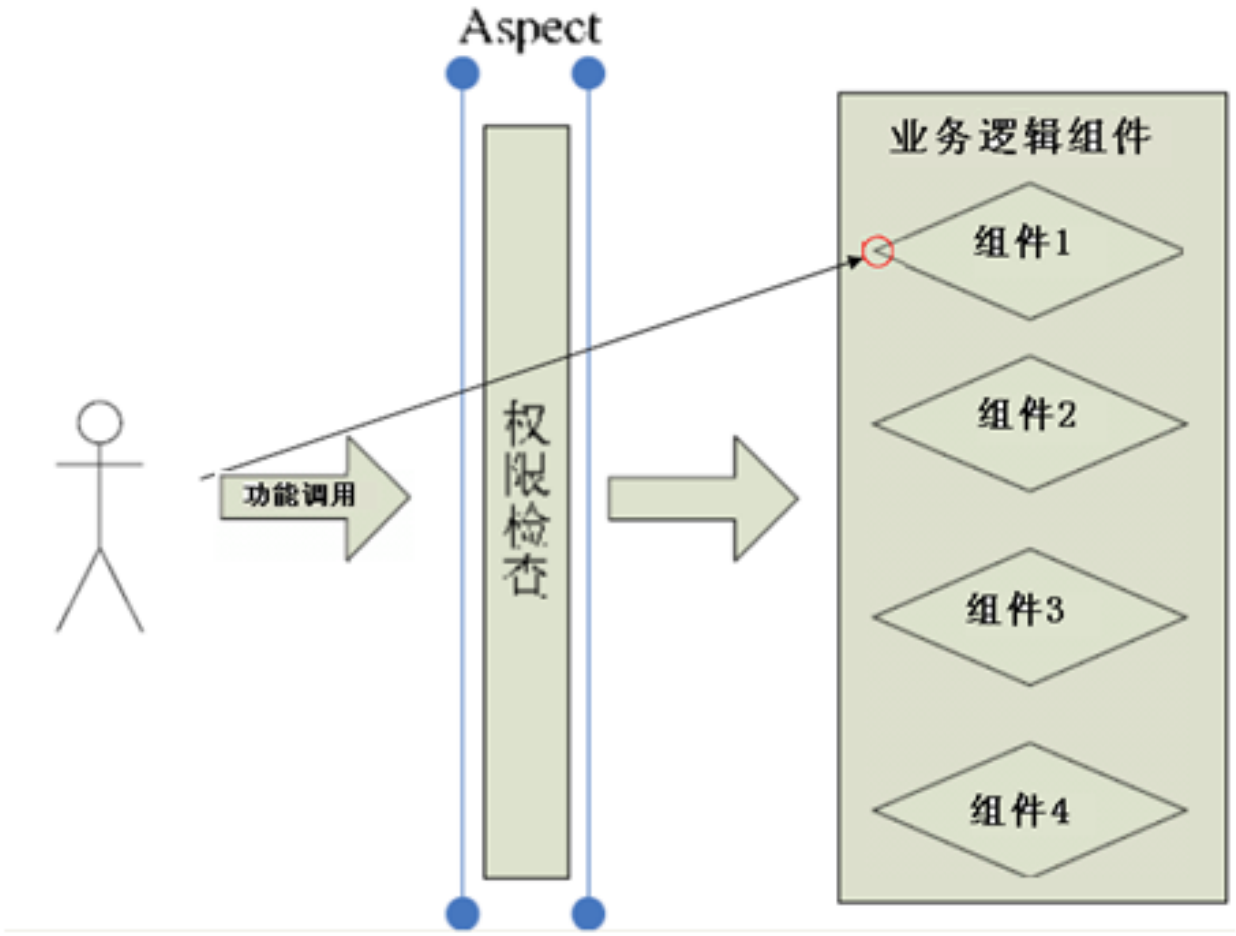


切面（Aspect）

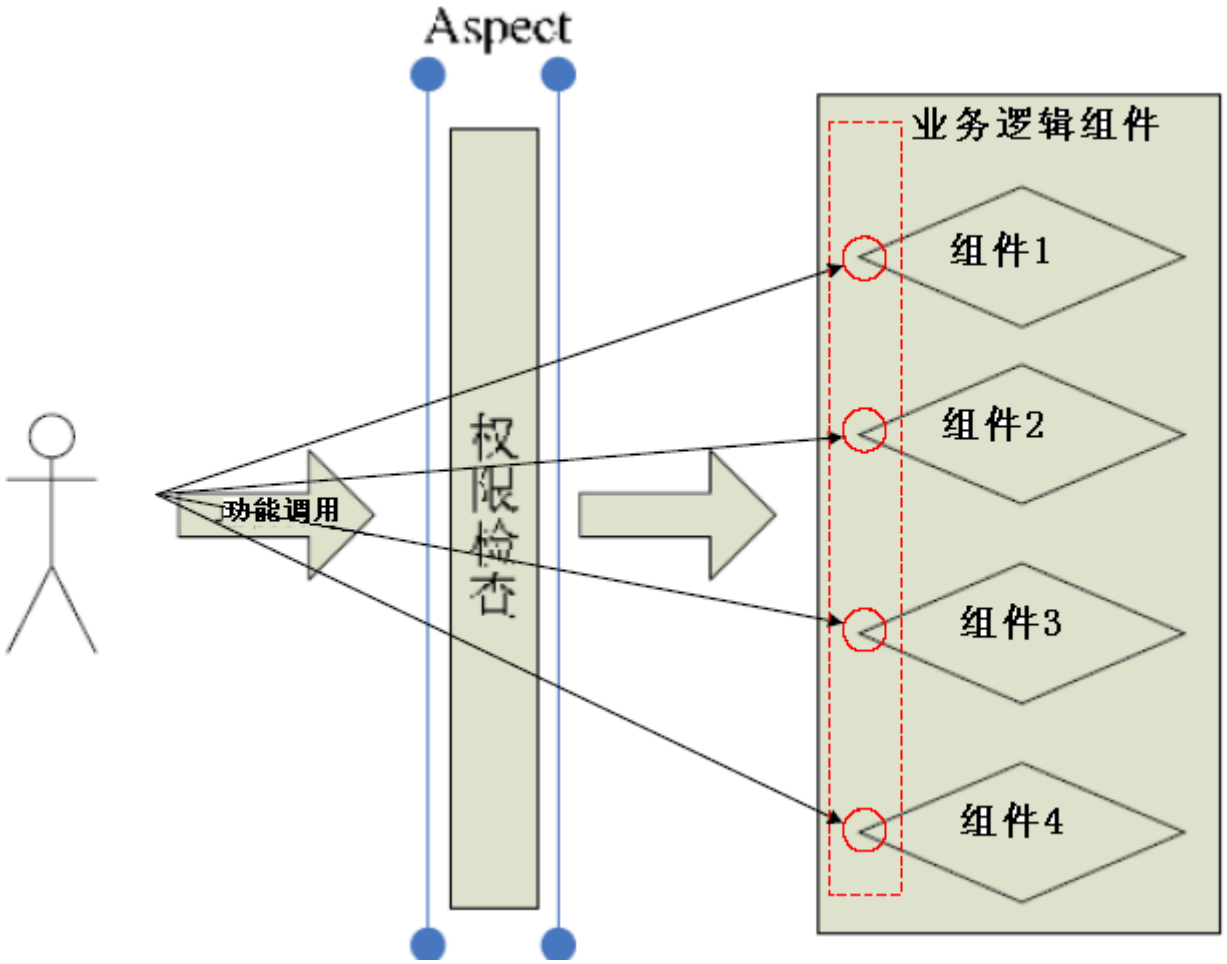
Neuedu



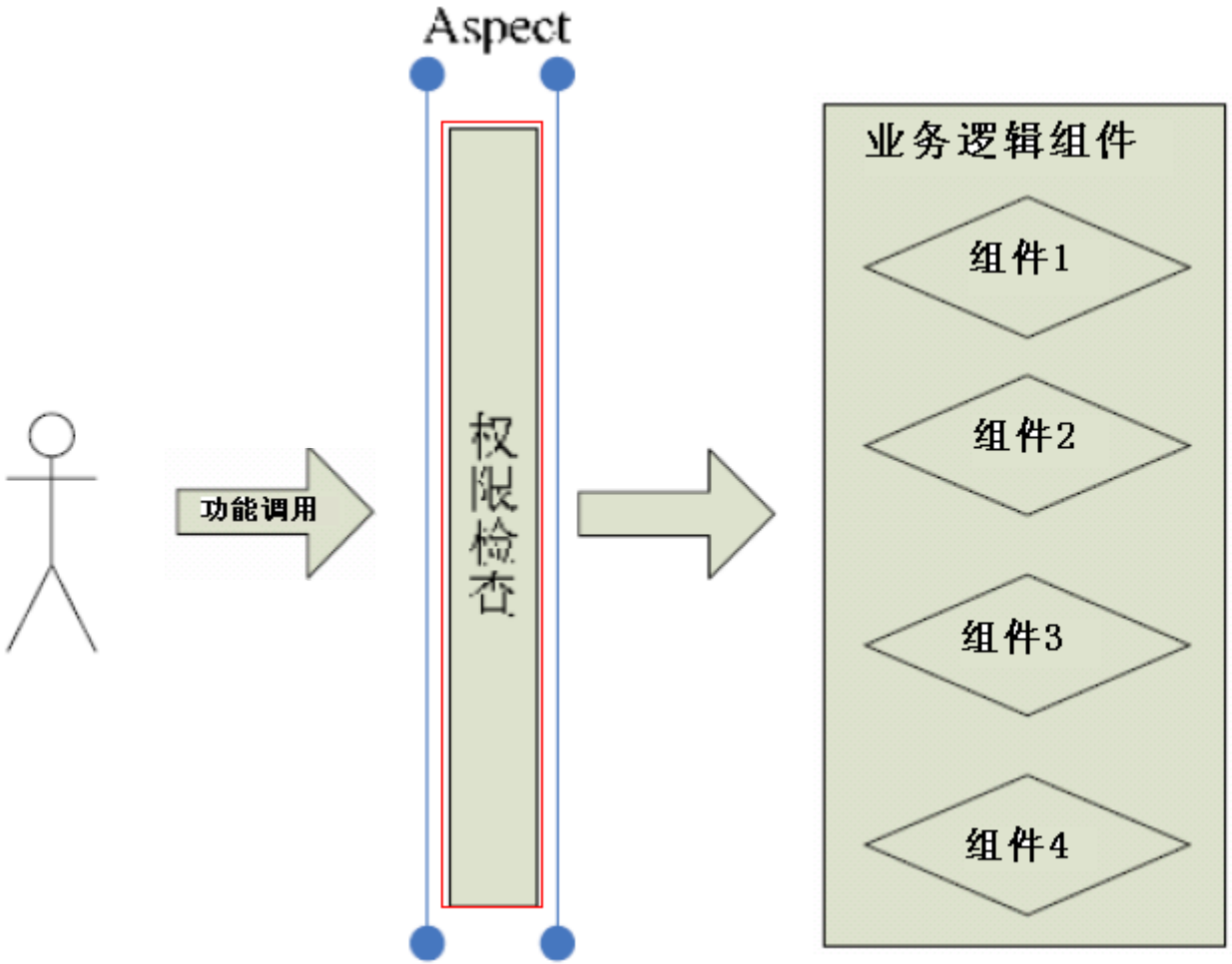
连接点（JoinPoint）



切点（PointCut）

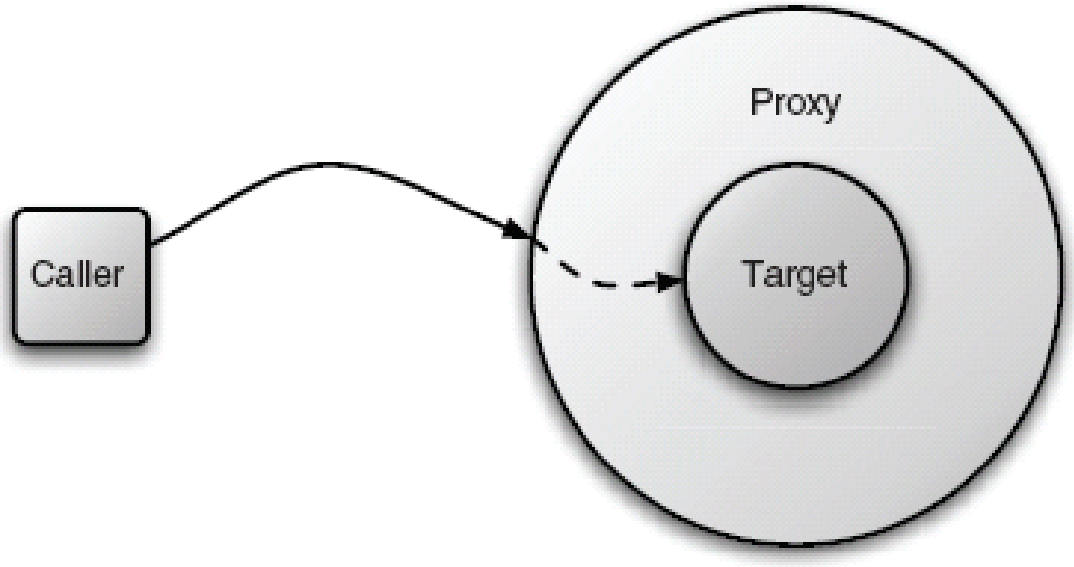


处理逻辑/通知（Advice）



目标对象

Neuedu



通知的类型

Neuedu

- 前置通知（Before advice）：在某连接点之前执行的逻辑，但这个逻辑不能阻止连接点的执行。
- 后置通知（After advice）：当某连接点退出的时候执行的逻辑。
- 环绕通知（Around advice）：包围连接点退出的时候执行的逻辑。
