Spring 框架引言

Spring 不是替换某项技术，而是对项目中的组件(大部分是类，如 service 里的类)进行集中管理的工具（创建、使用、销毁）,以 spring 管理的方式代替了原先 new 的方式，Spring 就是一个管理工具。

最简单的 spring 只是简单的几个 spring 核心 jar 包和一个 xml，xml 里通过 bean 标签里的 class 参数指定类的全限定名，id 为这个类做唯一标识，

Property 来完成 class 之间的调用关系，name 为组件中的属性名，然后根据属性类型的不同，进行不同处理。最后声明组件的成员变量并实现公有的 set 方法。除 set (Property) 注入外，还有自动注入 (autowrite) 和构造注入(constructor-arg) (三种方式)。具体见 spring 文档的图片笔记，或本人的源码。

在 jar 包作用下我们就把这个类放入了 spring 管理工具中。

代码中我们用 ClassPathXmlApplicationContext 加载该配置文件，用 ApplicationContext 去接受该配置文件中类的容器，用 ApplicationContext.getBean(id)去得到容器中的类，这样我们就从 spring 工厂中获得了需要的类，替代了原先用 new 的方式来创建。通过非 new 的方式减少了 JVM 内存的开支。

```
<!--通过spring管理组件 bean: 用来管理组件对象的创建
class: 用来指定管理组件对象的全限定名 包.类
 id: 用来指定spring框架创建的当前组件对象在spring(容器|工厂)中唯一标识
    全局唯一 推荐: 接口的首字母小写 userDAO -->
<bean class="com.example.demo.setdi.ClazzDAOImpl"
    id="clazzDAOImpl"></bean>
<bean class="com.example.demo.setdi.ClazzServiceImpl"
    id="clazzServiceImpl">
    <!-- 自定义的引用类型用ref注入 -->
    <property name="clazzDAO" ref="clazzDAOImpl"></property>
    <!-- set注入相关语法 非自定义类用的是value注入 -->
    <property name="name" value="小陈123"></property>
    <property name="age" value="28"></property>
    <property name="sex" value="true"></property>
    <property name="price" value="28.88"></property>
    <property name="counts" value="28.99"></property>
    <!-- 注意，在spring中，日期的默认写法为: YYYY/MM/dd HH:mm:ss -->
    <property name="bir" value="2021/12/12"></property>
    <!-- 数组类型 array来完成实现 -->
    <property name="qqs">
        <array>
            <value>小陈</value>
            <value>小王</value>
            <value>小张</value>
        </array>
    </property>
    <property name="clazzDAOs">
```

```java
private Double price;

private Float counts;

private Date bir;
//数组类型
private String[] qqs;
private ClazzDAO[] clazzDAOs;
//集合类型  list
private List<String> habbys;
private List<ClazzDAO> clazzDAOList;
//map类型 map
private Map<String,String> maps;


//properties
private Properties properties;

public void setProperties(Properties properties) {
    this.properties = properties;
}
public void setMaps(Map<String, String> maps) {
    this.maps = maps;
}
public void setClazzDAOList(List<ClazzDAO> clazzDAOList) {
    this.clazzDAOList = clazzDAOList;
}
```


```java
public static void main(String[] args) {
    // 启动工厂
    ApplicationContext context = new ClassPathXmlApplicationContext("setdi/setdi.xml");
    // 获取工厂中创建好的对象 参数:获取工厂中指定对应的唯一标识
    //IOC :inversion of controll 控制反转 控制权力的反转
    //就是将手动通过new关键字创建对象的权力，交给spring，由工厂创建spring对象
    //DI: dependency injection   依赖注入 spring不仅要创建对象 还要在创建对象的同时维护组件与组件之间的依赖关系
    //定义：给组件中成员变量进行赋值过程
    //语法：1.组件中需要谁就将谁声明为成员变量并提供公开SET方法
    //2.在spring配置文件中对应的组件内部使用property标签来完成赋值操作
    ClazzDAOImpl clazzDAOImpl = (ClazzDAOImpl) context.getBean("clazzDAOImpl");
    clazzDAOImpl.save("王二狗");
    ClazzServiceImpl clazzServiceImpl = (ClazzServiceImpl) context.getBean("clazzServiceImpl");
    clazzServiceImpl.save("王三狗");
}
```

Spring　IOC　和　DI

IOC　控制反转

其实就是控制权力的反转，因为在没有 spring 工厂之前，我们都是通过 new 的方式来创建类，想在项目中何时何地创建就创建，但是有了 spring 工厂之后，我们将创建的权力交给了 spring（即在 spring 配置文件中加入 bean 标签），只要加载配置文件，该类就已经被创建了，我们只是按需从工厂中取出来使用，这种创建权力的反转又叫控制反转（IOC）。

DI　依赖注入

可以从工厂中拿到各个组件的实例往往有依赖嵌套关系的，例如：service 的很多接口都需要调用 DAO 层的接口，为了维护组件之间的依赖关系，完成组件之间的调用，spring 提供依赖注入技术（bean 标签中的 property 标签等），来完成组件之间的调用，这样就用依赖注入的方式代替了组件中用 new 的方式来调用另一个组件。

总之;IOC 就是将各个组件的创建权力交给 spring,DI 是用来维护组件之间的调用关系。

Spring 工厂特性

1. 组件创建模式

- Spring 默认管理组件对像（bean 标签）是单例创建（singleton）的,也就是说某一个组件无论从 spring 中取多少次，都是从同一个内存地址取出来的，即用"=="比较时返回 true，会产生数据共享的问题。
- Spring 也可以改为多例模式（prototype），即 scope 参数，防止同一内存的数据共享问题。

2. 组件创建原理也是 IOC 的底层原理。

- Bean 标签创建组件的原理：反射+构造参数，即用反射的 forName（全限定名）来创建

```
// 工厂原理
TagDAO tt = (TagDAO) Class.forName("scope.TagDAOImpl").newInstance();
System.out.println(tt);
```

3. Spring 管理组建的声明周期

- 单例对象：
  1) spring 工厂启动，工厂里的单例对象就会被创建(当含有单例对象 bean 标签的 XML 文件被加载到 jar 后，单例对象就会被创建)
  2) spring 工厂正常销毁（执行 close 方法），单例对象也会被销毁。
  3) 总结：工厂启动（加载 XML 后）就被创建，工厂关闭，就被销毁。
- 多例对象：
  1) 从 Spring 工厂中获取的时候才会被创建（即执行 getBean 的时候）
  2) 销毁是在没有对象引用的时候，由 JVM 垃圾回收。

4. Spring 的好处（主要是相对传统的 new 的方式来说）
   1) 解耦和：通过依赖注入的方式来代替组件之间用 new 相互调用，以后需要更改只需要更改配置文件而不是一个个更改源代码，降低了组件之间的耦合度。
   2) 降低内存使用率：默认的单例共享内存的方式来取代 new。

Spring AOP

定义：通过在程序运行的过程中动态的为项目中某些组件生成代理对象，通过代理对象执行附加操作或额外功能，减少通用代码的冗余问题。

1) 首先明确什么是代理？

在日常工作中，存在这种情况，例如 A 类执行了主要的业务逻辑（从数据库按条件

A 查询 listA,条件 B 查询 listB 等结果），然后我们又需要在这主要业务逻辑的基础上将结果输出到页面、转为 WORD、PDF、XLS 等格式将查询到的结果输出到本地，这个时候我们就需要创建专门转 word 的类 B，转 PDF 的类 C，转 xls 的类 D 等，我们就把这些手动创建类（B、C、D）叫做 A 的静态代理。

所以我们可以这么理解代理，和主业务类用同一个接口，调用主业务类的方法并在此基础上实现附加操作，叫做代理类，手动创建的叫静态代理，由代码动态创建的叫动态代理。具体 demo 见本人代码或操作文档。

2) 动态代理的具体理解

a) 通过代码动态的为目标类生成代理对象，生成的对象就叫动态代理对象

b) 动态代理的底层原理就是反射，具体是 `Proxy.newProxyInstance（）方法`来传入对应参数生成代理对象， 并实现附加的一系列操作，用 invoke 来执行被代理的方法，具体代码如下：

```
//参数1: classLoader  类加载器 可以利用线程来获得
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

//参数2: Class[]  目标对象的接口的类型的数组 （单继承多实现特性）
Class[] classes = {UserService.class};

//参数3: InvocationHandler 接口类型  invoke方法 用来书写额外功能 附加操作

//mybatis用的也是动态代理
//dao  Userdao  userdao = sqlSession.getMapper(userdao.class);  userdao.save() =
//mapper的xml文件 底层封装的也是下段代码
//传入接口，生成代理对象，将mapper.xml解析到invoke中进行执行操作。
//所以mybatis中不能出现方法重载，因为要根据代理对象执行的方法名，到mapper中找到对应的ID来获得invoke里面的内容

//proxy生成动态对象的类
//返回值：创建好的动态代理对象
UserService proxy = (UserService)Proxy.newProxyInstance(classLoader, classes, new InvocationHandler() {

    //通过动态代理对象调用自己里面代理方法时会优先指定invokcationHandler类中invoke
    //参数1：当前创建好的代理对象
    //参数2：当前代理对象执行的方法对象
    //参数3：当前代理对象执行的方法参数
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        System.out.println("当前执行的方法："+method.getName());
        System.out.println("当前执行方法的参数："+args[0]);
        try {
            System.out.println("开启事务");
            //调用目标类中业务方法通过反射机制，调用目标类中当前方法
            Object invoke = method.invoke(new UserServiceImpl(), args);
            System.out.println("提交事务");
            return invoke;
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("回滚事务！");
        }
        return null;
    }
});

System.out.println(proxy.getClass());
String result = proxy.findAll("小陈");//通过动态代理对象调用代理中方法
System.out.println(result);
```

