## Spring框架引言

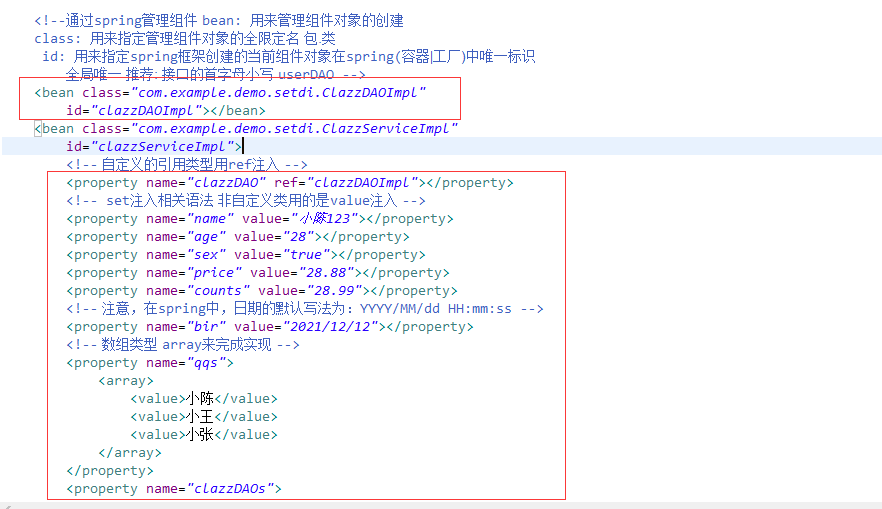
Spring不是替换某项技术，而是对项目中的组件(大部分是类，如service里的类)进行集中管理的工具（创建、使用、销毁）,以spring管理的方式代替了原先new的方式，Spring就是一个管理工具。

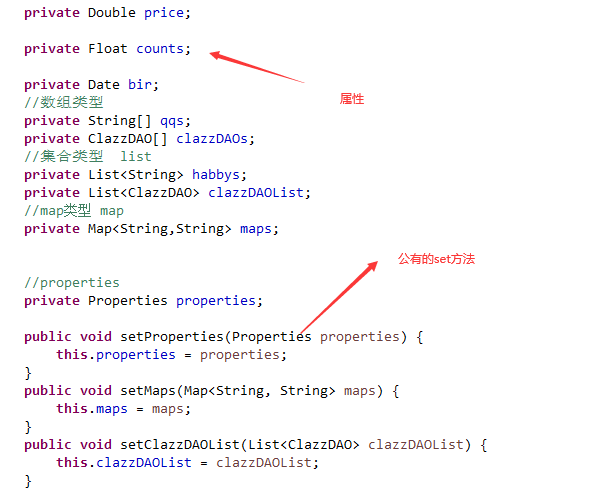
最简单的spring只是简单的几个spring核心jar包和一个xml，xml里通过bean标签里的class参数指定类的全限定名，id为这个类做唯一标识，

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。







### 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。

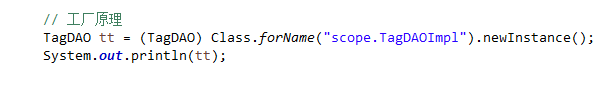
## Spring工厂特性

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。

1. 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



1. 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。
3. Spring的好处（主要是相对传统的new的方式来说）
4. 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。
5. 降低内存使用率：默认的单例共享内存的方式来取代new。

