



下载APP



## 35 | 分布式事务：使用 Nacos+Seata 实现 TCC 补偿模式

2022-03-04 姚秋辰

《Spring Cloud 微服务项目实战》

课程介绍 &gt;



讲述：姚秋辰

时长 20:38 大小 18.90M



你好，我是姚秋辰。

上节课我们落地了一套 Seata AT 方案，要我说呢，AT 绝对是最省心的分布式事务方案，一个注解搞定一切。今天这节课，我们来加一点难度，从 Easy 模式直接拉到 Hard 模式，看一个巨复杂的分布式事务方案：Seata TCC。

领资料

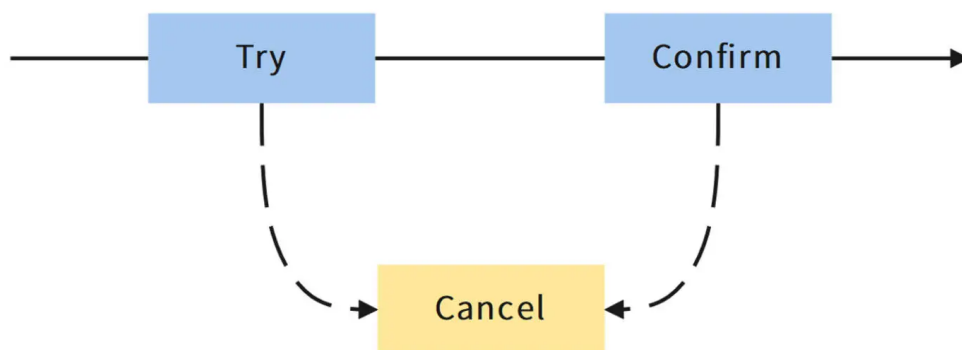
说 TCC 复杂，那是相对于 AT 来讲的。在 AT 模式下，你通过一个注解就能搞定所有事情，不需要对业务层代码进行任何修改。TCC 难就难在它的实现方式上，它是一个基于“补偿模式”的解决方案。补偿的意思就是，你需要通过编写业务逻辑代码实现事务控制。



那 TCC 是如何通过代码来控制事务状态的呢？这就要说到 TCC 的三阶段事务模型了。

## TCC 事务模型

TCC 名字里这三个字母分别是三个单词的首字母缩写，从前到后分别是 Try、Confirm 和 Cancel，这三个单词分别对应了 TCC 模式的三个执行阶段，每一个阶段都是独立的本地事务。



Try 阶段完成的工作是**预定操作资源（Prepare）**，说白了就是“占座”的意思，在正式开始执行业务逻辑之前，先把要操作的资源占上座。

Confirm 阶段完成的工作是**执行主要业务逻辑（Commit）**，它类似于事务的 Commit 操作。在这个阶段中，你可以对 Try 阶段锁定的资源进行各种 CRUD 操作。如果 Confirm 阶段被成功执行，就宣告当前分支事务提交成功。

Cancel 阶段的工作是**事务回滚（Rollback）**，它类似于事务的 Rollback 操作。在这个阶段中，你可没有 AT 方案的 undo\_log 帮你做自动回滚，你需要通过业务代码，对 Confirm 阶段执行的操作进行人工回滚。

我用一个考研占座的例子帮你理解 TCC 的工作流程。话说学校有一个专给考研学生准备的不打烊的考研复习教室，一座难求。如果你想要用 TCC 的方式坐定一个位子，那么第一步就是要执行 Try 操作，比如往座位上放上一块板砖，那这个座位就被你预定住了，后面来的人发现座位上面有块砖就去找其他座位了。第二步是 Confirm 阶段，这时候需要把板砖拿走，然后本尊坐在位子上，到这里 TCC 事务就算成功执行了。

如果 Try 阶段无法锁定资源，或者 Confirm 阶段发生异常，那么整个全局事务就会回滚，这就触发了第三步 Cancel，你需要对 Try 步骤中锁定的资源进行释放，于是乎，这块砖在 Cancel 阶段被移走了，座位回到了 TCC 执行前的状态。

