MiniCoreBank项目文档

MiniCoreBank

1. 结构

结构上实现最基本功能共有三个主要业务模块和三个辅助模块。三个主要业务模块分别是银行用户服务、信用服务以及订单服务。其中，银行用户服务包括用户服务（BankUserService）与信用卡卡管理服务（CreditCardService）。想法主要是User在这个项目中是和Credit Card一对一绑定的，因此卡的管理请求与用户账户管理的请求服务放在一个模块是合理的。

三个辅助模块分别是记录服务、数据模型以及运行模块。其中数据模型主要定义DAO层。运行模块目前有集成测试并可以运行整个项目。记录服务主要是因为各个服务均会有相应的记录请求，例如创建订单的记录，修改额度的记录等，但目前没有为银行用户服务模块提供记录服务。

2. 实现的一些注意点

2.1 工程setting

1. dependency的循环调用问题

由于各模块之间可能会互相调用服务，我出现过循环dependency的情况，这种情况一般是调用服务的设计出现问题，需要重新思考这种服务调用是否必要或者这个项目结构是否有问题并及时调整。

2. Sofa module的bean搜索以及模块隔离

需要添加相应的sofa module name以及搜索sofa service的xml文件，有时候我会忘记。

3. 对服务的集成测试

专门设置出run模块来进行各子模块的集成测试，由于服务若要成功运行，需要有对应的spring context，因此这种情况无法进行简单的单元测试，而是需要集成测试，并且需要加载spring context。

2.2 关键处理

1. 一锁二判三更新四释放

   Select for update可以在事务中锁住行记录，我原本认为一些更新操作结束后便可以释放锁，但是倘若出现回退，事务依然会出现意想不到的结果，对于资金敏感的业务这个问题后果会很严重，因此依然遵循整个事务中都将需要使用的数据先锁住，判断和更新结束后再等事务最终结束后释放锁。

2. 关于行级锁，最初的我的理解以为是@Transactional annotation的设定加上@Lock就足够了，实际上要在数据层就进行设置，而不是在业务层。如果不在数据层上锁，实际测试的业务层依然会出现问题。MySQL在主键确定时会使用行级锁。参考：MySQL读锁及其应用场景 MySQL update与insert时的行级锁
3. 幂等性，除了主键唯一id以外要注意在业务层需要有request_id，主要原因是用户调用时由于网络等外界因素可能会有多余重发。可以在数据层查找是否有duplicate来判断这个请求需不需要处理，拒绝处理已经处理过的请求。并且对于已经处理过的请求，应该返回上一次处理的结果，即按照正常处理完毕的结果返回，而不是以报错的形式返回。因为在用户调用的角度，已经处理过的也已经是有结果的。
4. 此前思考过如果有非常极端的并发情况出现应该做和应对的问题，例如，有并发的修改卡状态与修改卡额度的请求，从计算机的角度，哪一个请求到来就应该率先处理哪一个，而另一个只能依据上一个修改的结果继续修改。在我的例子中，如果修改卡状态的请求先到，那么修改卡额度就应该基于上一个的修改结果，反之亦然。但是这是我们业务实现的角度，倘若在用户的角度会有些许不同，在这个例子中，用户可能在修改额度之前先看了自己的卡状态，在看到自己卡状态时active的情况下他提交修改额度请求，那么假如有上述并发，用户会感到奇怪，即自己请求的时候明明卡状态正确，为何修改额度还是失败了，这时应该给用户展现业务处理的，例如应该告知其修改额度失败的原因是卡状态被修改了。

2.3 Spring，JPA实现细节

1. Order是Mysql的关键字，因此要避免将Order作为Mysql的表名。
2. 关于嵌套事务，如果将子事务设置为Propagation.Nested，那么会记录save point，如果出现异常会会滚到save point。参考：Differences between requires_new and nested propagation in Spring transactions
3. 需要在@Query中使用对象，参考这个链接 Using named parameters for method when I use @Query
4. 注意使用saveAndFlush，例如当我们创建订单的记录之前需要先保存订单时，就应该用saveAndFlush方法而不是save方法。参考：Difference Between save() and saveAndFlush() in Spring Data JPA | Baeldung
5. @Transactional会默认为RuntimeException rollback，如果有其他类型的exception或者自定义的exception也需要rollback，那么需要rollbackfor。

