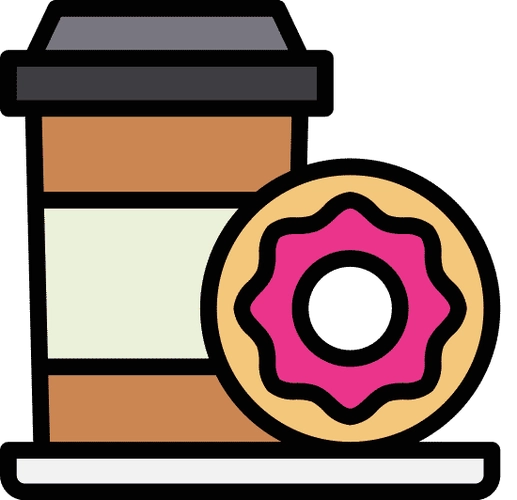
**15+ things you need to know when you want to use Spring @Transactional really well**

As an excellent Java Developer or Architect. Do you really master how to use @Transactional well?

The Transaction is a must to understand and use well Concept which will ensure the integrity of our business data and avoid many unnecessary bugs.

In this article. I will talk about how to use spring annotation @Transactional well on data operation. and ***you could find many good examples take away after reading.***



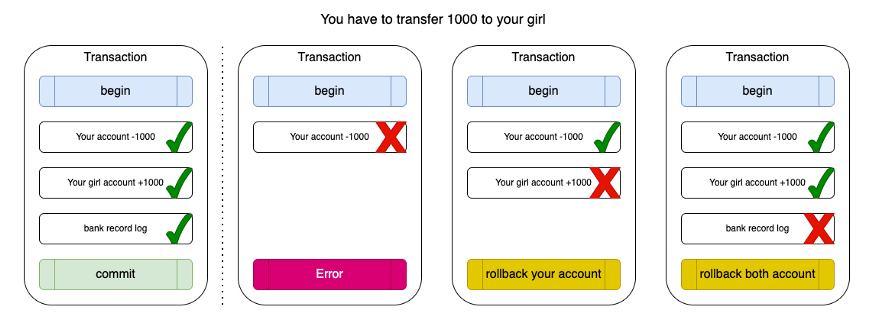
Take aways

**1. Before we start diving into the @Transactional annotation, we first need to understand what exactly is a transaction.**

What is a Transaction?

* A transaction is often defined as an indivisible unit of work. [What Is a Transaction? — The Java EE 5 Tutorial](https://docs.oracle.com/javaee/5/tutorial/doc/bncii.html)
* In java, it's a series of actions that must all complete successfully. If one or more action fails, all the other action must back out leaving the state of the application unchanged. <https://www.baeldung.com/java-transactions>

Let’s explain with a simple example on the diagram:



Simple Transaction Example

So it's about data integrity. We could see **only all the step success. the transaction could commit**. otherwise will fail or rollback to keep the data integrity.

In fact, in the computer field, there are multiple transaction types such as database transactions, message transactions for message middleware (Kafka, Rockmq), etc. And **This article will focus on database transactions.**

**1.1 Usage scenarios for transaction**

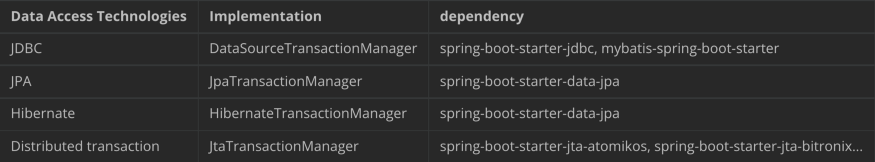
When to use it？

In short. when you try to create/update/delete data in the table in one or more steps of a unit of work. then you better use transactions to keep the data integrity.

**1.2 The transaction mechanism provided by Spring.**

All data access technologies have their own transaction handling mechanisms, and these technologies provide APIs for opening transactions, committing transactions to complete data operations, or rolling back data in the event of an error.

Spring’s transaction mechanism is a unified mechanism for handling transactions for different data access technologies, and Spring’s transaction mechanism provides a **PlatformTransactionManager** interface, Different data access technologies implement their different way from this interface.