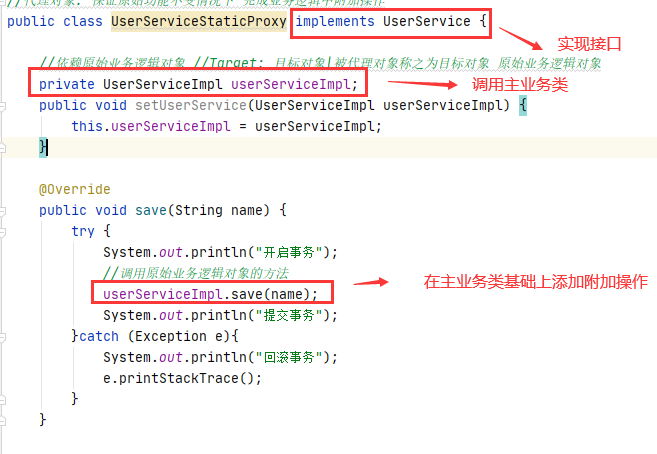
#### Spring AOP

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。

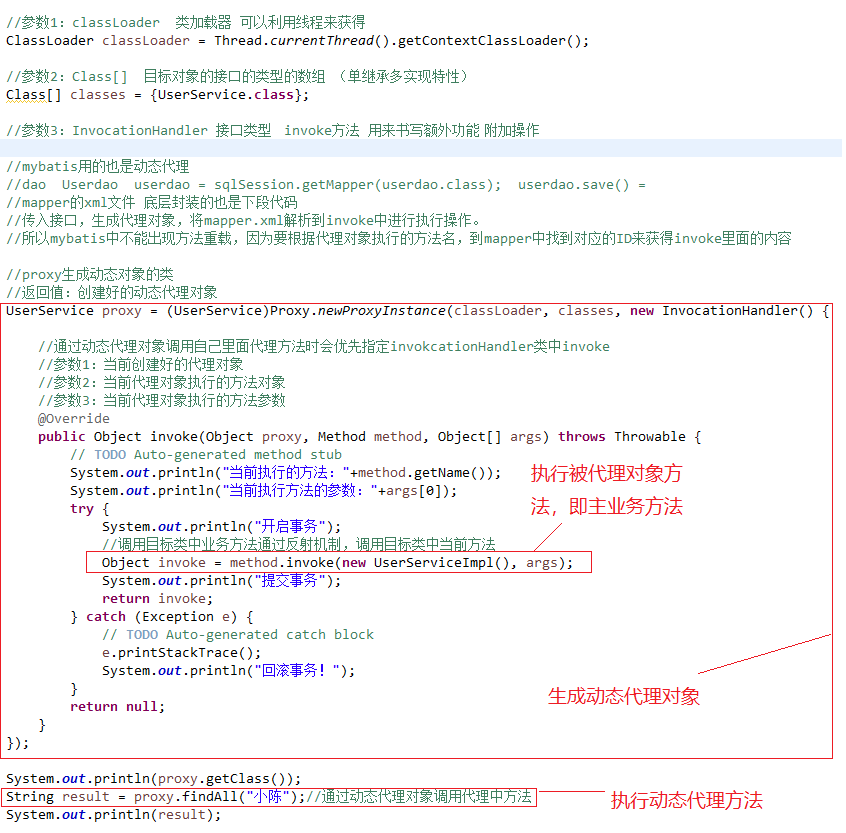
1. 首先明确什么是代理？

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。

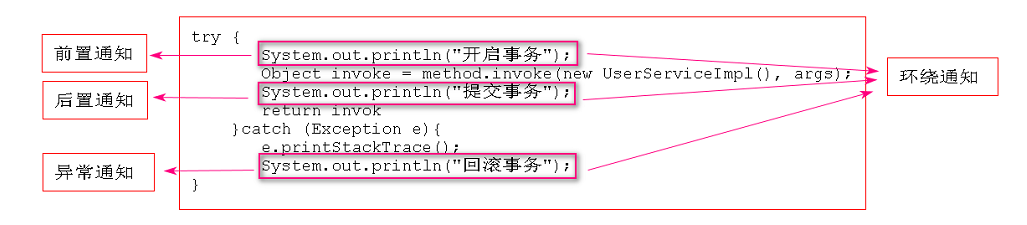
所以我们可以这么理解代理，和主业务类用同一个接口，调用主业务类的方法并在此基础上实现附加操作，叫做代理类，手动创建的叫静态代理，由代码动态创建的叫动态代理。具体demo见本人代码或操作文档。



1. 动态代理的具体理解
2. 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
3. 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：



1. 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
2. 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）



1. 把被代理对象的被代理方法叫做切入点
2. 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
3. 所以AOP的实现只需要传入切入点和通知即可。
4. 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。
5. 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。
6. 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。
7. AOP的具体实现方式

最原始的实现方式：

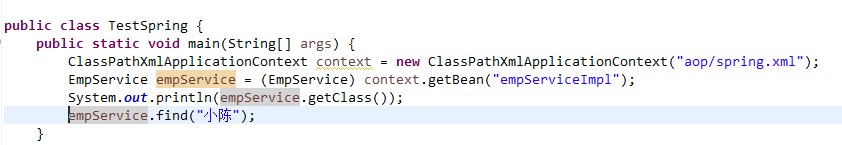
1. 实现通知的接口，完成附加操作的添加



1. 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）

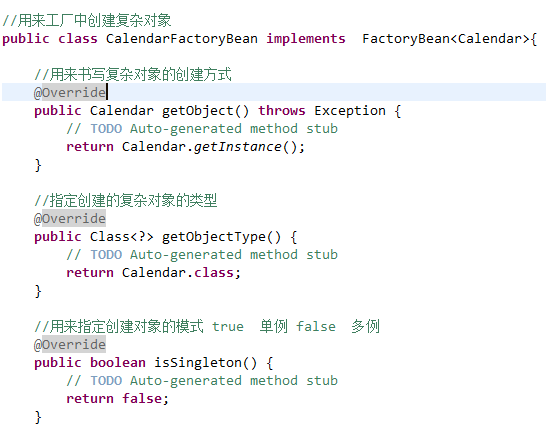


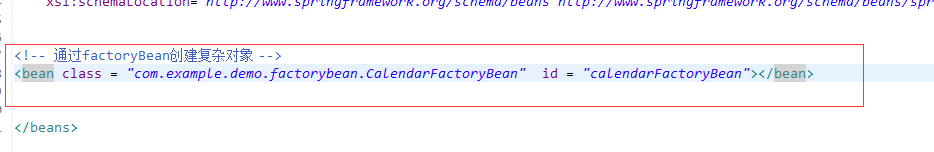
1. 然后对已经配置了AOP切面的方法的调用，其实调用的是底层生成的代理对象，并不是真正调用的原对象。

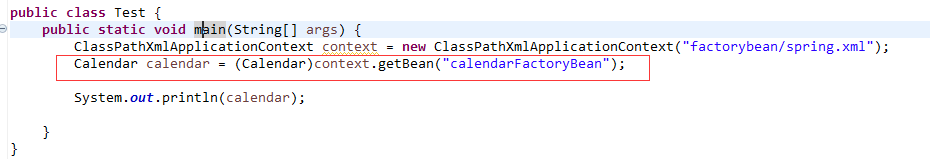




1. 环绕通知，异常通知等通知，见本人demo
2. **Bean标签的补充→复杂对象的创建**
3. 简单对象：可以通过new创建的叫简单对象，如实体，service等，可以直接用bean标签管理
4. 复杂对象：不能通过new创建的叫复杂对象，如接口（Connection等），抽象类（Calendar、MessageDisgest），不可以直接交给Bean标签进行管理。
5. 复杂对象IOC管理：手动创建该类的管理类A，实现FactoryBean<T>，泛型中填入复杂对象，并重写对应的方法（创建方式、复杂对象类型，单例或多例创建），然后A就可以用Bean标签管理，复杂类型就可以和简单类型一样使用了。