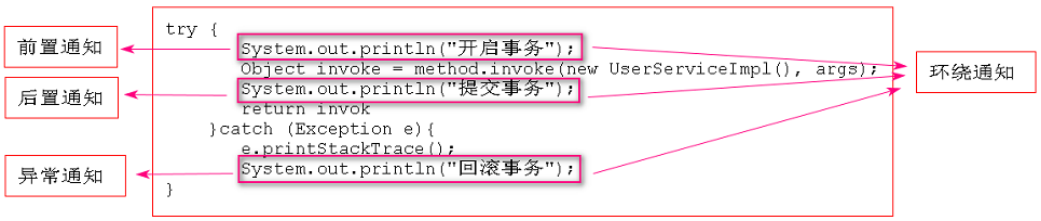
执行被代理对象方法，即主业务方法

生成动态代理对象

执行动态代理方法

3) Spring 的 AOP 底层封装了 `Proxy.newProxyInstance()` 生成动态代理对象的方法，利用反射生成代理对象，诞生了很多新名词

a) 把对被代理对象的附加操作，叫做通知。（环绕通知、前置通知、后置通知、异常通知）



b) 把被代理对象的被代理方法叫做切入点

c) 通知（附加操作）+切入点（被代理方法）叫做切面。AOP 又叫切面编程

d) 所以 AOP 的实现只需要传入切入点和通知即可。

e) 好处：在保证原业务不变的情况下，利用代理对象完成附加操作，实现了核心业务和附加业务的解耦，通常用来解决日志打印等冗余重复逻辑的代码。

f) Mybatis 底层也是利用的动态代理，传入 dao 接口，并将 mapper.XML 文件解析,通过 xml 文件里面的 ID 寻找与接口方法对应的 sql，将 sql 作为附加操作，完成动态

代理对象的生成，所以 mybatis 有个不能进行重载的特性，因为 xml 里面的 ID 会重复，影响动态代理对像的生成。

g) 默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。

4) AOP 的具体实现方式

最原始的实现方式：

a) 实现通知的接口，完成附加操作的添加

```
public class MyBeforeAdvice implements MethodBeforeAdvice{

    //before
    //参数1：当前执行方法对象
    //参数2：当前执行方法的参数
    //参数3：目标对象
    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        // TODO Auto-generated method stub
        System.out.println("==============前置通知====================");
        System.out.println("当前执行方法："+method.getName());
        System.out.println("当前执行方法参数："+args[0]);
        System.out.println("目标对象："+target);
        System.out.println("当前执行方法名称："+method.getName());
        System.out.println("==========================================");
    }
```

b) 配置文件中，配置通知类，并通过 aop 标签来完成切入点和切面的配置，切入点表达式常用两种（execution 方法级别，within 类级别）

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

    <!-- 管理service组件对象 -->
    <bean class="com.example.demo.aop.EmpServiceImpl"
        id="empServiceImpl"></bean>

    <!-- 注册通知 -->
    <bean class="com.example.demo.aop.MyBeforeAdvice"
        id="myBeforeAdvice"></bean>

    <!-- 组装切面 -->
    <aop:config>
        <!-- 配置切入点 pointCut id:切入点在工厂中唯一标识 expression: 用来指定切入项目中那些组件中那些方法 execution(返回值
            包.类名.*(..)) -->
        <aop:pointcut
            expression="execution(* com.example.demo.aop.*ServiceImpl.*(..))"
            id="pc"></aop:pointcut>
        <!-- 配置切面 advice-ref :工厂中通知ID pointcut-ref:工厂切入点唯一标识 -->
        <aop:advisor advice-ref="myBeforeAdvice" pointcut-ref="pc"></aop:advisor>
    </aop:config>
</beans>
```

c) 然后对已经配置了 AOP 切面的方法的调用，其实调用的是底层生成的代理对象，并不是真正调用的原对象。

```
public class TestSpring {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("aop/spring.xml");
        EmpService empService = (EmpService) context.getBean("empServiceImpl");
        System.out.println(empService.getClass());
        empService.find("小陈");
    }
```

d) 环绕通知，异常通知等通知，见本人 demo

5) Bean 标签的补充→复杂对象的创建

a) 简单对象 :可以通过 new 创建的叫简单对象，如实体，service 等，可以直接用 bean 标签管理

b) 复杂对象：不能通过 new 创建的叫复杂对象，如接口（Connection 等），抽象类（Calendar、MessageDisgest），不可以直接交给 Bean 标签进行管理。

c) 复杂对象 IOC 管理：手动创建该类的管理类 A，实现 `FactoryBean<T>`，泛型中填入复杂对象，并重写对应的方法（创建方式、复杂对象类型，单例或多例创建），然后 A 就可以用 Bean 标签管理，复杂类型就可以和简单类型一样使用了。

```java
//用来工厂中创建复杂对象
public class CalendarFactoryBean implements  FactoryBean<Calendar>{

    //用来书写复杂对象的创建方式
    @Override
    public Calendar getObject() throws Exception {
        // TODO Auto-generated method stub
        return Calendar.getInstance();
    }

    //指定创建的复杂对象的类型
    @Override
    public Class<?> getObjectType() {
        // TODO Auto-generated method stub
        return Calendar.class;
    }

    //用来指定创建对象的模式 true  单例 false  多例
    @Override
    public boolean isSingleton() {
        // TODO Auto-generated method stub
        return false;
    }
```

```
    xsi:schemaLocation= http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spr

<!-- 通过factoryBean创建复杂对象 -->
<bean class = "com.example.demo.factorybean.CalendarFactoryBean"  id = "calendarFactoryBean"></bean>

</beans>
```

```
public class Test {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("factorybean/spring.xml");
        Calendar calendar = (Calendar)context.getBean("calendarFactoryBean");

        System.out.println(calendar);

    }
}
```

6) 面试重点：循环依赖问题。

[1] 一个单例的 Bean 生命周期为：

a) 初始化化→创建原始对象，并赋予初始值

b) 实例化调用的对象→先查看单例池中有没有调用对象，如果有就拿来用，如果没有，将其实例化并放入单例池（一级缓存）

c) 填充属性

d) 如果有 AOP 进行 AOP 代理

e) 将代理对象或实例化后的对象放入单例池中。

[2] 但是这种流程在面对循环依赖时就会出现死锁的情况，例如 A 中调用 B，B 中调用 A 的问题。A 去实例化 B，B 又去实例化 A，就造成了死锁。

```
public class A {
    @Autowired
    private  B b;
}
```

```
public class B {
    @Autowired
    private A a;
}
```

[3] Spring 采用以下机制来解决循环引用的问题。

a) AOP 的各级缓存都是以 MAP 形式实现的

b) 一级缓存：单例池　　二级缓存：用来存放代理或原始对象　　三级缓存：用来存放原始对象及相关信息

c) 流程图见 PDF 文件，下面对其进行相关·流程的解释。

d) 任何 bean 的生命周期都要经过：初始化对象→实例化依赖→填充属性→AOP 代理对象生成→放入单例池

e) 但是由于代理对象和循环依赖的影响，使得每一环节的细节操作变得不那么简单，下面进行详述：

f) ①在初始化阶段，会将该对象存入一个 set 中，表示该对象正处于创建阶段；然后将半初始化的对象和其他信息以 lamuda 表达式的形式放入三级缓存中；

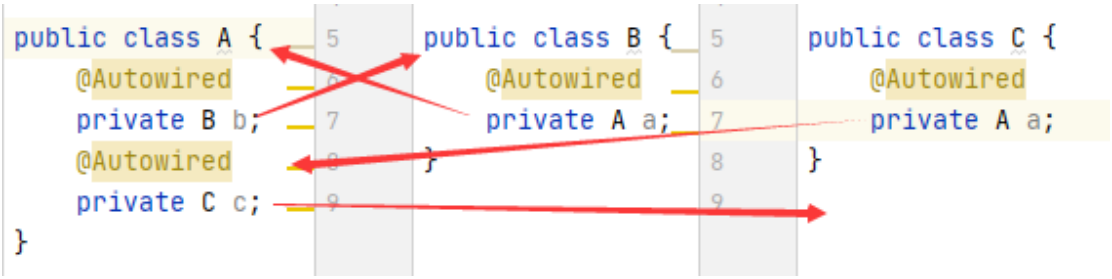
g) ②实例化依赖对象阶段：会先从单例池中（一级缓存）寻找，如果有就直接使用，如果没有就从二级缓存中寻找，有就直接使用，没有的话就判断 Set 结构中有无此实例对象，没有的话就按照 Bean 创建流程去创建，有的话就是发生了循环依赖，那么三级缓存一定不为空，

然后从三级缓存中，如果该对象有切面的话，就生成 AOP 代理对象，没有的话就生成初始化对象，存入二级缓存中，同时删掉三级缓存中的 lamuda 对象。

h) ③属性填充：对所有的半初始化对象，进行完整的赋值及创建工作。

i) ④AOP 代理：未发生循环依赖的话，正常的 AOP 代理发生在这个阶段。首先会判断二级缓存是否已经发生了 AOP 代理，如果是，那就直接从二级缓存中取出放入单例池中，如果没有，那就判断该对象是否需要 AOP 代理，如果需要就生成代理对象放入单例池中，如果不需要就将实例化好的对象放入单例池中。

j) 下面通过举例的方式来描述 Spring 循环引用的过程。



a) 首先创建 A 对象，Set 结构中放入 A 对象的一个实例，表示正在创建中；并在三级缓存中整合 lamuda 表达式放入 A 对象的及其他信息

b) 实例化对象 B，首先在单例池中寻找 B 的实例，然后并没有找到，然后去二级缓存中寻找，B 的实例，没有找到，所以就进入 class B 进行创建

c) 创建 B 对象，Set 结构放入 B 对象的一个实例，lamuda 整合初始化对象其它相关信息放入三级缓存；实例化对象 A，首先从单例池中寻找 A 的实例，没有找到，去二级缓存中寻找 A 的实例，没有找到，Set 结构中发现存在 A 的实例，表明发生了循环依赖问题，从三级缓存中取出 A 的 lamuda 表达式，因为 A 没有切面，所以直接将 A 的半初始化对象放入到二级缓存中，同时删除三级缓存中 A 的 lamuda 表达式，然后经过属性的填充，所有的半初始化对象变成了实例化对象，AOP 阶段 B 对象没有配置切面，所以直接将 B 对象的实例化对象放入单例池中，B 实例化完毕。

d) 创建对象 C，进入 class　C，Set 结构放入 C 对象的一个实例，去单例池中寻找 A 的实例化对象，没有找到,然后去二级缓存中寻找，找到了 A 的实例化对象，直接使用，然后，填充属性阶段，AOP 阶段，C 并没有配置切面，所以生成一个实例化对象放入单例池即可，C 实例化完毕。

e) B,C 都已实例化完毕，且都已经存入单例池中，A 对象创建过程进入了属性填充阶段

f) AOP 阶段：A 并没有配置切面，所以直接生成一个单例对象放入单例池即可。

g) A 对象创建完毕，且 B、C 创建完毕。

h) 如果看不懂请参考 CSDN　spring 模块

至此，IOC 和 AOP 部分完结

Spring 整合 mybatis

1. spring 整合 mybatis 具体思路就是:将 mybatis 管理对象的能力交给 spring,由 spring 统一管理，而 mybatis 所有的组件都来源于 SQLSessionFactory，所以接管该对象就接管

了 mybatis 的管理组件权力。

2. SQLSessionFactory 是接口（复杂对象），所以不能直接用 bean 标签管理，要实现 factoryBean<T>接口并加载 mybatis 配置文件，用 bean 标签来管理，由于上述步骤是重复不变的代码，mybatis 公司就整合成新的 jar 包，提供简单对象 SQLSessionFactoryBean,来减少代码量，但是这次数据源文件（填链接用户密码的文件）和 mapper 文件必须分开加载，这也是 springboot 加载数据库的雏形。