**1.2.1 Spring Declarative Transaction**

Spring provides a *@EnableTransactionManagement* annotation to enable declarative transaction support on configuration classes. After using *@EnableTransactionManagement*, the Spring container automatically scans for methods and classes annotated *@Transactional*. *@EnableTransactionManagement* is used in the following way.

@Configuration  
@EnableTransactionManagement  
public class ApplicationConfig {  
 ...  
}

uses the *@Transactional* annotation on the method to indicate that the method needs transaction support

@Transactional  
public void dbOperate(Object xxxx) {  
 //save or update or delete  
}

*@Transactional* can be annotated not only on methods but also on classes. When annotated on a class, all public methods of that class are transaction-enabled. If both class-level and method-level *@Transactional* annotations are used, the annotation used at the class level overrides the method-level annotation.

The Spring team **recommends using *@Transactional* on a specific class (or method of a class) and not on any interface that the class is intended to implement.**

If it has to be used on an interface, **it will only work if you set up an interface-based proxy**. This is because annotations are not inheritable, which means that once you set up an interface-based proxy while you are using a class-based proxy, the transaction settings will not be recognized by the class-based proxy and the object will not be wrapped by the transaction proxy.

**1.3 SpringBoot’s transaction support**

Spring Boot’s transaction configuration class is *org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration*.

@AutoConfiguration(after = { JtaAutoConfiguration.class, HibernateJpaAutoConfiguration.class,  
 DataSourceTransactionManagerAutoConfiguration.class, Neo4jDataAutoConfiguration.class })  
@ConditionalOnClass(PlatformTransactionManager.class)  
@EnableConfigurationProperties(TransactionProperties.class)  
public class TransactionAutoConfiguration {  
 ...  
}

It only works when the class PlatformTransactionManager exists on the classpath. And only after the following auto-configuration classes are applied:

* JtaAutoConfiguration
* HibernateJpaAutoConfiguration
* DataSourceTransactionManagerAutoConfiguration
* Neo4jDataAutoConfiguration

So how does SpringBoot choose which data access technology and corresponding transaction manager to use? It actually chooses based on the dependencies you have introduced.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

When you introduce spring-boot-starter-data-jpa, SpringBoot will use JpaTransactionManager as the transaction manager by default.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>

When you introduce spring-boot-starter-jdbc, SpringBoot will use the DataSourceTransactionManager as the transaction manager by default.

**1.4 Spring Data JPA's transaction support**

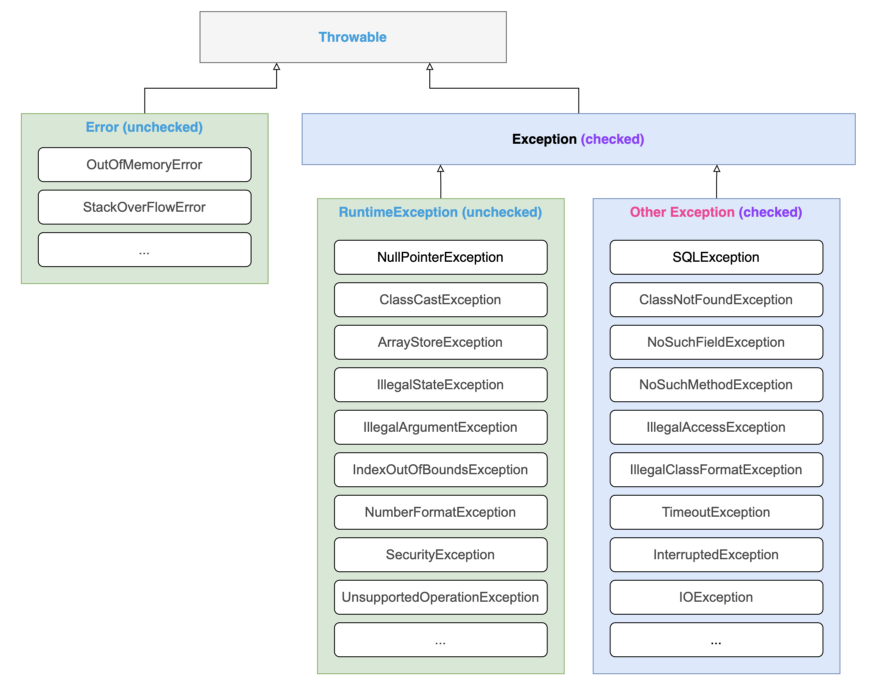
Spring Data JPA has transaction support enabled for all default methods, and the ***readOnly=true*** property is enabled by default for query transactions.