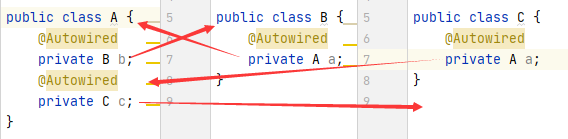




1. 面试重点：循环依赖问题。
2. 一个单例的Bean生命周期为：
3. 初始化化→创建原始对象，并赋予初始值
4. 实例化调用的对象→先查看单例池中有没有调用对象，如果有就拿来用，如果没有，将其实例化并放入单例池（一级缓存）
5. 填充属性
6. 如果有AOP进行AOP代理
7. 将代理对象或实例化后的对象放入单例池中。
8. 但是这种流程在面对循环依赖时就会出现死锁的情况，例如A中调用B，B中调用A的问题。A去实例化B，B又去实例化A，就造成了死锁。



1. Spring采用以下机制来解决循环引用的问题。
2. AOP的各级缓存都是以MAP形式实现的
3. 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
4. 流程图见PDF文件，下面对其进行相关·流程的解释。
5. 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
6. 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：
7. 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；
8. 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。
9. 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。
10. 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。
11. 下面通过举例的方式来描述Spring循环引用的过程。

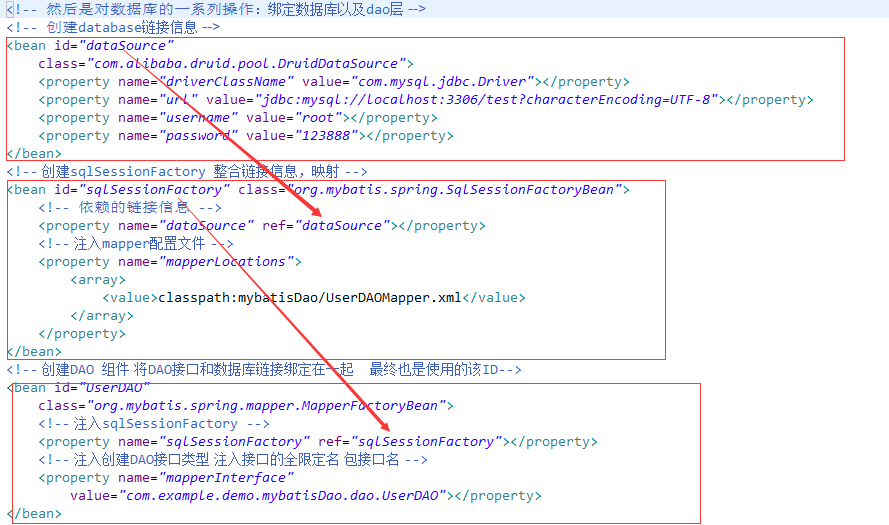


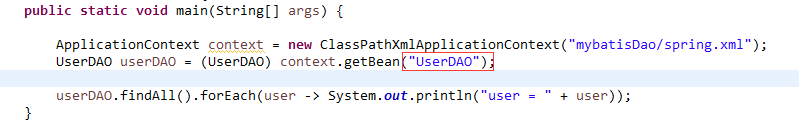
1. 首先创建A对象，Set结构中放入A对象的一个实例，表示正在创建中；并在三级缓存中整合lamuda表达式放入A对象的及其他信息
2. 实例化对象B，首先在单例池中寻找B的实例，然后并没有找到，然后去二级缓存中寻找，B的实例，没有找到，所以就进入class B进行创建
3. 创建B对象，Set结构放入B对象的一个实例，lamuda整合初始化对象其它相关信息放入三级缓存；实例化对象A，首先从单例池中寻找A的实例，没有找到，去二级缓存中寻找A的实例，没有找到，Set结构中发现存在A的实例，表明发生了循环依赖问题，从三级缓存中取出A的lamuda表达式，因为A没有切面，所以直接将A的半初始化对象放入到二级缓存中，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。
4. 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。
5. 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
6. 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。
7. A对象创建完毕，且B、C创建完毕。
8. 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

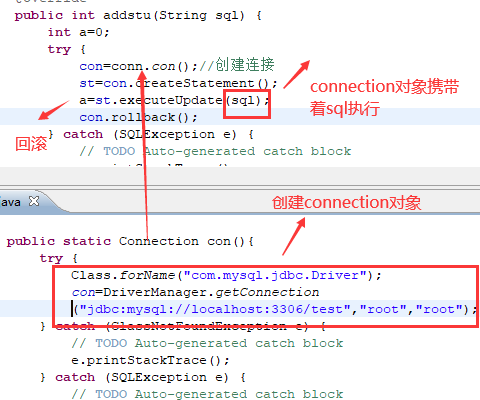
1. spring整合mybatis具体思路就是:将mybatis管理对象的能力交给spring,由spring统一管理，而mybatis所有的组件都来源于SQLSessionFactory，所以接管该对象就接管了mybatis的管理组件权力。
2. SQLSessionFactory是接口（复杂对象），所以不能直接用bean标签管理，要实现factoryBean<T>接口并加载mybatis配置文件，用bean标签来管理，由于上述步骤是重复不变的代码，mybatis公司就整合成新的jar包，提供简单对象SQLSessionFactoryBean,来减少代码量，但是这次数据源文件（填链接用户密码的文件）和mapper文件必须分开加载，这也是springboot加载数据库的雏形。



