tcc-transaction-spring模块下面的tcc-transaction.xml

<bean id="compensableTransactionAspect" class="org.mengyun.tcctransaction.spring.ConfigurableTransactionAspect"  
 init-method="init">  
 <property name="transactionConfigurator" ref="transactionConfigurator"/>  
</bean>

在使用tcc-transaction的项目中引入tcc-transaction.xml到类路径中，例如：在web.xml中

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath\*:config/spring/local/appcontext-\*.xml,classpath:tcc-transaction.xml,classpath:tcc-transaction-dubbo.xml  
 </param-value>  
</context-param>

在tcc-transaction.xml中引入了2个注解（@Compensable）拦截器

<bean id="compensableTransactionAspect" class="org.mengyun.tcctransaction.spring.ConfigurableTransactionAspect"  
 init-method="init">  
 <property name="transactionConfigurator" ref="transactionConfigurator"/>  
</bean>  
  
<bean id="resourceCoordinatorAspect" class="org.mengyun.tcctransaction.spring.ConfigurableCoordinatorAspect"  
 init-method="init">  
 <property name="transactionConfigurator" ref="transactionConfigurator"/>  
</bean>

ResourceCoordinatorAspect 资源协调器切面

CompensableTransactionAspect 可补偿事务切面

public Object interceptCompensableMethod(ProceedingJoinPoint pjp) throws Throwable {  
  
 CompensableMethodContext compensableMethodContext = new CompensableMethodContext(pjp);  
  
 boolean isTransactionActive = transactionManager.isTransactionActive();  
  
 if (!TransactionUtils.*isLegalTransactionContext*(isTransactionActive, compensableMethodContext)) {  
 throw new SystemException("no active compensable transaction while propagation is mandatory for method " + compensableMethodContext.getMethod().getName());  
 }  
  
 switch (compensableMethodContext.getMethodRole(isTransactionActive)) {  
 case *ROOT*:  
 return rootMethodProceed(compensableMethodContext);  
 case *PROVIDER*:  
 return providerMethodProceed(compensableMethodContext);  
 default:  
 return pjp.proceed();  
 }  
}

根据方法角色不同进行不同处理

根方法

收集延迟取消异常

获取被@UniqueIdentity注解标记的参数 这个参数就是TransactionXid

Transaction transaction = new Transaction(uniqueIdentify,TransactionType.*ROOT*);

将这个transaction持久化（我这里是用的mysql）

持久化成功后，将transaction放入缓存

private Cache<Xid, Transaction> transactionXidCompensableTransactionCache;

然后注册，就是将transaction放到当前线程的队列中

private static final ThreadLocal<Deque<Transaction>> *CURRENT* = new ThreadLocal<Deque<Transaction>>();

返回transaction事务对象

然后执行业务方法

如果执行业务方法异常了，异常处理，首先检查是不是延迟取消异常，延迟取消异常需要更新TCC库，从当前线程取出当前事务修改事务状态为cancelling，更新成功后，更新缓存。如果数据库更新失败，删除缓存的事务。需要执行事务回滚操作

如果是异步事务，进行异步回滚，所有的参与者回滚

terminator.invoke(new TransactionContext(xid, TransactionStatus.*CANCELLING*.getId()), cancelInvocationContext, transactionContextEditorClass);

下面这个方法好像是远程调用

public Object invoke(TransactionContext transactionContext, InvocationContext invocationContext, Class<? extends TransactionContextEditor> transactionContextEditorClass) {  
  
  
 if (StringUtils.*isNotEmpty*(invocationContext.getMethodName())) {  
  
 try {  
  
 Object target = FactoryBuilder.*factoryOf*(invocationContext.getTargetClass()).getInstance();  
  
 Method method = null;  
  
 method = target.getClass().getMethod(invocationContext.getMethodName(), invocationContext.getParameterTypes());  
  
 FactoryBuilder.*factoryOf*(transactionContextEditorClass).getInstance().set(transactionContext, target, method, invocationContext.getArgs());  
  
 return method.invoke(target, invocationContext.getArgs());  
  
 } catch (Exception e) {  
 throw new SystemException(e);  
 }  
 }  
 return null;  
}

如果业务功能执行成功没有异常，需要确认，

提供者方法provider

事务状态为trying、confirming、cancelling

Trying状态

根据事务上下文新生成一个事务，写入数据库，并放入当前线程，然后执行业务逻辑

如果是confirming状态

根据事务上下文中的事务id查询事务（当然是先到缓存中获取，再到数据库中获取）

修改事务状态并重新注册

然后执行事务提交业务操作。

{

从当前线程获取事务，修改事务状态为confirming，更新到数据库

然后提交事务{

所有的参与者都要提交

}

添加事务完成后，删除数据库中的事务信息。

}

其他（normal，consumer） 直接执行业务不做扩展，不做分布式事务控制

方法角色

public MethodRole getMethodRole(boolean isTransactionActive) {  
 if ((propagation.equals(Propagation.*REQUIRED*) && !isTransactionActive && transactionContext == null) ||  
 propagation.equals(Propagation.*REQUIRES\_NEW*)) {  
 return MethodRole.*ROOT*;  
 } else if ((propagation.equals(Propagation.*REQUIRED*) || propagation.equals(Propagation.*MANDATORY*)) && !isTransactionActive && transactionContext != null) {  
 return MethodRole.*PROVIDER*;  
 } else {  
 return MethodRole.*NORMAL*;  
 }  
}

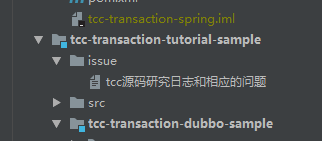
root角色: 传播特性为required、不活跃事务、事务上下文为null 或者传播特性为required\_new

provider角色: 传播特性为required、mandatory（托管），且不活跃事务，且事务上下文不为null

normal角色：其他就为normal角色

public static Method getCompensableMethod(ProceedingJoinPoint pjp) {  
 Method method = ((MethodSignature) (pjp.getSignature())).getMethod();  
  
 if (method.getAnnotation(Compensable.class) == null) {  
 try {  
 method = pjp.getTarget().getClass().getMethod(method.getName(), method.getParameterTypes());  
 } catch (NoSuchMethodException e) {   
 return null;  
 }  
 }  
 return method;  
}

tcc调试日志在我的github上tcc-transaction-learn仓库有，其中有两个疑问。



1、消费端获取的服务端@Compensable注解中的属性confirmMethod和cancelMethod怎么都是try方法record？在哪里设置的？

2、确认阶段，消费方通过参与者调用了record的try方法，在服务方RCI拦截没有起作用？这是什么操作？

Tcc原理解析的文档

https://www.jianshu.com/p/f10d2065c26c