This can be seen from the **SimpleJpaRepository** source code, defined at the class level *@Transactional (readOnly=true)*, but save() and delete() related operations override the *@Transactional* property, at this time the *readOnly* property is false, the rest of the query operation *readOnly* is still true.

@Repository  
@Transactional(readOnly = true)  
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {  
   
 ...  
  
 @Transactional  
 @Override  
 public <S extends T> S save(S entity) {  
 ...  
 }  
  
 ...  
  
 @Override  
 @Transactional  
 @SuppressWarnings("unchecked")  
 public void delete(T entity) {  
 ...  
 }  
}

**2. How to use this annotation well, we need to totally understand some basic rules.**

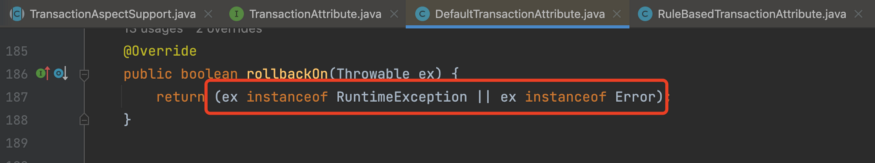
Note that by default, **rollback happens for runtime, unchecked exceptions only** (yellow green part in the diagram).



Exceptions

why this design? Spring believes that checked exception belongs to the business scope, programmers need to give a solution rather than directly to the framework or JVM to deal with.

**The checked exception does not trigger a rollback** of the transaction.

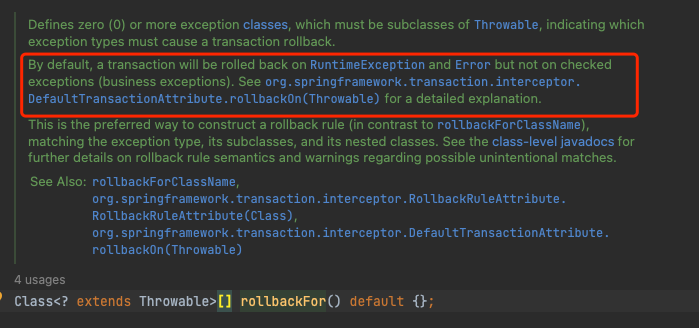


source code

Of course, We can configure this behavior with the *rollbackFor* and *noRollbackFor* annotation parameters.

The *@Transactional* annotation makes use of the attributes *rollbackFor* or *rollbackForClassName* to rollback the transactions, and the attributes *noRollbackFor* or *noRollbackForClassName* to avoid rollback on listed exceptions.

**rollbackFor**



source code

When checked Exception throws will do the rollback.

//without 'rollbackFor'  
 @Transactional  
 public void dbOperate() throws ClassNotFoundException {  
 svcA.save(); // svcA.save() will execute normally.  
 Class.forName("The Class do not Exist"); //throws  
 testBMapper.descNumB();  
 }  
  
 //with 'rollbackFor'  
 @Transactional(rollbackFor = {ClassNotFoundException.class})  
 public void dbOperate() throws ClassNotFoundException {  
 svcA.save(); // svcA.save() will rollback  
 Class.forName("The Class do not Exist"); //throws  
 svcB.update();  
 }

When non-RuntimeExceptions have been try-catched, *rollbackFor* does not work, and the transaction proceeds normally

@Transactional(rollbackFor = {ClassNotFoundException.class})  
public void changeNum() {  
 svcA.save(); // svcA.save() will execute normally.  
 try {  
 Class.forName("The Class do not Exist"); //throws  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 svcB.update(); // svcB.update() will execute normally.  
}

