day05_6_11_Spring的AOP

1. junit与spring的整合使用

spring中的所有jar包必须版本一致

将测试类注册到spring容器中，需要测试哪个对象，注入哪个对象

- 添加spring-test-5.3.4 的jar包
- 加载xml初始化容器，并把当前测试类注册到容器

  @RunWith(SpringJUnit4ClassRunner.class)
- classpath：绝对路径的写法，表示从根路径开始查找

  @ContextConfiguration("classpath:DITest.xml")

```
//加载xml初始化容器,并把当前测试类注册容器
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:spring.xml")//
classpath:绝对路径的写法,表示从根路径开始查找spring.xml
public class TestApplicationContext {

    @Resource
    private SysuserController uc;

    @Resource
    private ISysuserService us;

    @Test
    public void test1_ac(){
        System.out.println(uc);
    }

    @Test
    public void test2_us(){
        System.out.println(us);
    }
}
```

2. 什么是aop

aop：面向切面编程，在运行期间通过代理实现程序功能统一维护的一种技术，AOP是oop思想的扩展，利用aop可以对业务逻辑的各个部分进行隔离，从而使业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

通过oop面向对象编程实现了业务逻辑，对于后期需要添加的辅助性业务需求(日志，参数校验)使用aop思想进行扩展，辅助性的业务就是切面。

使用AOP利用一个代理将原业务对象包装起来，然后用该代理对象取代原始对象，可以在代理对象中对核心的业务逻辑进行扩展

aop中的几个概念：

切面（aspect）：辅助业务的业务对象，比如：日志，参数校验（切面类）

通知(advice)：切面类中的方法，分为5类：前置通知（方法，参数），返回通知（返回值），最终通知，异常通知（异常信息），环绕通知。

织入(weave)：在程序运行期间，通过动态代理模式，给目标对象创建代理对象，并把切面中的通知方法 织入到指定的连接点方法的前、后、异常、最终。

目标对象(target)：核心业务对象（需要扩展的业务对象）

连接点(joinPoint)：核心业务方法，目标对象中的某个业务方法

切入点(pointCut)：连接点的集合，通过一个切入点表达式可以灵活的定义多个连接点。

3. 通过动态代理实现aop

```
1  public static void main(String[] args) {
2    //被代理对象,该对象的pay方法需要扩展,添加日志,效率监测,参数校
   验.
3    IPay p = new AliPay();
4    IPay p2 = createProxy(p);//$Proxy100
5    p2.pay("aa","bb",11.1);//p.pay()
```

```
}

public static IPay createProxy(IPay p){
  //p:目标对象
  ClassLoader loader= p.getClass().getClassLoader();
  Class[] interfaces = new Class[]{IPay.class};
  InvocationHandler handler = new InvocationHandler()
{
    @Override
    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
      //proxy:代理对象
      String name = method.getName();
      if("pay".equals(name)){
        Object obj = null;
        //切面
        MyAspect a = new MyAspect();
        try{
          a.beforeAdvice();//通知
          obj = method.invoke(p,args);//执行连接点方法
          a.afterReturnAdvice();
        }catch (Exception e){
          a.exceptionAdvice();
        }finally {
          a.afterAdvice();
        }
        return obj;
      }

      return method.invoke(p,args);//p.toString()
    }
  };

  // Object obj = new $Proxy100(handler);
  //IPay proxy = (IPay)obj;
  IPay proxy = (IPay)
Proxy.newProxyInstance(loader,interfaces,handler);
  return proxy;
}
```

4. spring的aop注解使用

1. 添加spring-aop spring-aspect两个依赖包;spring的aop依赖于aspect第三方组建
2. 添加aspectj组件的三个依赖包
3. 添加cglib一个依赖包
4. 在spring的xml配置文件中aop注解识别
5. 创建切面类，创建通知方法

```
proxy-target-class="false"(默认) 默认使用Proxy类，对目标对象创建代理，如果目标对象没有接口，则采用cglib创建
proxy-target-class="true": 采用cglib创建代理
aspect组建的注解识别 @Aspect Pointcut Before AfterReturning AfterThrowing After
Before("")
After()
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

注解的使用方法

- 切入点注解@Pointcut("execution(* login(..))")
  - ```
    @Pointcut("execution(* login(..))")
    public void pc(){}
    ```
  -
  - execution（返回值类型，包名.类名.方法名(形参类型)），包名类名可以省略。
  - *:通配符
  - .. 不限制形参类型
- @Before("pc()")
    ```

```
@Before("pc()")
public void beforeAdvice(JoinPoint jb){
  jb.getArgs();//方法实参
  jb.getRarget();//源对象
  jb.getThis(); // 代理对象
 MethodSignature signature =（MethodSignature）
jp.getSingature()
 Method method = signature.getMethod();//获取方法
}
```

- @AfterReturning

```
@AfterReturning(poincut="pc()",returning="o")
public void AfterReturnAdive(Joinpoint jp,Object
o){

}
```

- @AfterThrowing

```
@AfterReturning(poincut="pc()",throwing="e")
public void AfterReturnAdive(Joinpoint
jp,Exception e){
  System.out.println(e)
}
```

- @After("pc()")
- 环绕通知

```
// 返回值必须是Object
// 形参必须是ProceedingJoinPoing
@Around("pc()")
public Object aroundAdvice(ProceedingJoinPoing
jp){
  Object result = null;
  //Object result = method.invoke(target,args)
  try{
    // 前置通知
    System.out.println("前置通知");
    //result = method.invoke(target,args)
    result = jp.proceed();
```

```
        // 返回通知
        System.out.println("返回通知");
    }catch(
        // 异常通知
        System.out.println("异常通知");
    )finally{
        // 最终通知
        System.out.println("最终通知");
    }
}
```

5. 切入点表达式

```
//切入点表达式两种写法:
//方法1:
@Pointcut(execution(返回值类型  包名.类名.方法名(形参类型))-----包名.类名可以省略
//@Pointcut("execution(* com.javasm.service.impl.*.*(..))")
// 方法2:自定义注解,@annotation(注解类名),表示带有指定注解的方法全部时连接点方法
// 第二种方式
@Pointcut("@annotation(com.javasm.annotation.tx)")
public void pc(){}
```

6. spring中的aop配置使用

```
<bean id="logAspect2"
class="com.javasm.aspect.LogAspect2"></bean>

<aop:config>
  <aop:aspect ref="logAspect2">
    <aop:pointcut id="pc"
expression="@annotation(com.javasm.annotation.tx)">
</aop:pointcut>
    <aop:around method="aroundAdvice" pointcut-
ref="pc"></aop:around>
    <!--<aop:before method="beforeAdvice" pointcut-
ref="pc"></aop:before>-->
    <!--<aop:after-returning
method="afterReturnAdvice" pointcut-ref="pc"
returning="o"></aop:after-returning>-->
    <!--<aop:after-throwing method="exceptionAdvice"
pointcut-ref="pc" throwing="e"></aop:after-throwing>--
>
    <!--<aop:after method="afterAdvice" pointcut-
ref="pc"></aop:after>-->
  </aop:aspect>
</aop:config>
</bean>
```

7. dom4j解析xml文件

Properties类解析.properties文件

Dom4j组件解析xml文件,pull组件解析xml文件.

中间件开发,必须需要进行xml解析

- 添加dom4j.jar
- Document doc = SAXReader.read(InputStream)
- Element root = doc.getRootElement()
- String str =root.attributeValue("属性名")
- List<Element> tags = root.elements("标签名")

8.easycode插件

- 安装插件

- 配置Type Mapper
- 配置Tmeplate settings