从这个例子可以看出，TCC 的每一个步骤都需要你通过执行业务代码来实现。那接下来，让我带你去实战项目中落地一个简单的 TCC 案例，近距离感受下 Hard 模式的开发体验。


## 实现 TCC

这次我们依然选择优惠券模板删除这个场景作为 TCC 的落地案例，我将在上节课的 AT 模式的基础之上，对 Template 服务做一番改造，将 deleteTemplate 接口改造为 TCC 风格。

前面咱提到过，TCC 是由 Try-Confirm-Cancel 三个部分组成的，这三个部分怎么来定义呢？我先来写一个 TCC 风格的接口，你一看就明白了。

## 注册 TCC 接口

为了方便你阅读代码，我在 Template 服务里单独定义了一个新的接口，取名为 CouponTemplateServiceTCC，它继承了 CouponTemplateService 这个接口。

 复制代码

```
1 @LocalTCC
2 public interface CouponTemplateServiceTCC extends CouponTemplateService {
3
4     @TwoPhaseBusinessAction(
5         name = "deleteTemplateTCC",
6         commitMethod = "deleteTemplateCommit",
7         rollbackMethod = "deleteTemplateCancel"
8     )
9     void deleteTemplateTCC(@BusinessActionContextParameter(paramName = "id") L
10
11     void deleteTemplateCommit(BusinessActionContext context);
12
13     void deleteTemplateCancel(BusinessActionContext context);
14 }
```

在这段代码中，我使用了两个 TCC 的核心注解：LocalTCC 和 TwoPhaseBusinessAction。

其中 @LocalTCC 注解被用来修饰实现了二阶段提交的本地 TCC 接口，而 @TwoPhaseBusinessAction 注解标识当前方法使用 TCC 模式管理事务提交。

在 `TwoPhaseBusinessAction` 注解内，我通过 `name` 属性给当前事务注册了一个全局唯一的 TCC bean name，然后分别使用 `commitMethod` 和 `rollbackMethod` 指定了它在 `Confirm` 阶段和 `Cancel` 阶段所要执行的方法。Try 阶段所要执行的方法，便是被 `@TwoPhaseBusinessAction` 所修饰的 `deleteTemplateTCC` 方法了。

你一定注意到了我在 `deleteTemplateCommit` 和 `deleteTemplateCancel` 这两个方法中使用了一个特殊的入参 `BusinessActionContext`，你可以使用它传递查询参数。在 TCC 模式下，查询参数将作为 `BusinessActionContext` 的一部分，在事务上下文中进行传递。

如果你对 TCC 注解的底层源码感兴趣，我推荐你从 `GlobalTransactionScanner` 这个类的 `wrapIfNecessary` 方法开始研究。它通过 `TCCBeanParserUtils` 工具类来判断当前资源是否为 TCC 的实现类，如果是 TCC 自动代理的话，就生成一个 `TccActionInterceptor` 作为当前 bean 对象的事务拦截器。

[复制代码](#)

```
1 if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName, applicationContext)) {
2     //TCC interceptor, proxy bean of sofa:reference/dubbo:reference, and Local
3     interceptor = new TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(
4 }
```

接口定义完成后，我们将 `CouponTemplateServiceImpl` 的接口类指向刚定义好的 `CouponTemplateServiceTCC` 方法，接下来就可以写具体实现了，按照 TCC 三阶段的顺序，我们先从一阶段 `Prepare` 写起。

## 编写一阶段 Prepare 逻辑

在一阶段 `Prepare` 的过程中，我们执行的是 Try 逻辑，它的目标是“锁定”优惠券模板资源。为了达成这个目标，我们需要对 `coupon_template` 数据库做一个小修改，引入一个名为 `locked` 的变量，用来标记当前资源是否被锁定。你可以直接执行下面的 SQL 语句添加这个属性。

[复制代码](#)

```
1 alter table coupon_template
2     add locked tinyint(1) default 0 null;
```



在 CouponTemplate 类中，我们也要加上 locked 属性。

[复制代码](#)

```
1 @Column(name = "locked", nullable = false)
2 private Boolean locked;
```