- 抛出异常后逻辑（After throwing advice):在方法抛出异常退出时执行的逻辑。

AOP代理

- Spring缺省使用JAVA 动态代理（*dynamic proxies*）来作为AOP的代理。这样任何接口都可以被代理。
- Spring也支持使用CGLIB代理. 对于需要代理类而不是代理接口的时候CGLIB代理是很有必要的。 如果一个业务对象并没有实现一个接口，默认就会使用CGLIB。作为面向接口编程的最佳实践，业务对象通常都会实现一个或多个接口。

AOP代理-静态代理

Neuedu

- 代理分类为：静态代理、动态代理。
  - 所有的静态代理，代理类是你确确实实能看到的；
  - 而动态的，是在运行期生成的，所以这两种方式我们都需要来领会。

- 示例：spring_static_proxy工程

AOP代理-静态代理

Neuedu

```
public class UserManagerImplProxy implements

UserManagerIface {
    private UserManagerIface userManager;
    public UserManagerImplProxy(UserManagerIface
    userManager){
       this.userManager = userManager;
     }
     ……
}
```

AOP代理-动态代理

Neuedu

- JDK动态代理：
  - Spring的AOP的默认实现就是采用jdk的动态代理机制实现的。
  - 通过之前的分析，我们要把横切性的关注点（例如安全性检查）单独的提取出来，这就是动态代理的思想。
  - 把散布在程序各个角落的关注点提取出来，进行模块化。
  - 模块化的好处：即模块化之后，我们只要单独维护这个模块就可以了，不用去维护散布在各处的关注点，否则，难度大，效率低。
  - AOP技术应该是OO的在技术上的补充，

示例：spring_dynamic_proxy工程

AOP代理-动态代理

Neuedu

- AOP通过动态代理技术在运行期织入增强代码，首先了解下AOP使用的两种代理机制：
  - 基于JDK的动态代理
  - 基于CGlib的动态代理
- JDK动态代理主要涉及两个类，
  - Java.lang.reflect.Proxy
  - Java.lang.reflect.InvocationHandler
- InvocationHandler是一个接口，可以通过实现该接口定义的横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑和业务逻辑编织在一起。
- Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