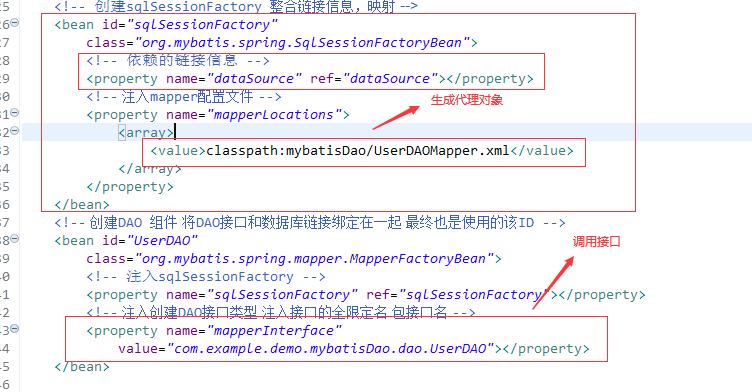


## mybatis和事务的交互

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。

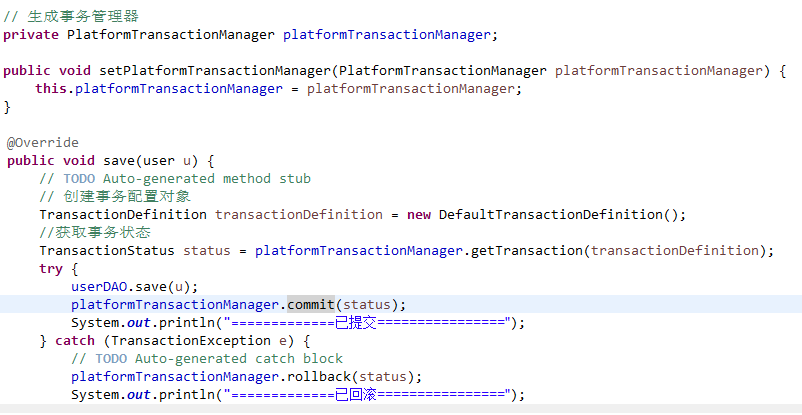


1. 由于mybatis使用的是动态代理，所以service层的connection对象无法显式的传送到dao层，很容易导致同一个线程service层new的connection对象和mybatis动态代理产生的对象不是同一个，即dao层出错，connection回滚，但是service层使用的是另一个connection对象，并不能一起回滚。所以spring提供了bean标签来管理事务保证同一线程链接对象（connection）的一致性。



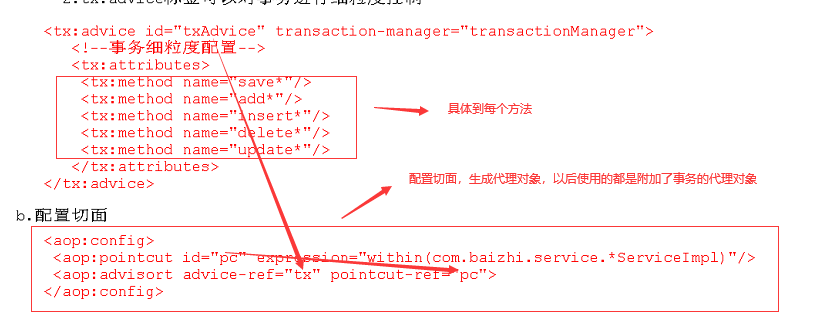
3.mybatis做了一个类*DataSourceTransactionManager，*全局的事务管理器，用来保证业务层当前线程使用连接对象（接口）和DAO层实现连接对象一致，即connection一致。这也是@transaction的雏形，但是注意这个标签是为了保证同一个线程上两个connection 对象的一致性，事务是JBDC层面的东西



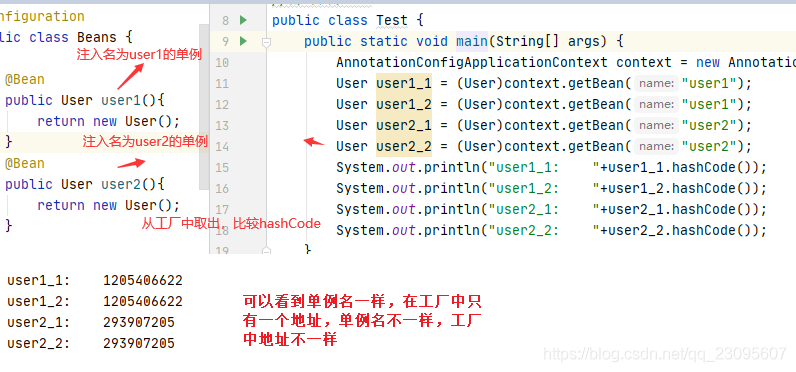


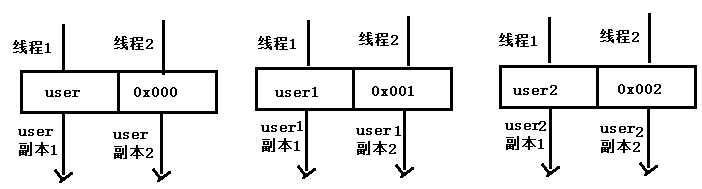
1. 起初我们需要为每一个方法编写事务，即上图所示，为每一个需要事务的方法手动的进行commit和rollback操作，称为编程式事务处理。由于代码重复冗余问题，我们就交给AOP利用反射生成代理对象来做，如下图，我们把这个事务叫做：声明式事务处理。

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



1. 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。
2. 此外事务还有读写（readonly）和异常性（rollback-for && no-rollback-for=）和超时性（timeout）
3. 面试重点：事务的AOP特性、传播特性以及异步特性（详见CSDN）
4. 基础知识
5. 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
6. 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。
7. 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
8. 所以事务只能在同一个线程中传播
9. this.a()就相当于直接调用a(),this指的是本实例