```
<!-- 然后是对数据库的一系列操作：绑定数据库以及dao层 -->
<!-- 创建database链接信息 -->
<bean id="dataSource"
    class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/test?characterEncoding=UTF-8"></property>
    <property name="username" value="root"></property>
    <property name="password" value="123888"></property>
</bean>
<!-- 创建sqlSessionFactory 整合链接信息，映射 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 依赖的链接信息 -->
    <property name="dataSource" ref="dataSource"></property>
    <!-- 注入mapper配置文件 -->
    <property name="mapperLocations">
        <array>
            <value>classpath:mybatisDao/UserDAOMapper.xml</value>
        </array>
    </property>
</bean>
<!-- 创建DAO 组件 将DAO接口和数据库链接绑定在一起  最终也是使用的该ID-->
<bean id="UserDAO"
    class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 注入sqlSessionFactory -->
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    <!-- 注入创建DAO接口类型 注入接口的全限定名 包接口名 -->
    <property name="mapperInterface"
        value="com.example.demo.mybatisDao.dao.UserDAO"></property>
</bean>
```

```
public static void main(String[] args) {

    ApplicationContext context = new ClassPathXmlApplicationContext("mybatisDao/spring.xml");
    UserDAO userDAO = (UserDAO) context.getBean("UserDAO");

    userDAO.findAll().forEach(user -> System.out.println("user = " + user));
}
```

mybatis 和事务的交互

1. 首先事务是 JDBC 层面的，任何一次对数据库的操作（执行 sql 语句，事务的提交、回滚等），都是 connection 对象携带链接、用户密码等信息进行的操作。

2. 由于 mybatis 使用的是动态代理，所以 service 层的 connection 对象无法显式的传送到 dao 层，很容易导致同一个线程 service 层 new 的 connection 对象和 mybatis 动态代理产生的对象不是同一个，即 dao 层出错，connection 回滚，但是 service 层使用的是另一个 connection 对象，并不能一起回滚。所以 spring 提供了 bean 标签来管理事务保证同一线程链接对象（connection）的一致性。

3.mybatis 做了一个类 *DataSourceTransactionManager*，全局的事务管理器，用来保证业务层当前线程使用连接对象（接口）和 DAO 层实现连接对象一致，即 connection 一致。这也是@transaction 的雏形，但是注意这个标签是为了保证同一个线程上两个 connection 对象的一致性，事务是 JBDC 层面的东西

```xml
<!-- 先对组件进行管理 -->
<bean class="com.example.demo.mybatisDao.service.UserServiceImpl"
    id="userServiceImpl">
    <property name="platformTransactionManager"
        ref="transactionManager"></property>
</bean>

<!-- 注册事务，对需要添加事务的添加该配置 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源对象 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

```java
// 生成事务管理器
private PlatformTransactionManager platformTransactionManager;

public void setPlatformTransactionManager(PlatformTransactionManager platformTransactionManager) {
    this.platformTransactionManager = platformTransactionManager;
}

@Override
public void save(user u) {
    // TODO Auto-generated method stub
    // 创建事务配置对象
    TransactionDefinition transactionDefinition = new DefaultTransactionDefinition();
    //获取事务状态
    TransactionStatus status = platformTransactionManager.getTransaction(transactionDefinition);
    try {
        userDAO.save(u);
        platformTransactionManager.commit(status);
        System.out.println("=============已提交================");
    } catch (TransactionException e) {
        // TODO Auto-generated catch block
        platformTransactionManager.rollback(status);
        System.out.println("=============已回滚================");
```

3. 起初我们需要为每一个方法编写事务，即上图所示，为每一个需要事务的方法手动的进行 commit 和 rollback 操作，称为编程式事务处理。由于代码重复冗余问题，我们就交给 AOP 利用反射生成代理对象来做，如下图，我们把这个事务叫做 :声明式事务处理。

```
TransactionAdvice implements MethodInterceptor{

  private PlatformTransactionManager ts; set;

  public Object invoke(MethodInvokcation mi){
    //创建事务配置对象
    TransactionDefinition transactionDefinition =  new DefaultTransactionDefinition();
    //获取事务状态
    TransactionStatus status = ts.getTransaction(transactionDefinition);
    //放行目标方法
    try{
        Object result = mi.proceed();//放行
        ts.commit(status);//提交事务
        return result;
     }catch(Exception ..){
        ts.rollback(status);
     }
  }
}
```

环绕通知

b.配置切面

1).配置通知对象

```
<bean  id="tx" class="xxx.TransactionAdvice">
   <property name="ts" ref="datasourceTransactionManager">
</bean>
```

为环绕通知加事务管理器

2).配置切面

```
<aop:config>
    <aop:pointcut id="pc" expression="within(com.baizhi.service.*ServiceImpl)"/>
    <aop:advisort advice-ref="tx" pointcut-ref="pc">
</aop:config>
```

配置切面，以后凡是用到该类的都会用它的代理对象，并执行上述的环绕通知来完成事务

由于通知，即附加操作都一样，都是进行 commit 和异常 rollback 操作，spring 对此进行进一步封装，tx 标签来取代原先手动写环绕通知的情况，并且默认是不加事务的，必须精确到方法名才行。如下图所示

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
   <!--事务细粒度配置-->
   <tx:attributes>
    <tx:method name="save*"/>
    <tx:method name="add*"/>
    <tx:method name="insert*"/>
    <tx:method name="delete*"/>
    <tx:method name="update*"/>
   </tx:attributes>
</tx:advice>
```

具体到每个方法

配置切面，生成代理对象，以后使用的都是附加了事务的代理对象

b.配置切面

```
<aop:config>
 <aop:pointcut id="pc" expression="within(com.baizhi.service.*ServiceImpl)"/>
 <aop:advisort advice-ref="tx" pointcut-ref="pc">
</aop:config>
```

4. 事务的传播属性，本质是将同一个链接（connecttion）传递给其他方法，以此来达到共用链接的目的，且公用同一个事务，所有相互调用的方法，要么全部成功，要么全部回滚。
5. 此外事务还有读写（readonly）和异常性（rollback-for && no-rollback-for=）和超时性（timeout）
6. 面试重点：事务的 AOP 特性、传播特性以及异步特性（详见 CSDN）

1) 基础知识

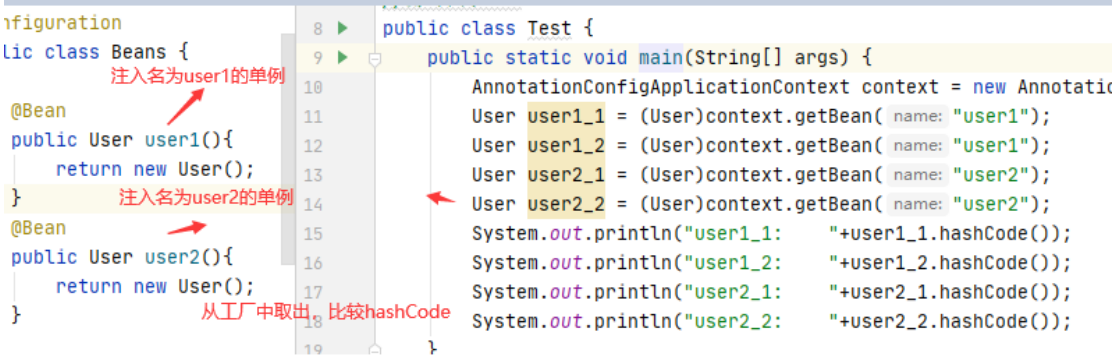
a) Bean 单例并不是单例模式,而是单例池以 Map 形式管理的同单例名的同一对象,默认是该类的首字母小写存入 Map

b) 当多个线程调用同一个单例，会用·ThreadLocal 形成线程副本，线程之间是相互隔离的。

c) Spring 对事务的封装，其实是对 Connection 对象的封装，connection 对象是单例 Bean，当多个线程进行访问时，会在每个线程形成 ThreadLocal 副本，线程间的 connection 是互不影响的

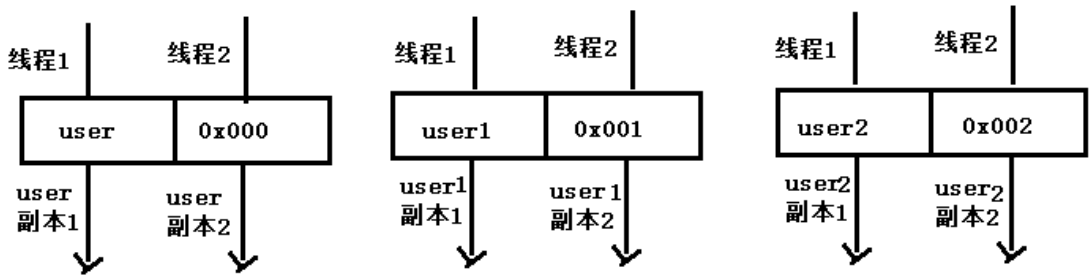
d) 所以事务只能在同一个线程中传播

e) this.a()就相当于直接调用 a(),this 指的是本实例