2.4 Java实现规范

1. 表必备三字段：id，create_time，update_time，create_time表明创建记录的初试时间，update_time则是在每次update的时候都会更新。
2. 数据库操作永远是逻辑删除而非物理删除，物理删除应该在定期任务中进行统一处理，以防止在业务中出现误删无法rollback的情况。

3. 不得使用外键与级联，一切外键概念必须在应用层解决。这主要考虑分布式应用外键的约束主要在数据层实现，但是一旦应用为分布式这种约束就会因为有一致性等同步问题。
4. 外部可调用的方法要注意方法变量和返回变量状态为包装类型，主要原因是防止默认数据造成不必要的调用结果。例如某方法返回整数类型，但方法返回的是int，结果int的默认初始数字为0，结果调用者就用0来继续其业务处理，造成损失。倘若设置为包装状态，则返回null，那么调用者可以及时发现并处理异常，就无实际损失。
5. 永远都不要使用select *，而是应该将需要使用的field明确标明，否则在一些调用中，如果后来table又加入了新的field会出现报错。
6. 使用slfj4等logger框架。
7. 不要使用java.sql.Date，应该使用java.util.Date

2.5 工程tip

1. IDEA在工程出现无法理解的行为时，可以点击navibar中的File->Invalid caches清空缓存，应对一些虽然在实现上均正确，但是工程仍然有问题的情况。例如明明语法均正确，却仍然出现找不到package或编译错误等。
2. 在合理范畴下尽量多commit，多code review，这样可以及时发现问题以及进行回滚。

3. 个人反思与总结

- 在工程实现上可以参考MVP理论，即Minimum viable project。在一开始熟悉sofaboot的时候，根据tutorial创建toy project，在tutorial的基础上自己稍微修改一些，既可以克服上手未知框架的畏难情绪，也可以获取一些自己动手操作的经验。同样在写这个project之前，也可以先把最基本的调用逻辑写通，因为一个庞大的项目除了结构设计工作以外实际上是相似业务的堆叠，一个基本逻辑的跑通往往意味着诸多业务可以在此基础上实现。

参考：Making sense of MVP (Minimum Viable Product) - YouTube

- 我在和mentor沟通的时候一开始是想要积累一些问题一并提问解决，但我觉得在理想情况下，更好的解决方案是有需要解答的问题就应该即时发布出来，mentor在自己有空的时候可以帮忙解答，这样做的好处是可以更及时解决问题，有的时候工程上的问题与学术上的问题不同，有经验的mentor可以在很短时间内帮你解决一个可能要思考很久的问题，借助mentor的经验积累自身的经验也是很有必要的，但我也觉得需要询问一下mentor他们最初是怎么解决或者怎么看出这个问题和此前遇到的问题有相似之处的，工程上的问题解决有时就是一个类推的过程。
- 好记性真的不如烂笔头。我觉得对于工程中遇到的问题以及解决方案需要及时记录下来，一方面是之后自己有可能会遇到，另一方面别人也有可能遇到，而工程中的问题如果此前遇到过，解决的方法往往是类似的，能节省大量时间，但前提是要能记得之前咋解决的，思路是什么。这时候人的记性很有限，这种经验的积累需要记录。另一点则是，人的想法在写下来的时候能够被优化、固定、趋于成熟，真正成为一个idea的时刻不是在脑子中诞生的时候，而是用笔记录下的时刻。

参考：Leslie Lamport: Thinking Above the Code - YouTube

- 这一次的工程实现没有即时地使用code review与pull request等git功能，现在我觉得这是一个即时与同事沟通的方法，他们可以把建议即时给出，不过说实话被别人review代码是一件会让我紧张的事情，尽管我很清楚这是一件有利于工程质量的事，但这不妨碍我的紧张，需要克服这种紧张，更高质量的代码是第一位的，别人指出代码中的问题确实不会让自己心情愉悦，毕竟这说明自己的工作有问题，但同时也要记住it's all about business, nothing personal. 要做的就是修正并提升，无限进步，力求没有错误。