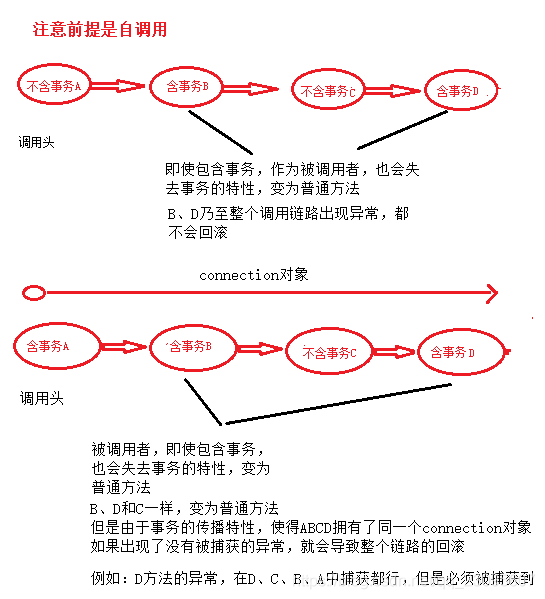


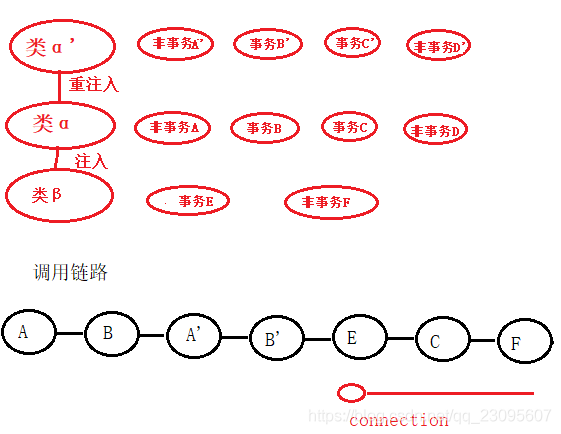


1. AOP的自调用、重注入和异步调用：
2. 类在实现AOP代理后，注入使用的就是代理对象，而不是原对象了
3. 本类重新注入和注入别的类的实例，都是引入了新的的实例对象
4. 同一实例的自调用，被调用方法会被抹去aop特性而走原始对象，不同实例间的调用，AOP特性不会受到影响
5. 类中存在重新注入又存在异步调用的时候，启动时会报错，建议把异步调用放在没有重新注入的类中



1. 事务的自调用、重注入以及异步调用：
2. 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
3. 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
4. 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
5. 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
6. 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
7. 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
8. 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
9. ⑥类中实现异步的前提是不能有重注入，见演示④
10. 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
11. 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





（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。

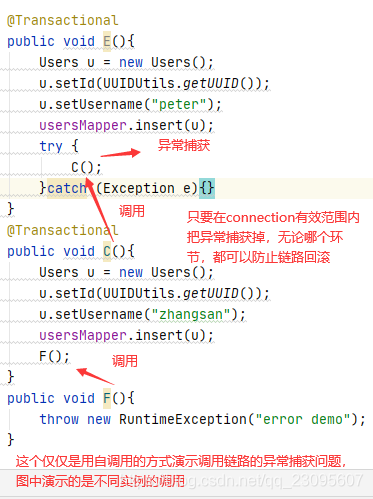
（7）.所以A到B’无论哪个环节出现异常都不会回滚，因为事务都没生效，connection也不一致。

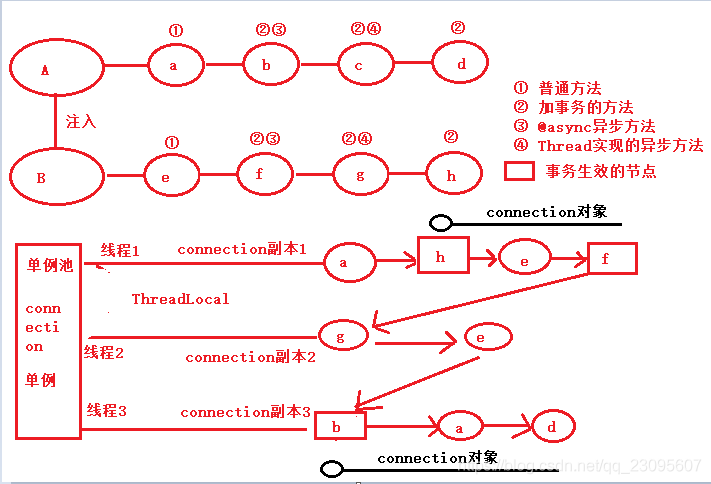
（8）.E到F无论哪个环节出现异常，且E到F没有捕获（try{}catch(){}）,E到F整个链路都会回滚。

（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。





（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

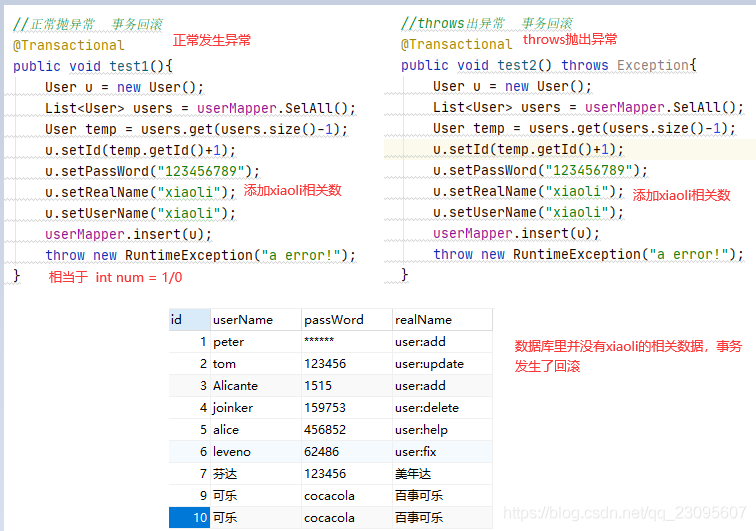
（9）.e调用b，属于实例间调用，@async是用AOP做的，事务也是AOP，所以全部生效，b将在另一个线程中开启事务，且将connection对象传递下去。

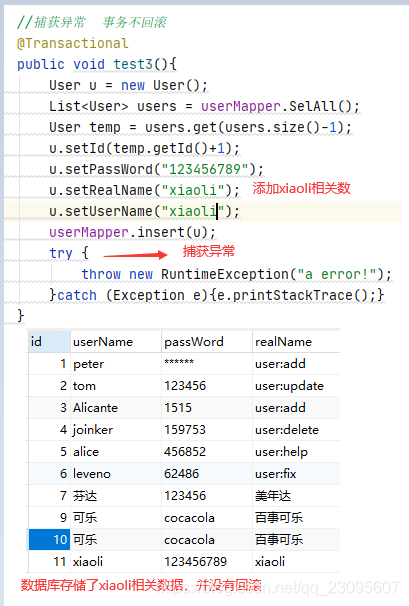
（10）.b调用a，属于自调用，a的事务失效，但是b的事务生效，所以connection将传播至a，a、b connection一致