**noRollbackFor**

when throwing RuntimeException will not do the rollback

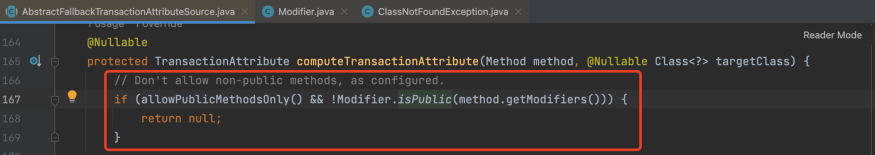
@Transactional(noRollbackFor = {ArithmeticException.class})  
public void doMath(){  
 svcA.save();  
 int a = 1/0; //this will throw AruthmeticException. but svcA.save() will no rollback.  
 svcB.update();  
}

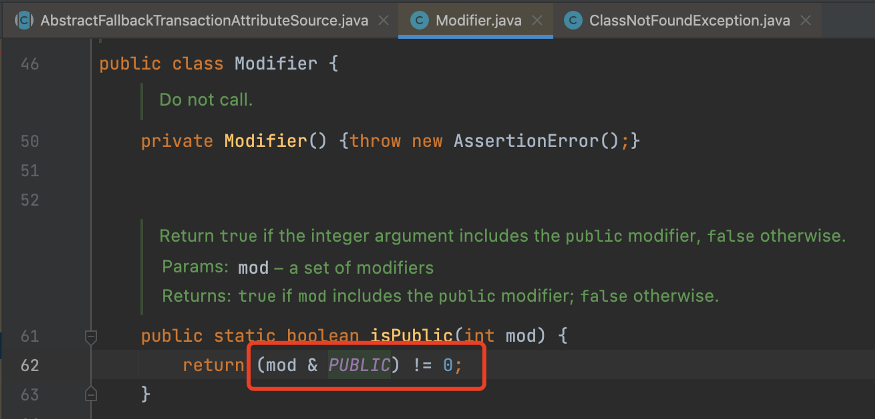
When RuntimeException has been try-catched, *noRollbackFor* does not work, the transaction proceeds normally

@Transactional(noRollbackFor = {ArithmeticException.class})  
public void changeNum(){  
 svcA.save(); // svcA.save() will execute normally.  
 try {  
 int a = 1/0;  
 }  
 catch (Exception e){}  
 svcB.update(); // svcB.update() will execute normally.  
}