```
user1_1:    1205406622
user1_2:    1205406622
user2_1:    293907205
user2_2:    293907205
```

可以看到单例名一样，在工厂中只有一个地址，单例名不一样，工厂中地址不一样



2) AOP 的自调用、重注入和异步调用：

a) 类在实现 AOP 代理后，注入使用的就是代理对象，而不是原对象了

b) 本类重新注入和注入别的类的实例，都是引入了新的的实例对象

c) 同一实例的自调用，被调用方法会被抹去 aop 特性而走原始对象，不同实例间的调用，AOP 特性不会受到影响

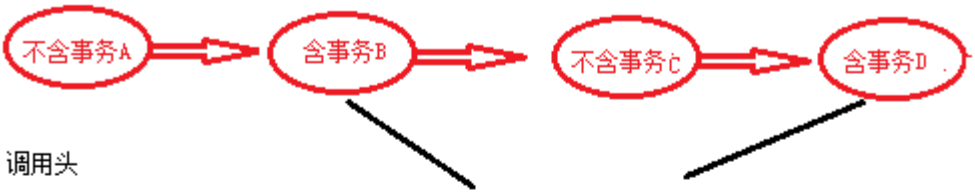
d) 类中存在重新注入又存在异步调用的时候，启动时会报错，建议把异步调用放在没有重新注入的类中

```
import org.springframework.bean
//都是使用了不一样的实例
public class AopTest {
    //自注入
    @Autowired
    private AopTest aopTest;

    //注入其他类
    @Autowired
    private Beans beans;
}
```

3) 事务的自调用、重注入以及异步调用：
   a) Spring 事务也是用 AOP 来实现的，它的后置通知是 connection.commit，它的异常通知是 connection.rollback
   b) 既然是用 AOP 来实现的，那么上述 AOP 的性质，事务同样拥有，下面是一些结论
   c) ①事务中的异常如果被捕获了，没有被 throw 或者正常抛出，那么是不会回滚的，我们称这个异常被消化掉了，见演示①
   d) ②根据 AOP 特性，同一实例的自调用，被调用方会失去代理特性走原始对象，所以同一实例事务的自调用，被调用方将失去事务特性而变成普通方法。又事务传播特性：如果调用头没有事务，那么整个调用链 connection 是不一致的，有异常也不会回滚，即使被调用方有事务也会失去，见演示②；如果调用头有事务，调用链 connection 对象是一致的，只要存在没有被捕获的异常就会回滚，虽然被调用方事务特性失去了，见演示③
   e) ③类的重注入和注入其他类的实例，都是引入了新的实例，只不过存在重注入的类又存在异步调用启动会报错，见演示④
   f) ④根据 AOP 特性，不同实例间 AOP 方法的调用，不会影响 AOP 方法的特性，走的是代理对象，就事务而言，不同实例间的调用，不会影响被调用方的事务特性，又事务的传播特性，调用链中，第一个生效的事务 A 将会把 connection 对象传至最后，A 及 A 之后的链路出现未捕获的异常，connection 将会回滚，但 A 之前的链路不会回滚，注意如果要捕获这个异常，必须在该异常节点链路上游中最近的事务生效的节点以及之前进行捕获，才能保证 connection 对象不会回退，见演示⑤
   g) ⑤上述讲的都是同步的情况，即在同一个线程中的情况，下面讲异步的情况
   h) ⑥类中实现异步的前提是不能有重注入，见演示④
   i) ⑦异步调用的话，事务 AOP 方面的性质和同步调用一样，即实例的自调用，被调用方将会失去事务特性而变成普通方法，实例间（自注入或注入它类的实例）的调用，被调用方不会失去事务特性，这个可以从动态代理方面分析。但是传播特性这一性质发生了改变，由上可知，connection 对象是单例 bean，在单例池中以 Map 形式管理，当多个线程同时访问 connection 对象时，会用 Thread Local 在每个线程中形成一个副本，线程之间互不影响，所以每个线程 connection 对象都是不一样的，connection 对象只能在同一个线程中传播，回滚，一个线程中 connection 对象的回滚，并不会影响到另一个线程 connection 对象，因为他们是不一样的，见演示⑥
   j) ⑧另外说一下，@async 异步调用标签也是用 AOP 做的，所以遵守 AOP 的特性，即实例自调用就会失效，实例间的调用才会生效，见演示⑦

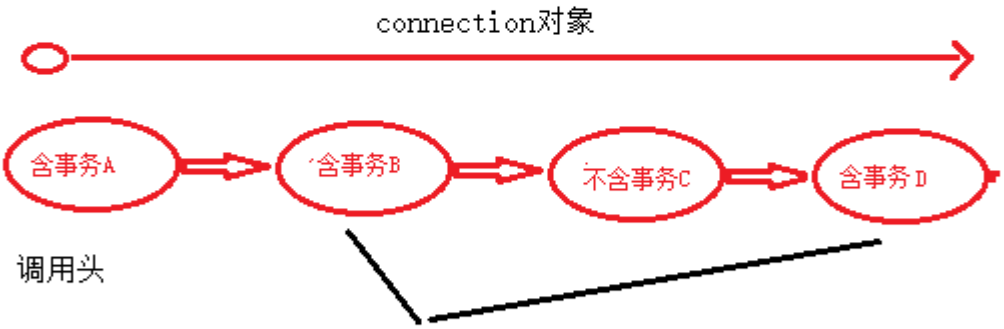
注意前提是自调用



调用头

即使包含事务，作为被调用者，也会失去事务的特性，变为普通方法

B、D乃至整个调用链路出现异常，都不会回滚



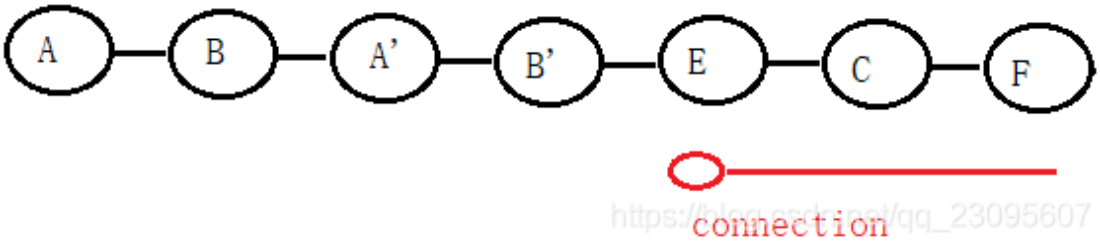
调用头

被调用者，即使包含事务，
也会失去事务的特性，变为
普通方法

B、D和C一样，变为普通方法
但是由于事务的传播特性，使得ABCD拥有了同一个connection对象
如果出现了没有被捕获的异常，就会导致整个链路的回滚

例如：D方法的异常，在D、C、B、A中捕获都行，但是必须被捕获到

调用链路



（1）. A 方法调用 B，属于实例自调用，B 方法失去事务特性。

（2）.B 方法调用 A'，属于实例间调用，但是 A'并没有事务。

（3）.A'方法调用 B',属于实例自调用，B'方法失去事务特性。

（4）.B'方法调用 E，属于实例间调用，E 方法事务生效，且 connection 开始向后传播。

（5）.E 方法调用 C，属于实例间调用，C 方法事务生效，且 connection 和 E 方法一致。

（6）.C 方法调用 F，属于实例间调用，F 方法没有事务，但是 connection 和 E,C 方法一致。

（7）.所以 A 到 B'无论哪个环节出现异常都不会回滚，因为事务都没生效，connection 也不一致。

（8）.E 到 F 无论哪个环节出现异常，且 E 到 F 没有捕获（try{}catch(){}）,E 到 F 整个链路都会回滚。

（9）.E 到 F 如果 F 环节出现异常，F 链路上游中，最近的事务生效的节点是 C，所以异常必须在 F 或者 C 中捕获，才能保证整个 connection 对象不会回滚。

（10）.E 到 F 如果 C 环节出现异常，C 链路上游中，最近事务生效的节点是 C，所以异常必须在 C 中捕获，才能保证整个 connection 对象不会回滚。

（11）.E 到 F 如果 E 环节出现异常，E 链路上游中，最近事务生效的节点是 E，所以异常必须在 E 中捕获，才能保证整个 connection 对象不会回滚。