（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。

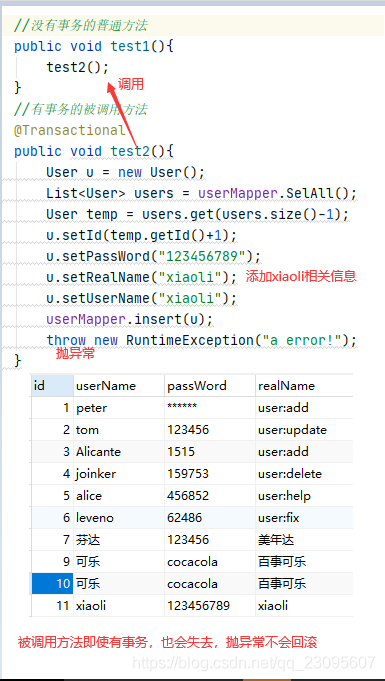
（12）.所以，a、g、e即使出错也不会发生回滚，根据异常节点链路上游中最近的事务生效的节点以及之前进行捕获，才能保证connection对象不会回退，h、e、f中，h、f为事务生效的节点，所以，f出现异常，必须在f中捕获才不会回滚，e出现异常，可以在e和h中捕获，h中出现异常，必须在h中捕获才不会回滚。同理，b、a、d段也是这么处理。