**3. What will cause a transaction to fail**

**3.1 The access modifier must be public**. so final/private/static/private/protected are all not working.

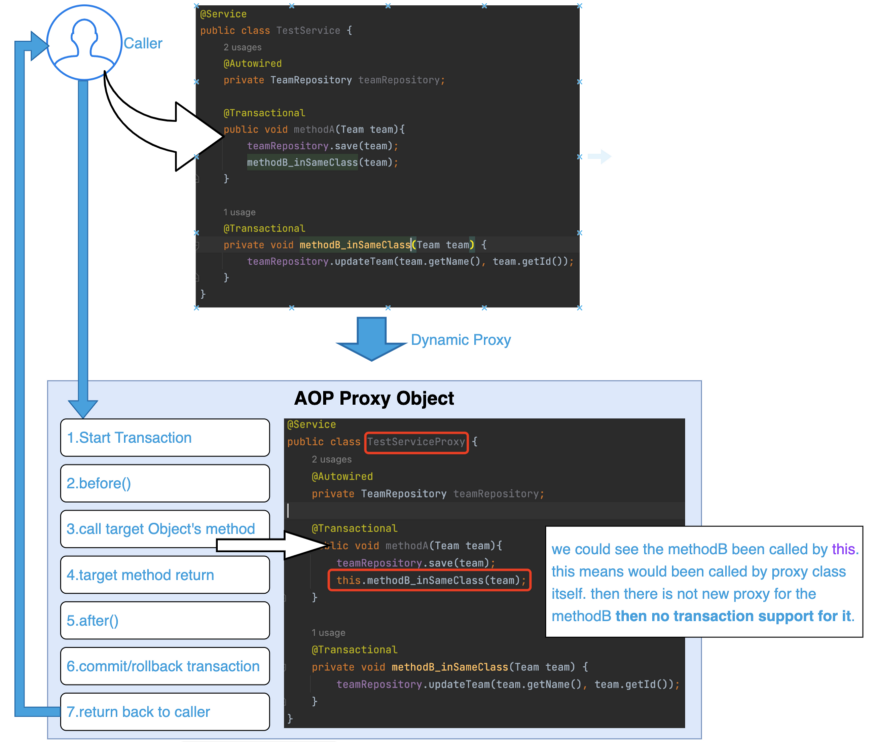




**3.2 The tagged method cannot be in the same class as the @Transactional method.**

I explained this in the last article (point 3.4) <https://medium.com/gitconnected/8-things-you-need-to-know-when-you-want-to-use-spring-async-really-well-e5af4af259c5>

But this time try to use a diagram to explain more clearly.



How AOP Proxy works

**3.3 Bean is not managed by spring**

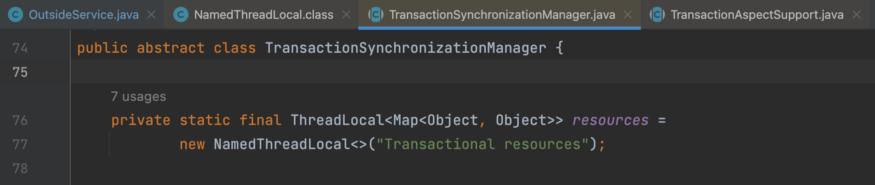
Ex: @Services @Component @Controller @Reposity is not placed on Bean class