```java
@Transactional
public void E(){
    Users u = new Users();
    u.setId(UUIDUtils.getUUID());
    u.setUsername("peter");
    usersMapper.insert(u);
    try {
        C();
    }catch (Exception e){}
}
@Transactional
public void C(){
    Users u = new Users();
    u.setId(UUIDUtils.getUUID());
    u.setUsername("zhangsan");
    usersMapper.insert(u);
    F();
}
public void F(){
    throw new RuntimeException("error demo");
}
```

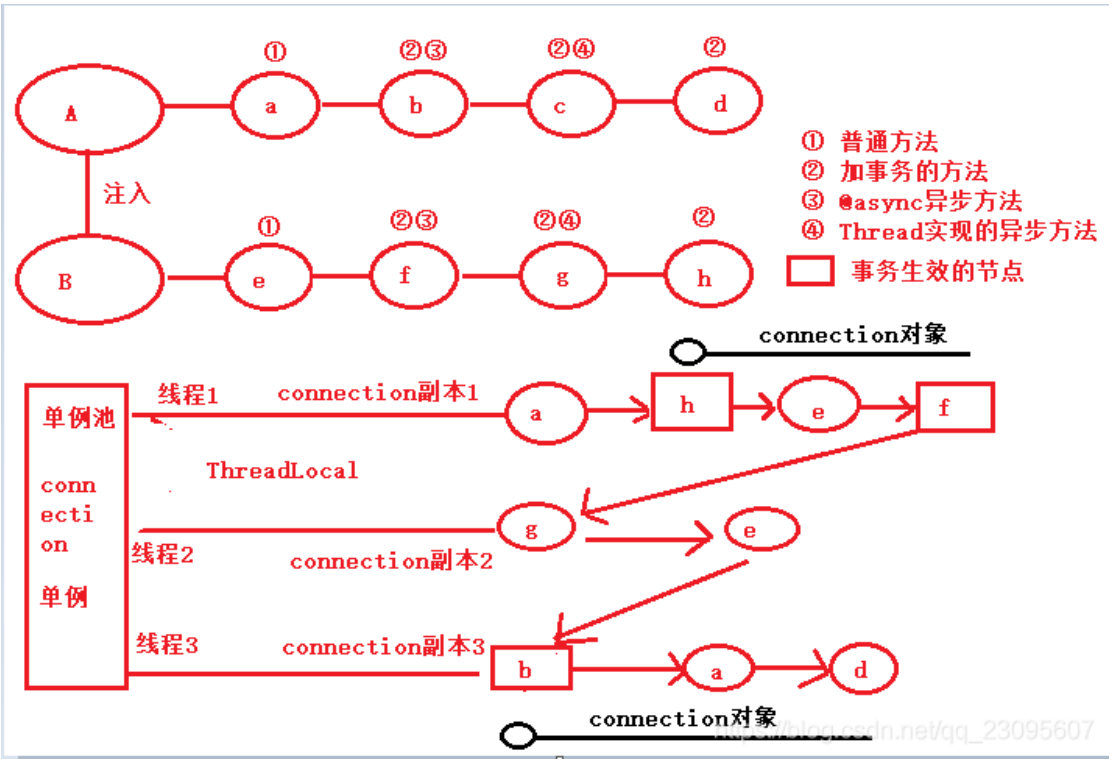
异常捕获

调用

只要在connection有效范围内把异常捕获掉，无论哪个环节，都可以防止链路回滚

调用

这个仅仅是用自调用的方式演示调用链路的异常捕获问题，图中演示的是不同实例的调用



（1）. A 中注入 B，A 中有 a、b、c、d 方法，B 中有 e、f、g、h 方法，他们都有事务和异步上的特性

（2）.根据以上说明，@async 也是用 AOP 来做的，所以也遵守 AOP 特性

（3）.下面说以上的调用过程：

（4）.a 调用 h，属于实例间的调用，根据 AOP 特性，h 的事务特性生效，且 connection 对象开始向后传播

（5）.h 调用 e，属于自调用，e 没有事务特性，但是由于 h 事务的生效，e 和 h 公用同一个 connection 对象

（6）.e 调用 f，属于自调用，f 有@saync 注解和事务，根据 AOP 特性，全部失效，变为普通方法，又事务的传播特性，f、h、e 公用同一个 connection 对象

（7）.f 调用 g，属于自调用，所以 f 的事务失效，又 f 使用的是 thread 来进行异步调用，所以会到另一个线程中执行，h 的 connection 将不会传播至 g，事务的传播特性也就此中断

（8）.g 调用 e，属于自调用，又 g 的事务特性由于自调用原因失去了，且 h 的传播特性到 g 由于异步的原因也失去了，所以 g 和 e 的 connection 对象将不再一致

（9）.e 调用 b，属于实例间调用，@async 是用 AOP 做的，事务也是 AOP，所以全部生效，b 将在另一个线程中开启事务，且将 connection 对象传递下去。

（10）.b 调用 a，属于自调用，a 的事务失效，但是 b 的事务生效，所以 connection 将传播至 a，a、b connection 一致

（11）.a 调用 d，属于自调用，d 没有事务，但是线程中 b 的事务生效使得 a、b、d 的 connection 一致。

（12）.所以，a、g、e 即使出错也不会发生回滚，根据异常节点链路上游中最近的事务生效的节点以及之前进行捕获，才能保证 connection 对象不会回退，h、e、f 中，h、f 为事务生效的节点，所以，f 出现异常，必须在 f 中捕获才不会回滚，e 出现异常，可以在 e 和 h 中捕获，h 中出现异常，必须在 h 中捕获才不会回滚。同理，b、a、d 段也是这么处理。

4) 事务的相关演示：

a) 演示①：事务只有在 throw 或者正常发生异常才会回滚，被捕获是不会回滚的

```java
//正常抛异常  事务回滚
@Transactional
public void test1(){
    User u = new User();
    List<User> users = userMapper.SelAll();
    User temp = users.get(users.size()-1);
    u.setId(temp.getId()+1);
    u.setPassWord("123456789");
    u.setRealName("xiaoli");
    u.setUserName("xiaoli");
    userMapper.insert(u);
    throw new RuntimeException("a error!");
}
```

正常发生异常

添加xiaoli相关数

相当于 int num = 1/0

```java
//throws出异常  事务回滚
@Transactional
public void test2() throws Exception{
    User u = new User();
    List<User> users = userMapper.SelAll();
    User temp = users.get(users.size()-1);
    u.setId(temp.getId()+1);
    u.setPassWord("123456789");
    u.setRealName("xiaoli");
    u.setUserName("xiaoli");
    userMapper.insert(u);
    throw new RuntimeException("a error!");
}
```

throws抛出异常

添加xiaoli相关数

id	userName	passWord	realName
1	peter	******	user:add
2	tom	123456	user:update
3	Alicante	1515	user:add
4	joinker	159753	user:delete
5	alice	456852	user:help
6	leveno	62486	user:fix
7	芬达	123456	美年达
9	可乐	cocacola	百事可乐
10	可乐	cocacola	百事可乐

数据库里并没有xiaoli的相关数据，事务发生了回滚

```java
//捕获异常  事务不回滚
@Transactional
public void test3(){
    User u = new User();
    List<User> users = userMapper.SelAll();
    User temp = users.get(users.size()-1);
    u.setId(temp.getId()+1);
    u.setPassWord("123456789");
    u.setRealName("xiaoli");
    u.setUserName("xiaoli");
    userMapper.insert(u);
    try {
        throw new RuntimeException("a error!");
    }catch (Exception e){e.printStackTrace();}
}
```

添加xiaoli相关数

捕获异常

id	userName	passWord	realName
1	peter	******	user:add
2	tom	123456	user:update
3	Alicante	1515	user:add
4	joinker	159753	user:delete
5	alice	456852	user:help
6	leveno	62486	user:fix
7	芬达	123456	美年达
9	可乐	cocacola	百事可乐
10	可乐	cocacola	百事可乐
11	xiaoli	123456789	xiaoli

数据库存储了xiaoli相关数据，并没有回滚

b) 演示②：自调用，被调用方的事务特性将会失去

```java
//没有事务的普通方法
public void test1(){
    test2();
}
//有事务的被调用方法
@Transactional
public void test2(){
    User u = new User();
    List<User> users = userMapper.SelAll();
    User temp = users.get(users.size()-1);
    u.setId(temp.getId()+1);
    u.setPassWord("123456789");
    u.setRealName("xiaoli");
    u.setUserName("xiaoli");
    userMapper.insert(u);
    throw new RuntimeException("a error!");
}
```

调用

添加xiaoli相关信息

抛异常

id	userName	passWord	realName
1	peter	******	user:add
2	tom	123456	user:update
3	Alicante	1515	user:add
4	joinker	159753	user:delete
5	alice	456852	user:help
6	leveno	62486	user:fix
7	芬达	123456	美年达
9	可乐	cocacola	百事可乐
10	可乐	cocacola	百事可乐
11	xiaoli	123456789	xiaoli

被调用方法即使有事务，也会失去，抛异常不会回滚

c) 演示③：含有事务的调用头调用链路的异常问题