JDK动态代理

```
public class JDKProxy implements InvocationHandler {
    private Object targetObject;//代理的目标对象
    public Object createProxyInstance(Object targetObject){
          this.targetObject = targetObject;
          /*
          * 第一个参数设置代码使用的类装载器,一般采用跟目标类相同的类装载器
          * 第二个参数设置代理类实现的接口
          * 第三个参数设置回调对象,当代理对象的方法被调用时,会委派给该参数指定
            对象的invoke方法
          */
    return
    Proxy.newProxyInstance(this.targetObject.getClass().getClassLoader(),
                           this.targetObject.getClass().getInterfaces(),
    this);
    }
    public Object invoke(Object proxy, Method method, Object[] args)
                  throws Throwable {
          return method.invoke(this.targetObject, args);//把方法调用委派给目标对象
    }
}
```

当目标类实现了接口，我们可以使用jdk的Proxy来生成代理对象。

基于CGlib的动态代理

Neuedu

- JDK只能为接口创建代理实例，对于那些没有通过接口定义业务方法的类，可以通过CGLib创建代理实例。
- CGLib采用底层字节码技术，可以为一个类创建子类，并在子类中采用方法拦截技术拦截所有父类方法的调用，这时可以顺势织入横切逻辑。

- 示例：spring_aop04_CGLIB工程

基于CGlib的动态代理

```
public class CGLIBProxy implements MethodInterceptor {
    private Object targetObject;//代理的目标对象
    public Object createProxyInstance(Object targetObject){
          this.targetObject = targetObject;
          Enhancer enhancer = new Enhancer();//该类用于生成代理对象
          enhancer.setSuperclass(this.targetObject.getClass());//设置父类
          enhancer.setCallback(this);//设置回调对象为本身
          return enhancer.create();
    }
    public Object intercept(Object proxy, Method method, Object[] args,
                    MethodProxy methodProxy) throws Throwable {
          return methodProxy.invoke(this.targetObject, args);
    }
}
```

CGLIB可以生成目标类的子类，并重写父类非final修饰符的方法。

AOP编程

Neuedu

- Spring提供了两种切面声明方式，实际工作中我们可以选用其中一种：
  - 基于XML配置方式声明切面。
  - 基于注解方式声明切面。
- 要进行AOP编程，首先我们要在spring的配置文件中引入aop命名空间：

```
<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/beans
           http://www.springframework.org/schema/beans/spring-beans-
   2.5.xsd
           http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
</beans>
```

基于注解方式的AOP编程

Neuedu

- 采用注解方式实现(Annotation)步骤:
  - 采用Aspect定义切面
  - 在Aspect定义Pointcut和Advice
  - 启用AspectJ对源数据注解的支持（添加sechma)
  - 将Aspect类和目标对象配置到IOC容器中

- 示例：spring_aop01_annotation工程

基于注解方式的AOP编程

Neuedu

```
//定义一个切面
@Aspect
public class TestAdvic {
//定义一个切入点，名称是addMethod(),此方法不能有参数和返
   回值
//表达式描述哪些对象的哪些方法执行 advice
@Pointcut("execution(* add*(..))")
public void addMethod(){}
//定义advice ,指定在哪个切入织入此方法
//@Around("addMethod()")
@Before("addMethod()")
public void check(){
System.out.println("验证用户");
}
```

基于注解方式的AOP编程

Neuedu

- 任意公共方法的执行：
  - execution(public * *(..))
- 任何一个以“set”开始的方法的执行：
  - execution(* set*(..))
- AccountService 接口的任意方法的执行：
  - execution(* com.AccountService.*(..))
- 定义在service包里的任意方法的执行：
  - execution(* com.*.*(..))
- 定义在service包或者子包里的任意方法的执行：
  - execution(* com..*.*(..))