有了 locked 字段，我们就可以在 Try 阶段借助它来锁定券模板了。我们先来看一下简化版的资源锁定代码吧。

[复制代码](#)

```
1 @Override
2 @Transactional
3 public void deleteTemplateTCC(Long id) {
4     CouponTemplate filter = CouponTemplate.builder()
5         .available(true)
6         .locked(false)
7         .id(id)
8         .build();
9
10    CouponTemplate template = templateDao.findAll(Example.of(filter))
11        .stream().findFirst()
12        .orElseThrow(() -> new RuntimeException("Template Not Found"));
13
14    template.setLocked(true);
15    templateDao.save(template);
16 }
```

在这段代码中，我在通过 ID 查找优惠券的同时，添加了两个查询限定条件来筛选未被锁定且状态为 available 的券模板。如果查到了符合条件的记录，我会将其 locked 状态置为 true。

在正式的线上业务中，Try 方法的资源锁定逻辑会更加复杂。我举一个例子，就拿转账来说吧，如果张三要向李四转账 30 元，在 TCC 模式下这 30 元会被“锁定”并计入冻结金额中，我们在“锁定”资源的同时还需要记录是“谁”冻结了这部分金额。比如你可以在生成锁定记录的时候将转账交易号也一并记下来，这个交易号就是我们前面说的那个“谁”，这样你就知道这些金额是被哪笔交易锁定的了。这样一来，当你执行回滚逻辑将金额从“冻结余额”里释放的时候，就不会错误地释放其他转账请求锁定的金额了。

接下来我们去看下二阶段 Commit 的执行逻辑。

## 编写二阶段 Commit 逻辑

二阶段 Commit 就是 TCC 中的 Confirm 阶段，只要 TCC 框架执行到了 Commit 逻辑，那么就代表各个分支事务已经成功执行了 Try 逻辑。我们在 Commit 阶段执行的是主体业务逻辑，即删除优惠券，但是别忘了你还要将 Try 阶段的资源锁定解除掉。

在下面的代码中，我们放心大胆地直接读取了指定 ID 的优惠券，不用担心 ID 不存在，因为 ID 不存在的话，在 Try 阶段就会抛出异常，TCC 会转而执行 Rollback 方法，压根进不到 Commit 阶段。读取到 Template 对象之后，我们分别设置 locked=false，available=true。

[复制代码](#)

```
1 @Override
2 @Transactional
3 public void deleteTemplateCommit(BusinessActionContext context) {
4     Long id = Long.parseLong(context.getActionContext("id").toString());
5
6     CouponTemplate template = templateDao.findById(id).get();
7
8     template.setLocked(false);
9     template.setAvailable(true);
10    templateDao.save(template);
11
12    log.info("TCC committed");
13 }
```

现在你已经完成了二阶段 Commit，最后让我们来编写 Rollback 的逻辑吧。

## 编写二阶段 Rollback 逻辑

二阶段 Rollback 对应的是 TCC 中的 Cancel 阶段，如果在 Try 或者 Confirm 阶段发生了异常，就会触发 TCC 全局事务回滚，Seata Server 会将 Rollback 指令发送给每一个分支事务。

在下面这段简化的 Rollback 代码中，我们读取了 Template 对象，并通过将 locked 设置为 false 的方式对资源进行解锁。

[复制代码](#)

```
1 @Override
```

```
2 @Transactional
3 public void deleteTemplateCancel(BusinessActionContext context) {
4     Long id = Long.parseLong(context.getActionContext("id").toString());
5     Optional<CouponTemplate> templateOption = templateDao.findById(id);
6
7     if (templateOption.isPresent()) {
8         CouponTemplate template = templateOption.get();
9         template.setLocked(false);
10        templateDao.save(template);
11    }
12    log.info("TCC cancel");
13 }
```

在线上业务中，Cancel 方法只能释放由当前 TCC 事务在 Try 阶段锁定的资源，这就要求你在 Try 阶段记录资源锁定方的信息，并在 Confirm 和 Cancel 阶段对这个信息进行判断。

你知道为什么我在 Cancel 里特意加了一段逻辑，判断 Template 是否存在吗？这就要提到 TCC 的空回滚了。

## TCC 空回滚

所谓空回滚，是在没有执行 Try 方法的情况下，TC 下发了回滚指令并执行了 Cancel 逻辑。

那么在什么情况下会出现空回滚呢？比如某个分支事务的一阶段 Try 方法因为网络不可用发生了 Timeout 异常，或者 Try 阶段执行失败，这时候 TM 端会判定全局事务回滚，TC 端向各个分支事务发送 Cancel 指令，这就产生了一次空回滚。