1. 事务的相关演示：
2. 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

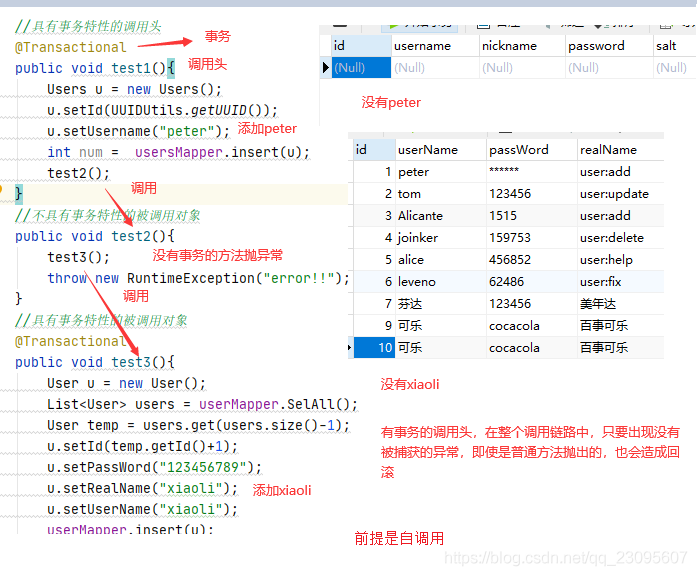


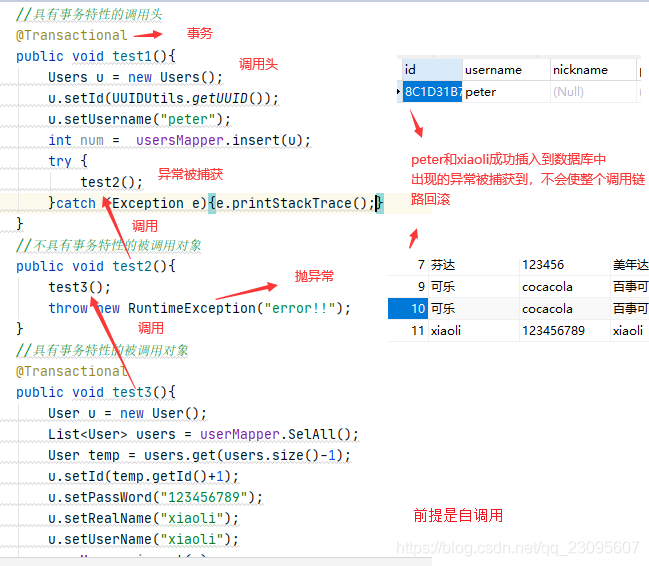


1. 演示②：自调用，被调用方的事务特性将会失去

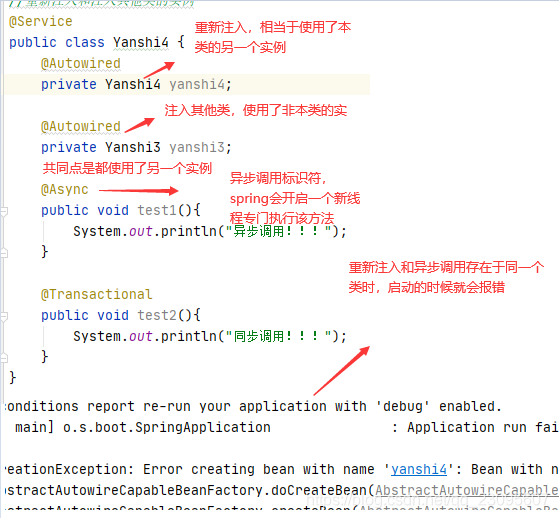


1. 演示③：含有事务的调用头调用链路的异常问题

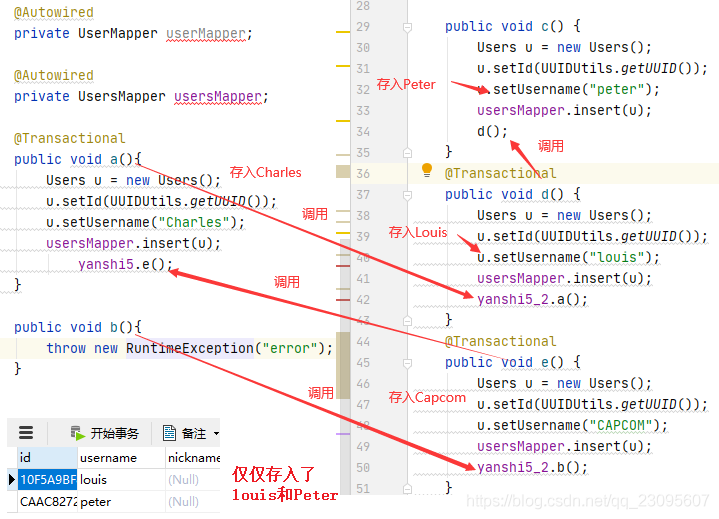


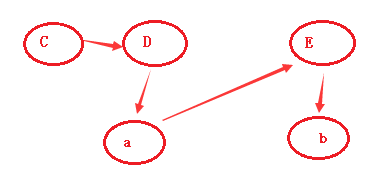


1. 演示④：异步调用+重注入出错



1. 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

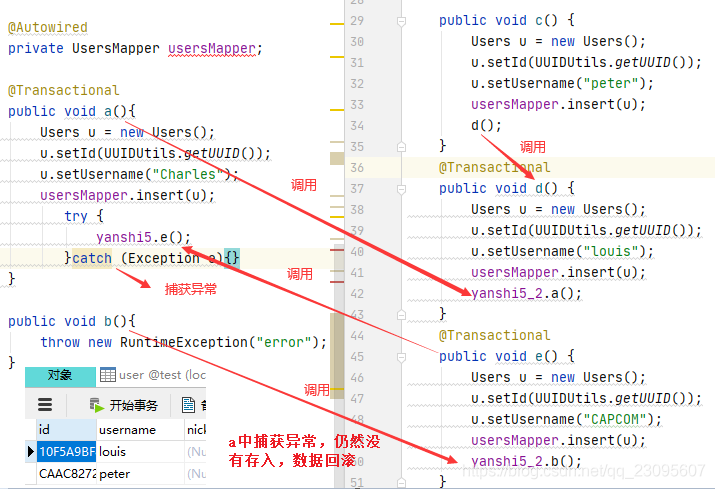
（2）.d调用a，是实例间调用，a事务生效，并开始将connection对象向后传

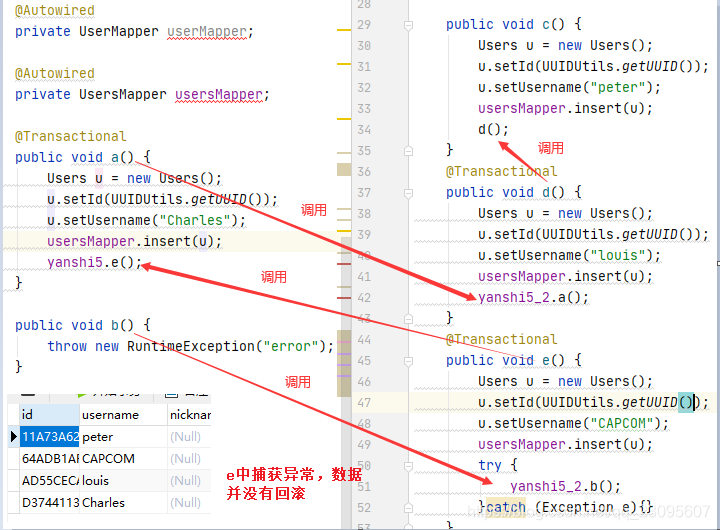
（3）.a调用e，是实例间调用，e事务生效，并和左边test1的connection一致

（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

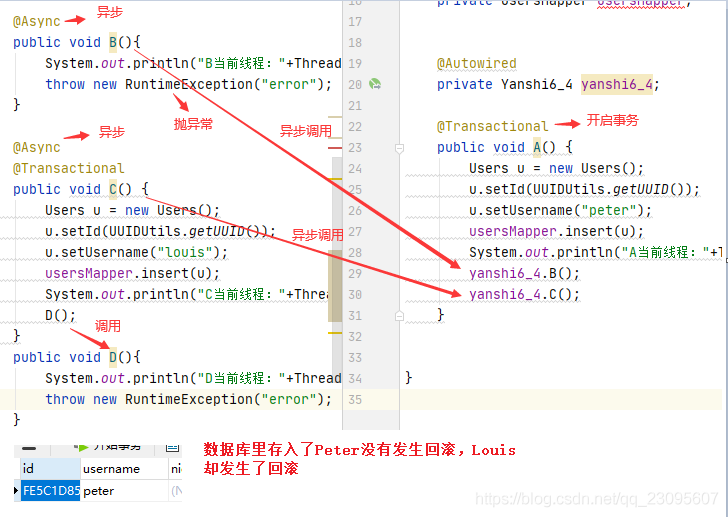
（5）.由于a事务的生效以及事务的传播，使得a、b和e connection一致，且抛出了异常没有被捕获到，所以connection对象发生了回滚，即a、b以及e发生了回滚，没有存入数据库，但是c和td不会发生回滚，因为不在一个connection对象上。