基于注解方式的AOP编程

- 启动对@AspectJ注解的支持(蓝色部分)：

```
<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/beans
           http://www.springframework.org/schema/beans/spring-beans-
   2.5.xsd
           http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    <aop:aspectj-autoproxy/>
   <bean id="orderservice" class="com.ttc.test.service.UserServiceBean"/>
   <bean id="log" class="com.ttc.test.service.LogPrint"/>
</beans>
```

基于注解方式的AOP编程

样例代码：

```
@Aspect
public class LogPrint {
    @Pointcut("execution(* com.ttc.test.service..*.*(..))")
    private void anyMethod() {}//声明一个切入点
    @Before("anyMethod() && args(userName)")//定义前置通知
    public void doAccessCheck(String userName) {
    }
    @AfterReturning(pointcut="anyMethod()",returning="revalue")//定义后置通知
    public void doReturnCheck(String revalue) {
    }
    @AfterThrowing(pointcut="anyMethod()", throwing="ex")//定义例外通知
     public void doExceptionAction(Exception ex) {
    }
    @After("anyMethod()")//定义最终通知
    public void doReleaseAction() {
    }
    @Around("anyMethod()")//环绕通知
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
          return pjp.proceed();
    }
}
```

基于配置方式的AOP编程

Neuedu

- 采用静态配置文件实现步骤
  - 定义一个切面类，编写处理逻辑（通知）
  - 确定连接点或切点：对象及方法
  - 确定处理逻辑（通知）调用模式
  - 配置

- 示例： spring_aop02_staticConfigFile工程

基于配置方式的AOP编程

Neuedu

- 采用静态配置文件实现
- <aop:config>
- <!-- 定义一个切面,并指定通知类 -->
- <aop:aspect id="tasp" ref="tadvic">
- <!-- 定义一个切入点，定义切入点名和匹配表达式 -->
- <aop:pointcut id="allMethod" expression="execution(* *.add*(..))"/>
- <!-- 定义切入点执行方法 -->
- <aop:after method="check" pointcut-ref="allMethod"/>
- </aop:aspect>
- </aop:config>
- <bean id="deptDao" class="com.qhit.DeptDao"></bean>
- <bean id="tadvic" class="com.qhit.TestAdvic"></bean>

ProceedingJoinPoint

Neuedu

- 使用JoinPoint可以拿到被拦截方法的参数和方法名:

```
public Object check(ProceedingJoinPoint  jp) throws
  Throwable{
//if(jp.getArgs()[0].equals("aaaa"))
//return null;
//else
System.out.println("验证用户");
//return jp.proceed();
return null;
}
```

- 示例： spring_aop03_joinpoint工程

基于配置方式的AOP编程

Neuedu

```
public class LogPrint {
    public void doAccessCheck() {}定义前置通知
    public void doReturnCheck() {}定义后置通知
     public void doExceptionAction() {}定义例外通知
    public void doReleaseAction() {}定义最终通知
    public Object doBasicProfiling(ProceedingJoinPoint pjp)
    throws Throwable {
         return pjp.proceed();环绕通知
    }
}
```

基于配置方式的AOP编程

```
<bean id="orderservice" class="com.ttc.test.service.OrderServiceBean"/>
<bean id="log" class="com.ttc.test.service.LogPrint"/>
<aop:config>
  <aop:aspect id="myaop" ref="log">
    <aop:pointcut id="mycut" expression="execution(*
    com.ttc.test.service..*.*(..))"/>
    <aop:before pointcut-ref="mycut" method="doAccessCheck"/>
    <aop:after-returning pointcut-ref="mycut" method="doReturnCheck "/>
    <aop:after-throwing pointcut-ref="mycut" method="doExceptionAction"/>
    <aop:after pointcut-ref="mycut" method= “doReleaseAction"/>
    <aop:around pointcut-ref="mycut" method="doBasicProfiling"/>
  </aop:aspect>
</aop:config>
```

本章重点总结

Neuedu

- 理解AOP的相关概念
- 了解动态代理机制
- 掌握AOP编程
  - 注解方式
  - 配置文件方式

Neuedu