第4章 Transaction Management 事务管理

> So what was true then is true now. Don’t waste your time living someone else’s life. Don’t try to emulate the people who came before you to the exclusion of everything else, contorting into a shape that doesn’t fit.
> It takes too much mental effort – effort that should be dedicated to creating and building. You’ll waste precious time trying to rewire your every thought, and, in the mean time, you won’t be fooling anybody.

2023.09.16

- What is a transaction? 事务的概念
- container-managed transactions 容器管理的事务
- isolation and database locking 隔离和数据库锁
- updaring multiple databases 更新多个数据库

1. 事务的概念

> 转账：
>
> 1. 把钱从第一个人的账户里面扣除
> 2. 把钱加到第二个人的账户里面
>   如果第一个动作成功，第二个动作失败，我们就需要回滚第一个动作；这两个动作在我们看来是**原子性**的。
>
> 本质上就是你要告诉我，事务从哪里开始，到哪里结束?
>
> 转账：
>
> 1. A和B同时转账给C，转10元；C原本有100元
> 2. 如果是同时转账，结果应该是120，但实际可能是110——事务的**隔离性**

定义
在java EE应用程序中，事务是一系列必须全部成功完成的操作，否则每个操作中的所有更改都将被撤消。事务以提交或回滚结束。

事务的四个特性（ACID）

1. 原子性（Atomicity）： 事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
2. 一致性（Consistency）： 在事务开始和完成时，数据都必须保持一致状态。
3. 隔离性（Isolation）： 数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。
4. 持久性（Durability）： 事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

从编程的角度来说，我们要考虑的就是**原子性**和**隔离性**。剩下两件事交给数据库就好。

在spring中，我们都希望以 `annotation` （声明）的方式来实现事务管理（而不是自己编码）。

2. 容器管理的事务

在Spring中，容器管理事务是指Spring容器负责管理应用程序的事务生命周期，以确保事务的正确执行和一致性。它提供了一种声明式的、基于配置的方式来管理事务，使得开发人员可以将精力集中在业务逻辑上，而不必关心事务的细节。

2.1 事务传播行为

- `@Transaction(propagation = Propagation.REQUIRED)`： 如果当前线程上面有事务就使用当前的事务，如果没有就创建一个事务；注意！在进入和退出方法的时候都会做检查！
- `@Transaction(propagation = Propagation.NOT_SUPPORT)`： 容器不为这个方法开启事务；你有事务就挂起，你没事务就算了
- `@Transaction(propagation = Propagation.REQUIRED_NEW)`： 不管是否存在事务，都创建一个新的事务，原来的挂起，新的执行完毕之后，继续执行老的事务
- `@Transaction(propagation = Propagation.MANDATORY)`： 必须在一个已有的事务里面执行，否则抛出异常
- `@Transaction(propagation = Propagation.NEVER)`： 必须在一个没有的事务里面执行，否则抛出异常
- `@Transaction(propagation = Propagation.SUPPORT)`： 如果其他的bean调用这个方法，在其他的bean里面声明了事务就使用事务，如果其他的bean里面没有声明事务，那就不用事务（主打一个佛系）

举例

1. 在下面的例子中，如果 `method1` 和 `method2` 都是 `REQUIRED` ，那么 `method2` 会加入到 `method1` 的事务中，如果 `method2` 抛出异常，那么 `method1` 也会回滚。

2. 当程序执行完 `method2` 的时候，还会做一次检查，发现诶这个事务就不是你开启的，所以就不会提交。当 `method1` 执行完毕的时候，才会提交。
3. 如果 `method2` 上面的注解是 `REQUIRES_NEW` ，那么 `method2` 就会开启一个新的事务， `method1` 和 `method2` 就是两个事务了。 `method2` 结束的时候，它就直接把自己的事务提交掉了。**如果 `method2` 抛出异常，那么 `method1` 的事务不会回滚。**

```
// bean1
@Transactional(propagation = Propagation.REQUIRED)
public void method1() {
    // do something
    bean2.method2();
    // do something
}

// bean2
@Transactional(propagation = Propagation.REQUIRED)
public void method2() {
    // do something
}
```

2.1.1 回滚

spring如何回滚呢?

- 抛出 `RuntimeException` 或者 `Error` 的时候，spring会自动回滚

```java
// BankServiceImpl.java
@Service
public class BankServiceImpl implements BankService {
    @Autowired
    private BankDao bankDao;

    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void transfer(String from, String to, int money) {
        bankDao.withdraw(from, money);

        // int result = 10 / 0;    // 1. 在deposit之前

        try {
            bankDao.deposit(to, money);
        } catch (Exception e) {
            e.printStackTrace();    // 打印异常
        }

        // int result = 10 / 0;    // 2. 在deposit之后
    }

// BankDaoImpl.java
@Repository
public class BankDaoImpl implements BankDao {
    @Autowired
    private BankRepository bankRepository;

    @Transactional(propagation = Propagation.REQUIRED)
    public void withdraw(String from, int money) {
        Bank bank = bankRepository.findById(from).get();
        bank.setMoney(bank.getMoney() - money);
        bankRepository.save(bank);

        // int result = 10 / 0;
    }

    @Transactional(propagation = Propagation.REQUIRED)
    public void deposit(String to, int money) {
        Bank bank = bankRepository.findById(to).get();
        bank.setMoney(bank.getMoney() + money);
        bankRepository.save(bank);
```

```
        // int result = 10 / 0;
    }
}
```