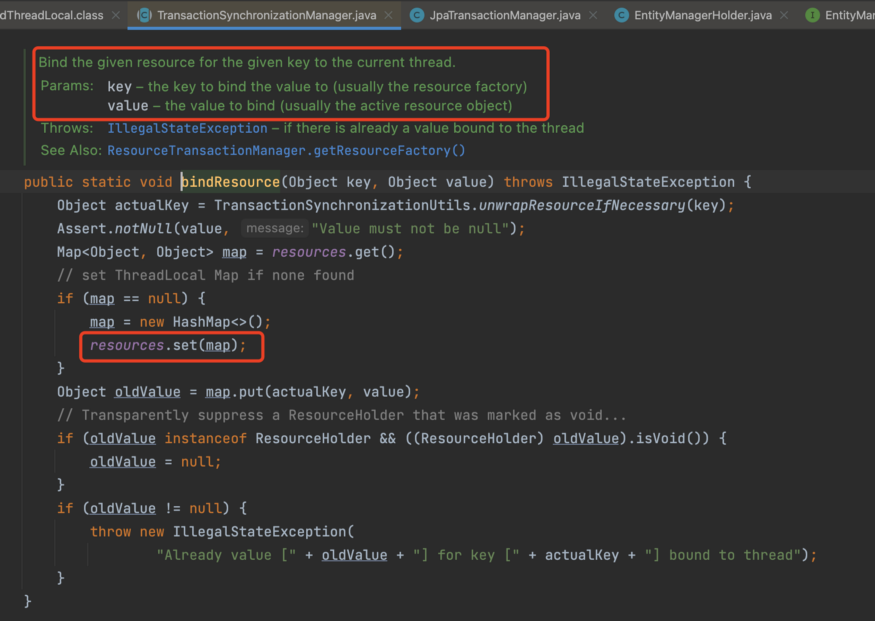
**3.4 Multi-Threaded calls**

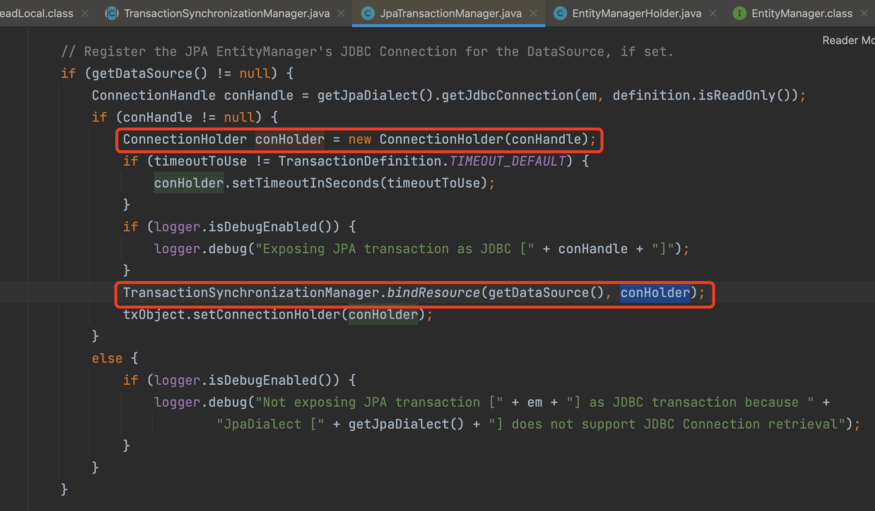
@Service   
public class OutsideService {  
 ...  
  
 @Transactional  
 public void multipleThreadTest(Team team){  
 teamRepository.save(team);  
 new Thread(() -> insideService.doSth()).start(); //this would been called by new thread.  
 }  
}   
  
@Service   
public class InsideService {  
  
 @Transactional  
 public void doSth(){  
 log.info("do sth");  
 }  
}

We can see that *insideService.doSth()* will be called in another new thread, which will coz two methods in different threads, thus getting different database connections, because the transaction is opened and committed/rollbacked in the same database connection, then 2 different database connections will naturally be handled in two different transactions. This leads to transaction failure.

From the source code, we can also see that spring’s transactions are implemented through the database connection, **the current thread will save a map, the key is the data source, value is the database connection**.







How JPATransactionManager call this bindResource(dataSource, connection)

Let’s see some code examples:

3.4.1 Parent Thread Throw Exception. rollback happens. parentThread team will not save. But as the child thread is out of this transaction, the child thread operation will not rollback. then the childThread team will save success.

@Service  
 @Slf4j  
 public class OutsideService {  
 @Transactional  
 public void parentThreadThrowException(Team team){  
 teamRepository.save(team);  
 new Thread(() -> insideService.doSth(new Team("childThread"))).start();  
 throw new RuntimeException("Parent Thread throw Exception");  
 }  
 }  
   
 @Service  
 @Slf4j  
 public class InsideService {  
 @Transactional  
 public void doSth(Team team){  
 teamRepository.save(team);  
 log.info("do sth");  
 }  
 }  
   
 @Test  
 void parentThread\_throw\_RuntimeException\_rollback\_but\_childThread\_save\_success() {  
 Team team = new Team("parentThread");  
 assertThrows(RuntimeException.class, () -> outsideService.parentThreadThrowException(team));  
 Optional<Team> parentThreadTeam = teamRepository.findByName("parentThread");  
 assertFalse(parentThreadTeam.isPresent());  
 Optional<Team> childThreadTeam = teamRepository.findByName("childThread");  
 assertTrue(childThreadTeam.isPresent());  
 }

3.4.2 Child Thread Throw Exception. rollback happens. childThread team will not save. But as the child thread is out of the Parent transaction, so parent thread operation will not rollback. then the parentThread team will save success.

@Service  
@Slf4j  
public class OutsideService {  
 @Transactional  
 public void childThreadThrowException(Team team){  
 teamRepository.save(team);  
 new Thread(() -> insideService.save\_ThrowException(new Team("childThread"))).start();  
 }  
}  
  
@Service  
@Slf4j  
public class InsideService {  
 @Transactional  
 public void save\_ThrowException(Team team){  
 teamRepository.save(team);  
 throw new RuntimeException("Child Thread throw Exception");  
 }  
}  
  
  
@Test  
void inside\_method\_throw\_RuntimeException\_with\_REQUIRED\_NEW\_transaction() {  
 Team team = new Team("parentThread");  
 assertDoesNotThrow(() -> outsideService.childThreadThrowException(team));  
 Optional<Team> newTeam = teamRepository.findByName("parentThread");  
 assertTrue(newTeam.isPresent());  
 Optional<Team> childThreadTeam = teamRepository.findByName("childThread");  
 assertFalse(childThreadTeam.isPresent());  
}

**3.5 Database engine does not support transaction**

Ex: MySQL-MyISAM does not support the transaction.