```java
//具有事务特性的调用头
@Transactional
public void test1(){
    Users u = new Users();
    u.setId(UUIDUtils.getUUID());
    u.setUsername("peter");
    int num =  usersMapper.insert(u);
    test2();
}
//不具有事务特性的被调用对象
public void test2(){
    test3();
    throw new RuntimeException("error!!");
}
//具有事务特性的被调用对象
@Transactional
public void test3(){
    User u = new User();
    List<User> users = userMapper.SelAll();
    User temp = users.get(users.size()-1);
    u.setId(temp.getId()+1);
    u.setPassWord("123456789");
    u.setRealName("xiaoli");
    u.setUserName("xiaoli");
    userMapper.insert(u);
```

事务

调用头

添加peter

调用

没有事务的方法抛异常

调用

添加xiaoli

id	username	nickname	password	salt
(Null)	(Null)	(Null)	(Null)	(Null)

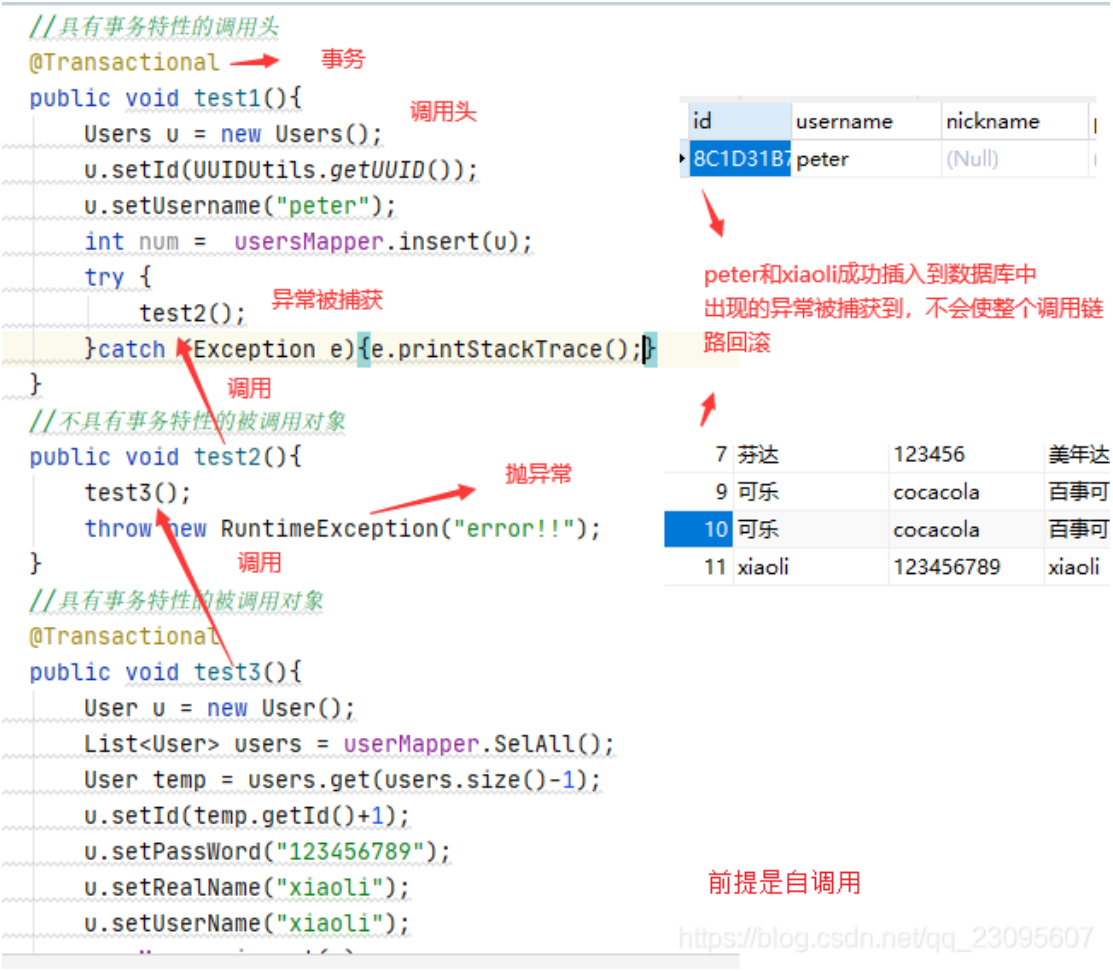
没有peter

id	userName	passWord	realName
1	peter	******	user:add
2	tom	123456	user:update
3	Alicante	1515	user:add
4	joinker	159753	user:delete
5	alice	456852	user:help
6	leveno	62486	user:fix
7	芬达	123456	美年达
9	可乐	cocacola	百事可乐
10	可乐	cocacola	百事可乐

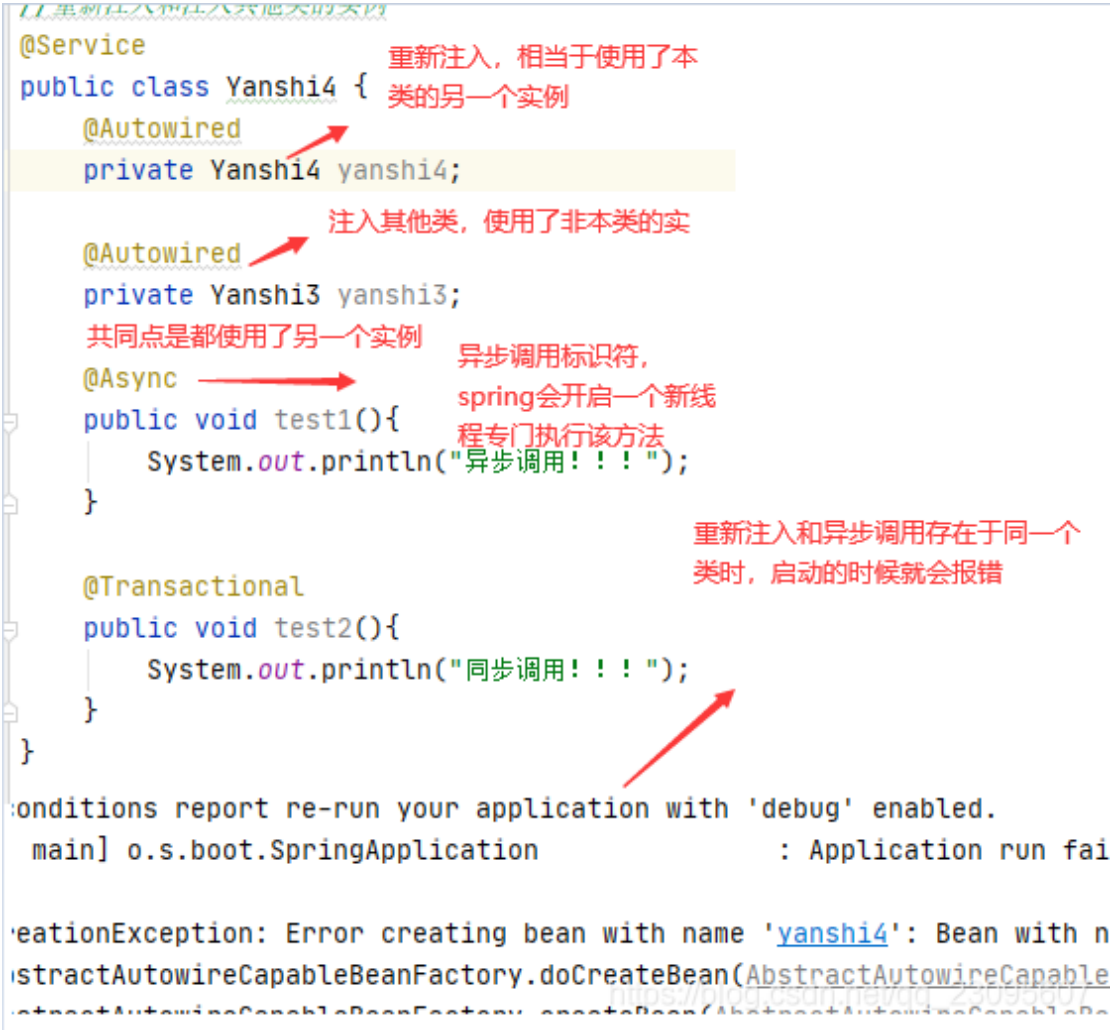
没有xiaoli

有事务的调用头，在整个调用链路中，只要出现没有被捕获的异常，即使是普通方法抛出的，也会造成回滚

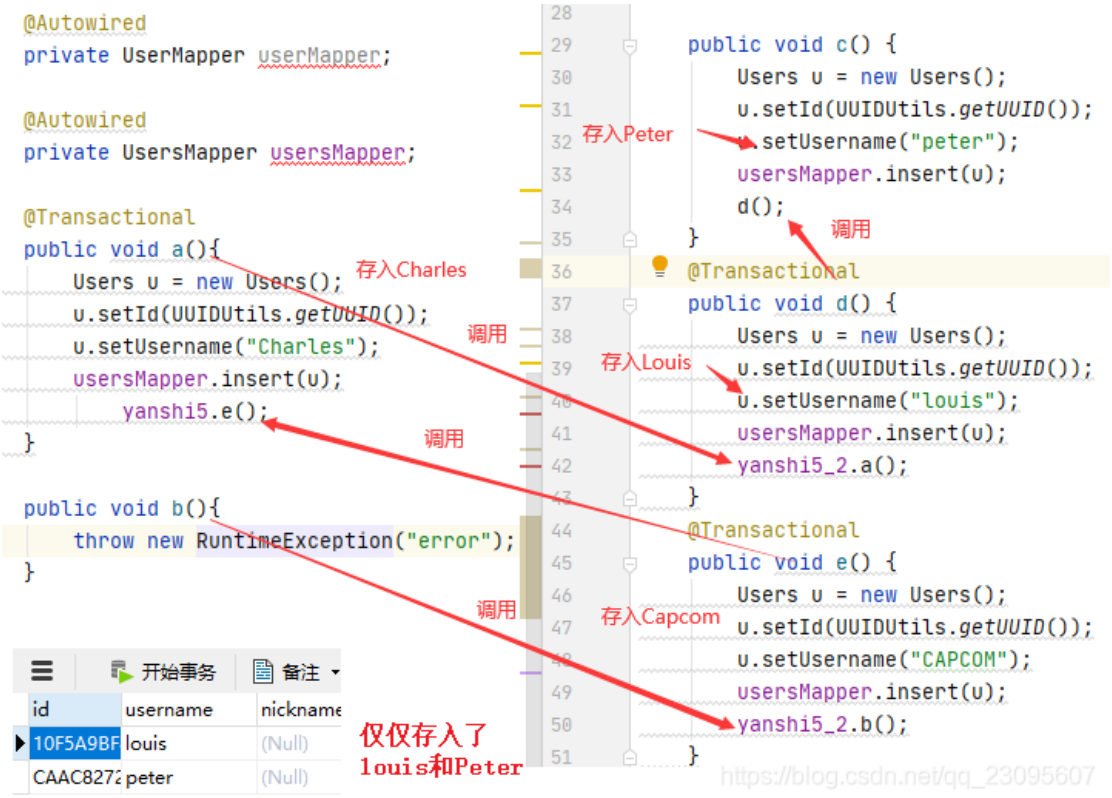
前提是自调用

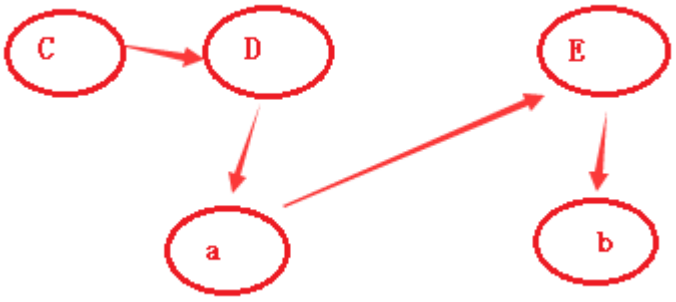


d) 演示④：异步调用+重注入出错



e) 演示⑤：实例间调用，事务生效部分才会回滚





（1）.c 调用 d，因为是自调用，所以 d 事务失效

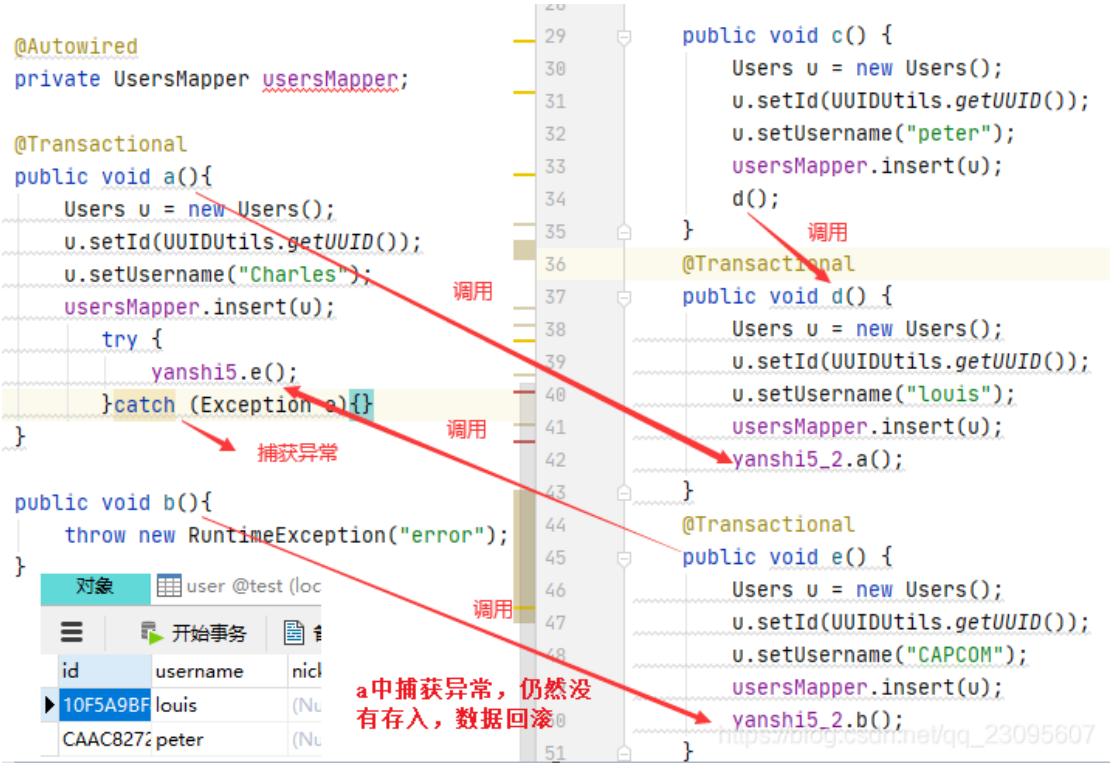
（2）.d 调用 a，是实例间调用，a 事务生效，并开始将 connection 对象向后传

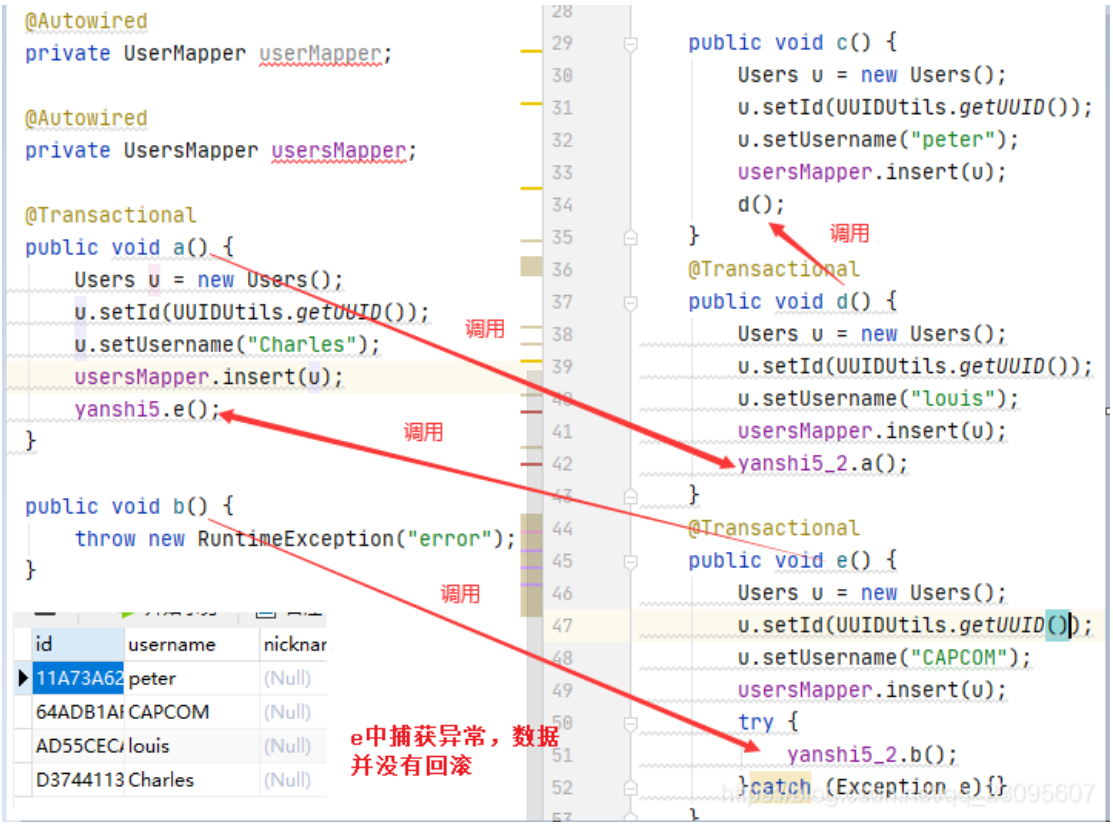
