# @Transactional 能和 @Async 一起用吗？

springdoc.cn/spring-transactional-async-annotation

2024年4月14日

## 1、简介

本文将带你了解 Spring 中 `@Transactional` 和 `@Async` 注解之间的兼容性。

## 2、 了解 @Transactional 和 @Async

`@Transactional` 注解是 Spring 提供的声明式事务注解。可以让多个业务方法在同一个事务中执行，只有所有方法都正常执行完毕后事务才会提交。如果任何一个方法在调用过程中抛出了异常，那么事务就会回滚。

`@Async` 注解用于执行异步任务，如果从一个线程调用 `@Async` 方法或类，Spring 会使用另一个线程来运行该方法，从而提高执行效率。

在有些情况下，我们需要在代码中同时使用 `@Transactional` 和 `@Async` 来保业务数据的一致性以及性能。

## 3、@Transactional 能和 @Async 一起用吗？

**异步** 和 **事务** 如果使用不当，可能会带来数据不一致等问题。

关于这一点，需要充分了解 Spring 的事务上下文和上下文之间的数据传播。

### 3.1、创建示例应用

本文使用银行的转账功能来说明事务和异步代码的使用。简而言之，是一个转账的场景，从一个账户中扣除资金并将其添加到另一个账户。

我们可以把它想象成数据库操作，比如 `select` 相关账户并 `update` 其资金余额：

```
public void transfer(Long depositorId, Long favoredId, BigDecimal amount) {
    Account depositorAccount = accountRepository.findById(depositorId)
      .orElseThrow(IllegalArgumentException::new);
    Account favoredAccount = accountRepository.findById(favoredId)
      .orElseThrow(IllegalArgumentException::new);

    depositorAccount.setBalance(depositorAccount.getBalance().subtract(amount));
    favoredAccount.setBalance(favoredAccount.getBalance().add(amount));

    accountRepository.save(depositorAccount);
    accountRepository.save(favoredAccount);
}
```

首先使用 `findById()` 查找相关账户，如果给定 ID 的账户不存在，则抛出 `IllegalArgumentException` 异常。

然后，用新金额更新检索到的账户。最后，使用 `CrudRepository` 的 `save()` 方法保存新更新的账户。

在这个简单的示例中，可能会出现一些异常。例如，可能找不到 `favoredAccount`，从而导致异常失败。或者，`save()` 操作完成了 `depositorAccount`，但 `favoredAccount` 却失败了。

由于查询和修改操作都在不同的事务，因此可能会造成数据一致性问题。例如，可能会从一个账户中扣除资金，而没有有效地将其转移给另一个账户。

## 3.2、在 @Async 中调用 @Transactional

如果我们在 `@Async` 方法中调用 `@Transactional` 方法，Spring 就会正确管理事务并传播其上下文，从而确保数据的一致性。

如下，从 `@Async` 中调用注解了 `@Transactional` 的 `transfer()` 方法：

```
@Async
public void transferAsync(Long depositorId, Long favoredId, BigDecimal amount) {
    transfer(depositorId, favoredId, amount);

    // 其他操作，但是和 transfer 不在同一个事务
}

@Transactional
public void transfer(Long depositorId, Long favoredId, BigDecimal amount) {
    Account depositorAccount = accountRepository.findById(depositorId)
      .orElseThrow(IllegalArgumentException::new);
    Account favoredAccount = accountRepository.findById(favoredId)
      .orElseThrow(IllegalArgumentException::new);

    depositorAccount.setBalance(depositorAccount.getBalance().subtract(amount));
    favoredAccount.setBalance(favoredAccount.getBalance().add(amount));

    accountRepository.save(depositorAccount);
    accountRepository.save(favoredAccount);
}
```

`transferAsync()` 方法与调用线程在不同上下文中并行运行，因为它是 `@Async` 注解的。

然后，调用事务 `transfer()` 方法来运行关键的业务逻辑。在这种情况下，Spring 会正确地将 `transferAsync()` 线程上下文传播到 `transfer()`。因此，不会在该交互中丢失任何数据。