**3.6 Using the wrong propagation**

Usecase: we just want to catch some RuntimeException happened in the inside transaction and not trigger the outside transaction rollback.

Wrong Example:

this will trigger outside transaction rollback.

The default @Transactional propagation is ***REQUIRED***. If the *RuntimeException* throws out of the transactional proxy, Spring marks the current transaction as rollback only. below *insideService.method()* will throw *NullPointerException* [*RuntimeException]*. Spring will mark the current transaction as rollback only and throw ‘*UnexpectedRollbackException*’

@Service  
@Slf4j  
public class OutsideService {  
 @Transactional  
 public Team dbOperateAndCallTransactionalMethod(Team team){  
 teamRepository.save(team);  
 try{  
 insideService.REQUIRED\_TransactionalMethodAndThrowNullPointerException();  
 } catch (NullPointerException e){  
 log.error("Exception Been catch without throws in current method transaction flow: {}", e.getMessage());  
 }  
 log.info("New Saved Team Id: {}", team.getId());  
 return team;  
 }  
}  
  
   
@Service  
public class InsideService {  
 @Transactional  
 public void REQUIRED\_TransactionalMethodAndThrowNullPointerException(){  
 throw new NullPointerException("sth is null");  
 }  
}  
  
public class PropagationUT {  
 @Test  
 void inside\_method\_throw\_RuntimeException\_with\_default\_REQUIRED\_transaction() {  
 Team team = new Team("NGU");  
 assertThrows(UnexpectedRollbackException.class, () -> outsideService.dbOperateAndCallTransactionalMethod(team),  
 "Transaction silently rolled back because it has been marked as rollback-only");  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertFalse(newTeam.isPresent());  
 }  
}

To accomplish the UseCase. we could do it by use *@Transactional(propagation = Propagation.REQUIRES\_NEW)* on inside transaction

@Service  
@Slf4j  
public class OutsideService {  
 ...  
  
 @Transactional  
 public Team dbOperateAndCallTransactionalMethod\_REQUIRES\_NEW(Team team){  
 teamRepository.save(team);  
 try{  
 insideService.REQUIRES\_NEW\_TransactionalMethodAndThrowRuntimeException();  
 } catch (NullPointerException e){  
 log.error("Exception Been catch without throws in current method transaction flow: {}", e.getMessage());  
 }  
 log.info("New Saved Team Id: {}", team.getId());  
 return team;  
 }  
}  
  
@Service  
@Slf4j  
public class InsideService {  
 ...  
  
 @Transactional(propagation = Propagation.REQUIRES\_NEW)  
 public void REQUIRES\_NEW\_TransactionalMethodAndThrowRuntimeException(){  
 throw new NullPointerException("sth is null");  
 }  
}  
  
public class PropagationUT {  
 ...  
  
 @Test  
 void inside\_method\_throw\_RuntimeException\_with\_REQUIRED\_NEW\_transaction() {  
 Team team = new Team("NGU");  
 assertDoesNotThrow(() -> outsideService.dbOperateAndCallTransactionalMethod\_REQUIRES\_NEW(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
 }  
}

**3.7.1 Exception has been try-cathed by method self.**

@Transactional  
 public void tryCatch(Team team){  
 try {  
 teamRepository.save(team);  
 int a = 1/0;  
 } catch (Exception e) {  
 log.error(e.getMessage());  
 }  
 }  
   
 @Test  
 void tryCatch\_no\_rollback() {  
 Team team = new Team("NGU");  
 assertDoesNotThrow(() -> outsideService.tryCatch(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
 }

**3.7.2 Exception been try-cathed and throw same one in catch**

@Transactional  
 public void tryCatchAndThrow(Team team) throws Exception{  
 try {  
 teamRepository.save(team);  
 int a = 1/0;  
 } catch (Exception e) {  
 log.error(e.getMessage());  
 throw new Exception();  
 }  
 }  
   
 @Test  
 void tryCatchAndThrow\_no\_rollback() {  
 Team team = new Team("NGU");  
 assertThrows(Exception.class, () -> outsideService.tryCatchAndThrow(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
 }