（3）.a 调用 e，是实例间调用，e 事务生效，并和左边 test1 的 connection 一致

（4）.e 调用 b，实例间调用，虽然 b 没有事务，但是依然会接收到 e 的 connection 对象

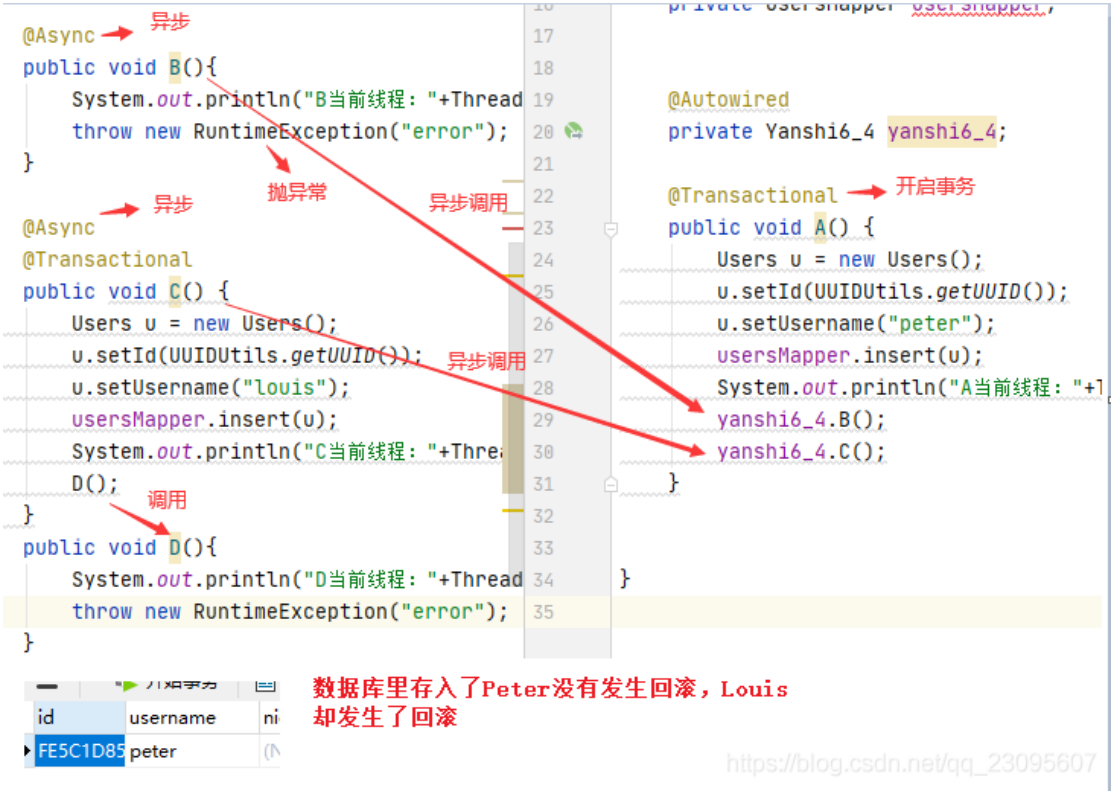
（5）.由于 a 事务的生效以及事务的传播，使得 a、b 和 e connection 一致，且抛出了异常没有被捕获到，所以 connection 对象发生了回滚，即 a、b 以及 e 发生了回滚，没有存入数据库，但是 c 和 td 不会发生回滚，因为不在一个 connection 对象上。

（6）如果 b 抛出的异常,根据事务的第 6 条，必须在该异常节点链路上游中最近的事务生效的节点以及之前进行捕获，所以，必须在 b 或者 e 中进行捕获，否则会造成整个 connection 对象的回退，即 a,e ,b 方法的回滚。

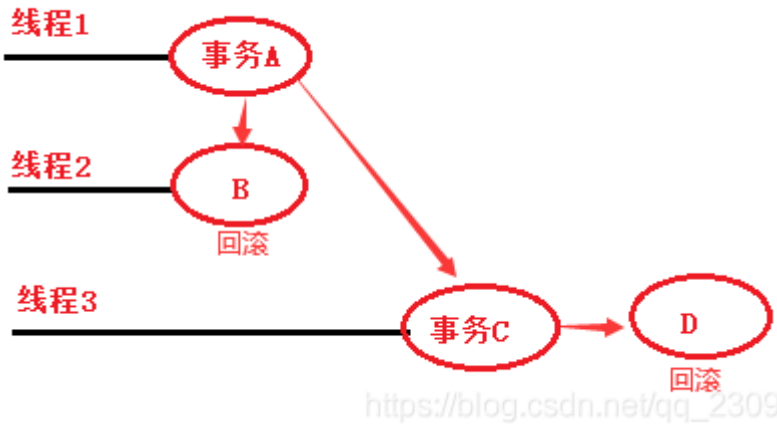




f) 演示 6：异步调用下，事务的传播及回滚问题



上述流程可以简化为

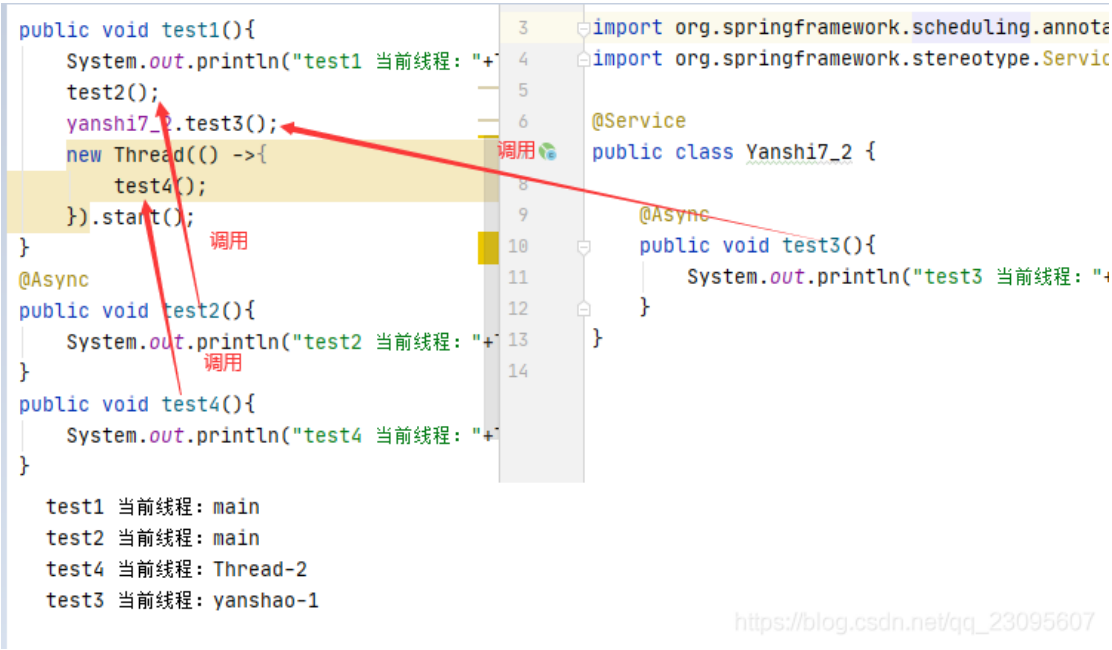


（1）.A 拥有事务，会将 connection 向后传递，如果是同步调用的话，B 会与 A 的 connection 一致，B 抛出异常 A 回滚，但是异步调用会阻断 connection 的传递，A 与 B 不再共用一个 connection 对象，B 抛异常并不会导致 A 的回滚

（2）…A 拥有事务，会将 connection 向后传递，如果是同步调用的话，C 也会变成事务生效的节点且会与 A 的 connection 一致，即 ACD 共用一个 connection，但是异步调用会阻断 connection 对象的传递，A 用一个 connection 对象，CD 共用一个 connection 对象，D 抛异常会导致 C 的回滚而不会导致 A 的回滚

（3）.所以 A 提交，C 回滚

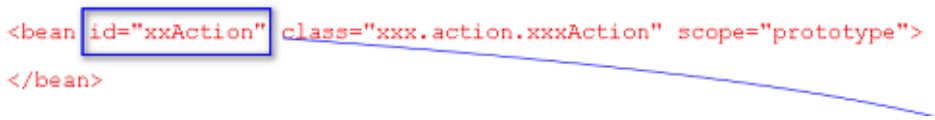
g) 演示 7：@async 同步调用失效，异步调用生效



可见 test1 和 test2 用了同一个线程，异步注解并没有生效，test1 和 test3 用的不是同一个线程，异步注解生效，test1 和 test4，不用 AOP 的方式，以最原始的方式也可以实现异步调用。

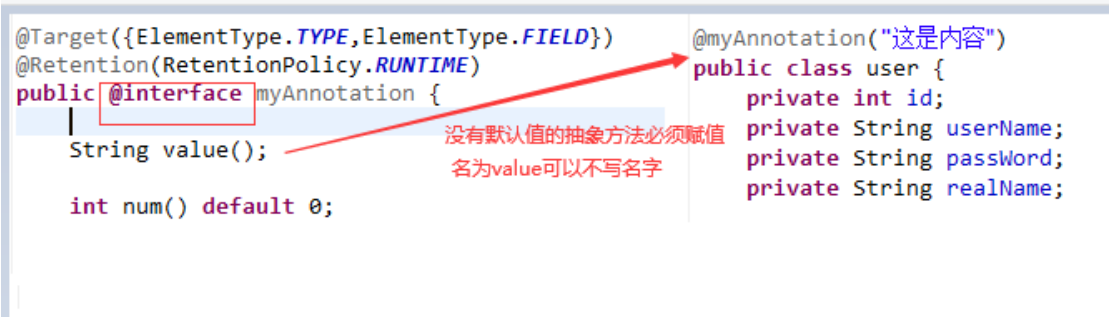
Spring 整合 structs2

1. Spring 整合 structs2,同整合 mybatis 一样，就是将创建核心对象的权力交给 spring 管理，由 spring 来创建
2. Structs2 的核心对象是 action,且是简单对象，所以可以直接用 bean 标签来管理。



Spring 注解

1. 注解是一种特殊的接口**@interface** 默认继承 Annotation 接口，内部的抽象方法作为可赋值的属性，含有默认值的可以不赋值，方法名为 value 的赋值时可以不写方法名。