`transfer()` 方法定义了一组关键的数据库操作，如果出现异常，这些操作必须回滚。Spring 只处理 `transfer()` 事务，它将 `transfer()` 方法之外的所有代码与事务隔离。因此，只有出现异常时，Spring 才会回滚 `transfer()` 代码。

从 `@Async` 方法中调用 `@Transactional` 可以与调用线程并行执行操作，而不会在特定内部操作中出现数据不一致的情况，从而提高性能。

## 3.3、在 @Transactional 中调用 @Async

Spring 目前使用 `ThreadLocal` 来管理当前线程事务。因此，它不会在应用的不同线程之间共享线程上下文。

因此，如果在 `@Transactional` 方法中调用 `@Async` 方法，那么当前方法和异步方法不在同一个事务中。

在 `transfer()` 中调用注解了 `@Async` 的 `printReceipt()`：

```
@Async
public void transferAsync(Long depositorId, Long favoredId, BigDecimal amount) {
    transfer(depositorId, favoredId, amount);
}

@Transactional
public void transfer(Long depositorId, Long favoredId, BigDecimal amount) {
    Account depositorAccount = accountRepository.findById(depositorId)
      .orElseThrow(IllegalArgumentException::new);
    Account favoredAccount = accountRepository.findById(favoredId)
      .orElseThrow(IllegalArgumentException::new);

    depositorAccount.setBalance(depositorAccount.getBalance().subtract(amount));
    favoredAccount.setBalance(favoredAccount.getBalance().add(amount));

    printReceipt();
    accountRepository.save(depositorAccount);
    accountRepository.save(favoredAccount);
}

@Async public void printReceipt() {
    // 打印转账结果
}
```

`transfer()` 逻辑与之前的相同，但现在我们调用 `printReceipt()` 来打印转账结果。由于 `printReceipt()` 注解了 `@Async`，因此 Spring 会在另一个上下文的不同线程上运行其代码。

问题在于，转账信息取决于整个 `transfer()` 方法的正确执行。此外，`printReceipt()` 和 `transfer()` 代码在不同的线程上运行，数据也不同，这使得应用的行为变得不可预测。例如，可能会打印未成功保存到数据库的转账交易结果。

**因此，为了避免这种数据一致性问题，必须避免从 `@Transactional` 中调用 `@Async` 方法。**

## 3.4、在类级别使用 @Transactional

在类上注解 `@Transactional`，会为类中所有 `public` 方法都创建声明式事务。

在类级别使用 `@Transactional` 注解时，可能会出现与 @Async 注解在同一个方法中混合使用的情况。实际上，我们是创建了一个围绕该方法的事务单元，该方法在与调用线程不同的线程中运行。

```
@Transactional
public class AccountService {
    @Async
    public void transferAsync() {
        //  这是一个异步和事务处理方法
    }

    public void transfer() {
        // 事务方法
    }
}
```

在示例中，`transferAsync()` 方法是事务性的异步方法。因此，它定义了一个事务单元，并在不同的线程上运行。因此，它可用于事务管理，但与调用线程不在同一上下文中。

如果出现异常，`transferAsync()` 中的代码就会回滚，因为它注解了 `@Transactional`。不过，由于该方法也注解了 `@Async`，Spring 不会将调用上下文传播给它。因此，在异常时，Spring 不会回滚 `trasnferAsync()` 之外的任何代码，就像我们调用事务性方法时一样。因此，这与从 `@Transactional` 中调用 `@Async` 一样，存在数据完整性问题。

类级注解便于编写较少的代码来创建一个定义了一系列事务性方法的类。

不过，这种混合事务和异步的行为可能会在排查异常时造成混乱。例如，我们期望在一系列事务方法的调用中，当发生异常时，所有的方法都会回滚。然而，如果该调用中的某个方法具有 `@Async` 注解，行为就可能出乎意料。

## 4、总结

本文从数据完整性的角度介绍了何时可以安全地同时使用 `@Transactional` 和 `@Async` 注解。

一般来说，在 `@Async` 方法中调用 `@Transactional` 可以保证数据完整性，因为 Spring 会正确传播相同的上下文。而从 `@Transactional` 调用 `@Async` 时，可能会导致数据完整性问题。

---

Ref：`https://www.baeldung.com/spring-transactional-async-annotation`