**3.8 Manually throws checked exceptions**

@Transactional  
 public void tryCatchAndThrowOtherCheckedException(Team team) throws TimeoutException {  
 try {  
 teamRepository.save(team);  
 int a = 1/0;  
 } catch (ArithmeticException e) {  
 log.error(e.getMessage());  
 throw new TimeoutException();  
 }  
 }  
   
 @Test  
 void tryCatchAndThrowOtherCheckedException\_no\_rollback() {  
 Team team = new Team("NGU");  
 assertThrows(Exception.class, () -> outsideService.tryCatchAndThrowOtherCheckedException(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
 }

**3.9 Manually throws a customized checked exception.**

public class MyCheckedException extends Exception {  
  
 private static final long serialVersionUID = 1L;  
 private String message;  
  
 @Override  
 public String getMessage() {  
 return message;  
 }  
  
 public void setMessage(String message) {  
 this.message = message;  
 }  
  
 public MyCheckedException(String message) {  
 this.message = message;  
 }  
}  
  
@Transactional  
public void tryCatchAndThrowCustomizeCheckedException(Team team) throws MyCheckedException {  
 try {  
 teamRepository.save(team);  
 throw new MyCheckedException("my custom exception");  
 } catch (MyCheckedException e) {  
 log.error(e.getMessage());  
 throw new MyCheckedException("MyException");  
 }  
}  
  
  
@Test  
void tryCatchAndThrowCustomizeCheckedException\_no\_rollback() {  
 Team team = new Team("NGU");  
 assertThrows(Exception.class, () -> outsideService.tryCatchAndThrowCustomizeCheckedException(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
}

But what if we want it to rollback when throwing customize Exception. there are two ways.

1. customize exception extends from RuntimeException. (code omitted)

2.setRollbackOnly.

@Transactional  
 public void tryCatchAndThrowCustomizeCheckedExceptionSetRollbackOnly(Team team) throws MyCheckedException {  
 try {  
 teamRepository.save(team);  
 throw new MyCheckedException("my custom exception");  
 } catch (MyCheckedException e) {  
 log.error(e.getMessage());  
 TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();  
 throw new MyCheckedException("MyException");  
 }  
 }  
   
 @Test  
 void tryCatchAndThrowCustomizeCheckedException\_setRollbackOnly\_will\_rollback() {  
 Team team = new Team("NGU");  
 assertThrows(Exception.class, () -> outsideService.tryCatchAndThrowCustomizeCheckedExceptionSetRollbackOnly(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertFalse(newTeam.isPresent());  
 }

**3.10 catch custom exception but do not throw. insert won’t be rollback.**

@Transactional  
 public void tryCatchCustomizeException(Team team){  
 try {  
 teamRepository.save(team);  
 throw new MyCheckedException("my custom exception");  
 } catch (MyCheckedException e) {  
 log.error(e.getMessage());  
 }  
 }  
  
 @Test  
 void tryCatch\_CustomizeException\_without\_throw\_no\_rollback() {  
 Team team = new Team("NGU");  
 assertDoesNotThrow(() -> outsideService.tryCatchCustomizeException(team));  
 Optional<Team> newTeam = teamRepository.findByName("NGU");  
 assertTrue(newTeam.isPresent());  
 }

**4. Best practices**

4.1 Avoid the long/big transaction as possible as we can.

As Long/Big Transaction will coz many problems.

* When rollback happens will cost a long time.
* when the concurrent operation may take over all the active DB connections
* If the endpoint has a long transaction may coz a response timeout.
* deadlock
* lock wait

4.2 Put the select/query DB operation method out of the transaction.

4.3 Avoid one transaction handling too much data.

4.4 Avoid putting the remote call into a transaction. (remote call may take a while)

4.5 some needed functions maybe we don’t need to use transaction. or we may use @Async to handle it.

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

That’s it. I’m trying not to make the article too long, and I’ll consider writing Part 2 based on your feedback.

*Part2. maybe focus on****Dive into the propagation behavior of transactions and the isolation level of transactions.***If you would like to see it pls leave it in the comments.

Thanks for reading! If you like it or feel it helped pls click Applaud :)

Happy coding. See you next time :)