id	transfer	withdraw	deposit	result
1	result = 10/0 在deposit之前	正常	正常 REQUIRES_NEW	整个事务回滚
2	result = 10/0 在deposit之后	正常	正常 REQUIRES_NEW	deposit成功，withdraw和transfer回滚
3	正常	result = 10/0	正常 REQUIRES_NEW	整个事务回滚 （因为withdraw抛出异常， 一路抛上去没人接收， 所以整个事务回滚）
4	正常	正常	result = 10/0 REQUIRES_NEW	withdraw和transfer成功，deposit回滚 （因为deposit扔出去的错误被catch了， 所以deposit回滚，剩下两个没事）

2.2 事务隔离级别

要理解事务隔离级别，我们需要先了解我们会碰到的一些问题。

2.2.1 一些问题

- **脏读**： 事务T1将某一值修改，然后事务T2读取该值，此后T1因为某种原因撤销对该值的修改，这就导致了T2所读取到的数据是无效的。这就是脏读。
- **不可重复读**：事务T1读取某一数据，事务T2读取并修改了该数据，T1为了对读取值进行检验而再次读取该数据，发现得到了不同的结果。（我把数据读走之后锁死了，不让你动）
- **幻读**：事务A读取值，把读取走的数据锁死了，但是事务B往里面添加了几行数据，这就导致事务A再次读取的时候发现可能多了几行符合条件的。就像幻影一样。（这个时候要锁的不仅仅是你读走的数据，要锁的是整个表格；应该在这个事务的执行期间，我们按照同一个逻辑，读到的东西是相同的）

这三个问题都是事务没有隔离好导致的；综上需要权衡，在性能和隔离程度直接选择。

2.2.2 锁！

- **读锁**：读取锁防止其他事务在事务结束之前更改事务期间读取的数据，从而防止不可重复的读取。**其他事务可以读取数据，但不能写入数据。当前事务也被禁止进行更改。**
- **写锁**：写入锁用于更新。写锁防止其他事务在当前事务完成之前更改数据，但**允许其他事务和当前事务本身进行脏读取。**换句话说，事务可以读取自己未提交的更改。（第一种事务隔离就是解决这个问题）
- **独占写锁**：独占写入锁用于更新。防止其他事务读取或更改数据，直到当前事务完成。它还可以防止其他事务的脏读取。

还有什么处理concurrency的方法呢?

> 离线锁：所谓“离线”指的是flush之前，数据读出来之后，放在了tomcat的内存里；这个时候是离线的；所谓“在线”，就是这个对象实时反映了数据库的数据；但显然OR映射/JDBC都是离线的。

- **乐观离线锁**：保持乐观的状态，默认不会出现冲突（所以数据库就不加锁）。用户写入数据的时候，检查一下读取的数据是否发生了改变，如果发现不一致，那就是说明用户是基于陈旧的数据在进行修改，就会抛出异常。（实现方法：增加一个版本号，表里面加一列，更新之后版本号码加1，校验的时候检查版本号码是否有问题，当然也可以用时间戳）
- **悲观离线锁**：对于冲突这个事情保持悲观太多，也就是认为冲突的概率非常高。当有用户来获取Session的时候就会获取锁，一旦有人在读取或者修改，就无法获取锁。
- **粗粒锁Coarse-Grained Lock** ： 一次性用一个锁，把根数据相关的，比如外键关联的所有的数据全部锁死。实现原理：两组数据通过引用共享同一个版本号码，**通过引用的方法，两个数据引用的是同一个版本号码**（所谓的版本号不是一个数字，而是一个**对象**，这个对象里面有一个数字属性，这个数字就是版本号码）。

2.2.2 隔离级别

- `@Transaction(isolation = Isolation.READ_UNCOMMITTED)`：读取未提交的数据（说穿了就是**啥也不干**，上面三个问题都有）（用的少，因为实在是太垃圾了）
- `@Transaction(isolation = Isolation.READ_COMMITTED)`：读取已经提交的数据（**解决了脏读**，会出现不可重复读还有幻读）SQLSERVER默认。因为读取已提交的数据，未提交的数据不能读，所以避免了脏读的问题。
- `@Transaction(isolation = Isolation.REPEATABLE_READ)`：可重复性读（会出现幻读）MYSQL默认。因为**读走的数据加了锁（行锁）**，这样别的数据就不能改，避免了不可重复性读的问题。但是还是会出现幻读，因为可能会有别的事务添加几行新数据，这就导致可能两次相同的读取在后一次读取的时候发现多出来几行
- `@Transaction(isolation = Isolation.SERILIZABLE)`：串行化执行，最严格的级别。**把整个表格锁死**，幻读也可以避免了！（用的少，因为这样几乎就没有并发了）

2.2.3 更新多个数据库——分布式事务

只要不是同一个数据库，就是分布式事务。

多数据源：两阶段提交协议

1. prepare阶段：去问！（数据库A能不能提交？数据库B能不能提交？）
2. commit/rollback阶段：根据prepare阶段的结果，决定commit还是rollback

实现

通过 `@Transactional` 注解的 `transactionManager` 属性指定事务管理器，从而实现多数据源的事务管理。