```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Cache {

    //缓存key
    String  key();


    //缓存时间
    long cacheTime() default 1;

    //缓存时间单位
    TimeUnit timeUnit() default TimeUnit.SECONDS;
}
```

```java
@Cache(key = "user")
public user getUser(Integer id) {
    System.out.println("获取用户方法被调用，id为："
    return users[id];
}
```

2. 注解只是一个可赋值的标识，必须配合封装好的代码才能起作用。
3. 封装好的代码中，利用反射，通过 `getAnnotation` 得到注解的类型和值，注解的作用域，被注解修饰的类或方法，来完成一系列操作。

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Cache {
    //缓存key
    String  key();
    //缓存时间
    long cacheTime() default 1;
    //缓存时间单位
    TimeUnit timeUnit() default TimeUnit.SECONDS;
}
```

自定义缓存注解

```java
public class cacheBean {
    private user[] users = { new user(1, "张三", "23", "123"),
            new user(3, "王五", "25", "345"), new user(4, "赵六
            new user(6, "王八", "28", "678"), new user(7, "老九

    @Cache(key = "user")
    public user getUser(Integer id) {
        System.out.println("获取用户方法被调用，id为：" + id);
        return users[id];
    }
```

在有返回值的方法前使用自定义注解

```java
@Test
public void testCache() throws InterruptedException {
    cacheBean cacheBean = new cacheBean();
    Object user1 = CacheUtils.invokeMethod(cacheBean, "getUser", 0);

    //Thread.sleep(2000);
    Object user2 = CacheUtils.invokeMethod(cacheBean, "getUser", 0);
    System.out.println(user1);
    System.out.println(user2);
}
```

以工具类的形式调用方法，这个可能和cacheBean.getUser()不同

```
private static Map<String, Object> cacheMap = new ConcurrentHashMap<>();

private CacheUtils() {
}
```

处理注解的工具类，利用反射调用方法，将上一个结果存储在静态类中，下一次直接走缓存

```
执行当前指定对象的指定方法
public static Object invokeMethod(Object obj, String methodName, Object... params) {
    Class<?> objClass = obj.getClass();
    Object result = null;

    try {
        Method method;
        if (params.length > 0) {
            Class<?>[] classArr = new Class[params.length];
            Object[] valueArr = new Object[params.length];
            for (int i = 0; i < params.length; i++) {
                classArr[i] = params[i].getClass();
                valueArr[i] = params[i];
            }
            method = objClass.getDeclaredMethod(methodName, classArr);

            // 获取缓存注解
            Cache cacheAnnotation = method.getAnnotation(Cache.class);
            // 先判断注解是否为空
            if (cacheAnnotation != null) {
                // 方法有参数，以第一个参数为小key。
                Object paramsKey = params[0];
                // 获取大key
                String key = cacheAnnotation.key();
```

4. 通过在 spring 配置文件中开启注解扫描，用注解的方式来代替配置文件中繁杂的配置，

前置条件：必须在工厂配置文件中完成注解扫描

```
<context:component-scan base-package="com.baizhi"/>
```

如@component 来代替 bean 标签，@autowire 来代替 property，完成属性注入，并且省去了公开的 set 方法，@Transactional 来代替声明式事务）（AOP）原先的一堆配置，而且遵从局部优先原则（即如果 A 类上有事务,A 类中的方法上也有事务，优先使用方法上的事务）

@Component注解　通用组件创建注解

作用：用来负责对象的创建　==========>　<bean id="" class="">

修饰范围：只能用在类上

注意：默认使用这个注解在工厂中创建的对象的唯一标识为 类名首字母小写　UserDAOImpl　====>userDAOImpl

value属性作用：用来指定创建的对象在工厂中唯一标识 推荐:存在接口接口首字母小写|不存在使用默认

@Repository　作用:一般用来创建DAO中组件的注解

@Service　作用:一般用来创建Service中组件的注解

@Controller　作用:一般用来创建Action中组件中注解

.控制对象在工厂中创建次数

a.配置文件修改　<bean id="" class="" scope="singleton|prototype">

b.注解如何控制

@Scope

作用:　用来指定对象的创建次数默认单例

修饰范围:只能加在类上

value属性：singleton 单例 默认|prototype(多例)

注意：在管理struts的action时必须加入@Scope注解值必须为prototype

5. 自定义注解实现，就是实现@interface 接口，然后定义它的反射解析类即可，见本人的

demo