（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。

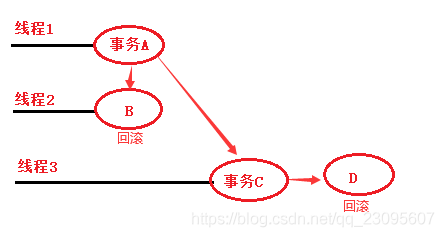




1. 演示6：异步调用下，事务的传播及回滚问题



上述流程可以简化为

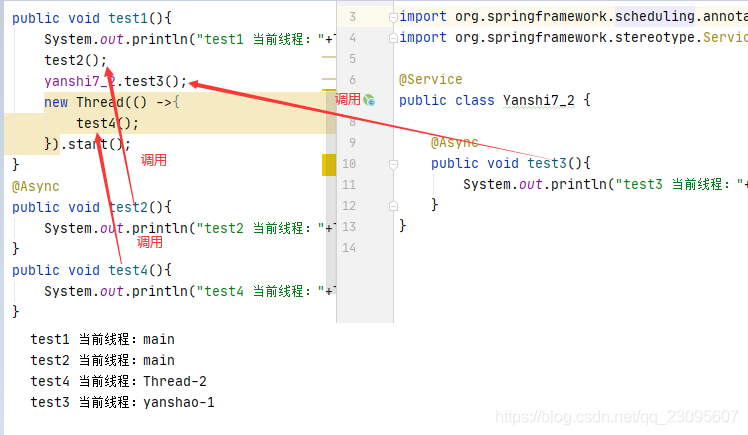


（1）.A拥有事务，会将connection向后传递，如果是同步调用的话，B会与A的connection一致，B抛出异常A回滚，但是异步调用会阻断connection的传递，A与B不再共用一个connection对象，B抛异常并不会导致A的回滚

（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

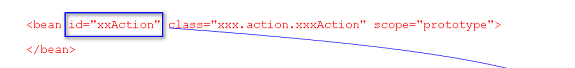
1. 演示7：@async同步调用失效，异步调用生效



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。

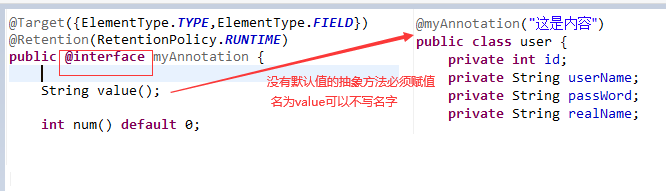
## Spring整合structs2

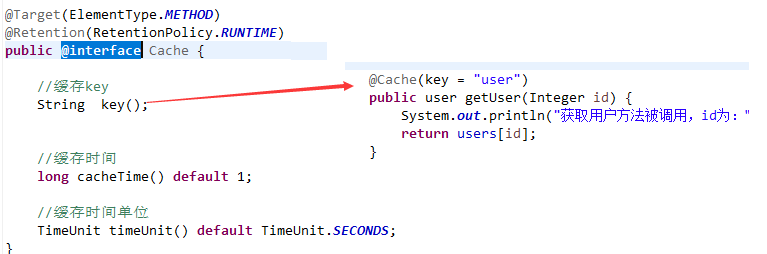
1. Spring整合structs2,同整合mybatis一样，就是将创建核心对象的权力交给spring管理，由spring来创建
2. Structs2的核心对象是action,且是简单对象，所以可以直接用bean标签来管理。



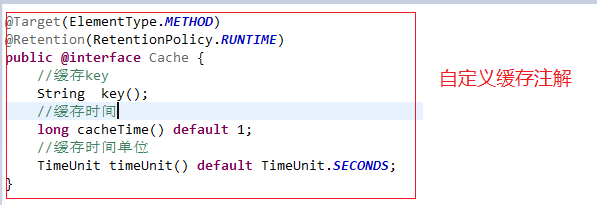
## Spring注解

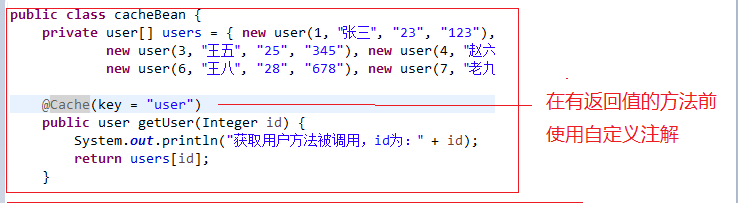
1. 注解是一种特殊的接口**@interface**默认继承Annotation接口，内部的抽象方法作为可赋值的属性，含有默认值的可以不赋值，方法名为value的赋值时可以不写方法名。

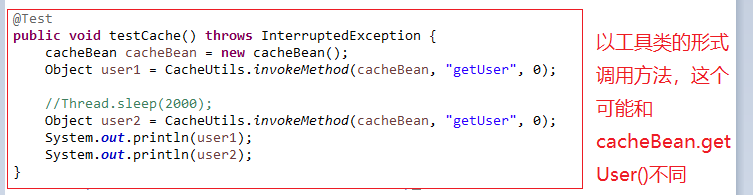




1. 注解只是一个可赋值的标识，必须配合封装好的代码才能起作用。
2. 封装好的代码中，利用反射，通过getAnnotation得到注解的类型和值，注解的作用域，被注解修饰的类或方法，来完成一系列操作。





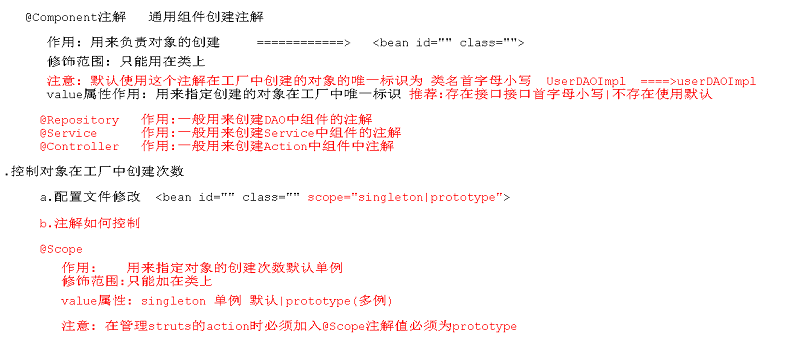




1. 通过在spring配置文件中开启注解扫描，用注解的方式来代替配置文件中繁杂的配置，



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）



1. 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,详细的原理见CSDN及GitHub上反射的笔记