处理空回滚的正确的做法是，在 Cancel 阶段，你应当先判断一阶段 Try 有没有执行成功。示例程序中的判断方式比较简单，我先是判断资源是否已经被锁定，再执行释放操作。如果资源未被锁定或者压根不存在，你可以认为 Try 阶段没有执行成功，这时在 Cancel 阶段直接返回成功即可。

更为完善的一种做法是，引入独立的事务控制表，在 Try 阶段中将 XID 和分支事务 ID 落表保存，如果 Cancel 阶段查不到事务控制记录，那么就说明 Try 阶段未被执行。同理，Cancel 阶段执行成功后，也可以在事务控制表中记录回滚状态，这样做是为了防止另一个 TCC 的坑，“倒悬”。

## TCC 倒悬

倒悬又被叫做“悬挂”，它是指 TCC 三个阶段没有按照先后顺序执行。我们就拿刚讲过的空回滚的例子来说吧，如果 Try 方法因为网络问题卡在了网关层，导致锁定资源超时，这时 Cancel 阶段执行了一次空回滚，到目前为止一切正常。但回滚之后，原先超时的 Try 方法经过网关层的重试，又被后台服务接收到了，这就产生了一次倒悬场景，即一阶段 Try 在二阶段回滚之后被触发。

在倒悬的情况下，整个事务已经被全局回滚，那么如果你再执行一次 Try 操作，当前资源将被长期锁定，这就造成了一种类似死锁的局面。解法很简单，你可以利用事务控制表记录二阶段执行状态，并在 Try 阶段中检查该状态，如果二阶段回滚完毕，那么就跳过一阶段 Try。

到这里，我们就落地了一套 TCC 业务，下面让我们来回顾下这节课的重要内容吧。

## 总结

TCC 相比于 AT 而言，代码开发量至少要 double，它以开发量为代价，换取了事务的高度可控性。不过我仍然不建议头脑一热就上 TCC 方案，因为 TCC 非常考验开发团队对业务的理解深度，为什么这样说呢？一个重要原因是，你需要把串行的业务逻辑拆分成 Try-Confirm-Cancel 三个不同的阶段执行，如何设计资源的锁定流程、如果不同资源间有关联性又怎么锁定、回滚的反向补偿逻辑等等，你需要对业务流程的每一个步骤了如指掌，才能设计出高效的 TCC 流程。

还有需要特别注意的一点是幂等性，接口幂等性是保证数据一致性的重要前提。在大厂中通常有框架层面的幂等组件，或者幂等性服务供你调用，对于中小业务来说，通过本地事务控制表来确保幂等性是一种简单有效的低成本方案。

## 思考题

通过这两节课我们落地了 AT 和 TCC 方案，Seata 里还有一个 SAGA 方案，你能举一反三自选 SAGA 并落地几个小 demo 吗？

到这里，我们就学完了这个专栏的最后一节正课内容。欢迎你把这个专栏分享给更多对 Spring Cloud 感兴趣的朋友。我是姚秋辰，我们结束语再见！



分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

赞 0

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 分布式事务：使用 Nacos+Seata 实现AT模式

## 精选留言 (3)

写留言



奔跑的蚂蚁

2022-03-04

对于tcc 和 at 如果涉及到分布式事务 小公司 和 团队人数少的公司选择哪个好呢？还有别的更简单的方法吗



奔跑的蚂蚁

2022-03-04

老师能讲下 微服务的 后端接口 版本升级怎么控制的嘛，多个版本兼容怎么做的呢？是通过网关转发到不同的服务上吗。



peter

2022-03-04

请教老师几个问题：

Q1：接口类指向是什么意思？

文中有一句话“将 CouponTemplateServiceImpl 的接口类指向刚定义好的 CouponTemplate ServiceTCC 方法”，怎么指向的？

Q2：阿里这种级别的公司，其入口是怎么做的？...

展开 ∨



