事务及其运用

1.事务简介

事务就个人理解而言往往是为了实现一个为一个数据存储系统实现类似备忘录的功能，事务开启相当于标记当前状态，事务回滚是回到标记状态，提交事务则是保存操作后的状态。为了实现这样的功能，事务至少拥有如下方法，begin、commit、rollback。因此一个支持事务的最简单接口应该如下：

```java
/**
 * 事务处理支持
 *
 * @author payno
 * @date 2020/07/19
 */
public interface TransationSupport {
    void begin();
    void commit();
    void rollback();
}
```

最简单的可以实现一个简单的集合去支持事务，begin时拷贝保存副本，rollback时用副本替代真实操作的数据，commit则提交真实操作的数据到副本。但是真实的数据库肯定在事务的操作的颗粒度上更细，猜测是记录产生影响的行数，然后局部更新恢复。

```java
public class TransactionList<E> extends ArrayList<E> implements TransationSupport {

    List<E> memento;

    public TransactionList() {...}

    @Override
    public void begin() {
        memento = this;
    }

    @Override
    public void commit() {
        memento = this;
    }

    @Override
    public void rollback() {
        this.clear();
        this.addAll(memento);
    }
}
```

以上只是简单的demo，真实的事务实现肯定颗粒度更细也更复杂，可以参考Git如何实现回滚以及提交等相关操作，而且这个demo只是单线程的。如果多线程就会遇到事务互相影响的问题，如果渴望事务之间无影响肯定是串行执行，但是那样效率就会很低，因此事务的隔离级别的

概念就成了必要，根据不同的情况优化事务的执行效率，如下：

“读未提（Read Uncommitted）”能预防啥？啥都预防不了。

“读提交（Read Committed）”能预防啥？使用“快照读（Snapshot Read）”，避免“脏读”，但是可能出现“不可重复读”和“幻读”。

“可重复读（Repeated Red）”能预防啥？使用“快照读（Snapshot Read）”，锁住被读取记录，避免出现“脏读”、“不可重复读”，但是可能出现“幻读”。

“串行化（Serializable）”能预防啥？排排坐，吃果果，有效避免“脏读”、“不可重复读”、“幻读”，不过效果谁用谁知道。

2.JDBC事务

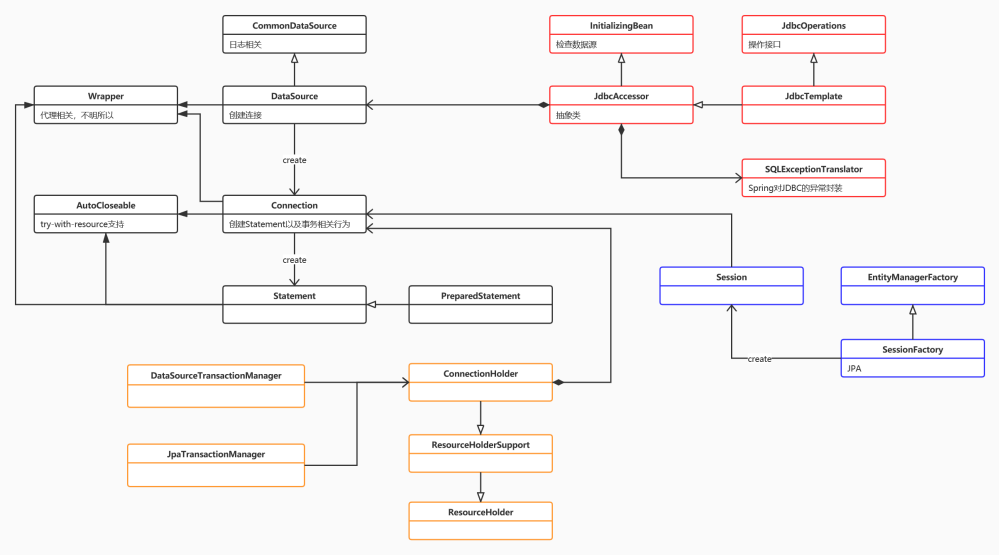
不同的数据库的访问接口是不一样的，为了让Java开发者在操作不同数据库拥有相同的api，Java提供了一套SPI作为访问数据库的接口，每个数据库厂商实现对应接口，这样Java开发者在操作数据库某种层面上对于数据库是无感知的，不需要知道数据库是什么数据库，统一的使用Connection、Statement等接口操作数据库。

其中跟数据库事务相关的接口就是Connection，无论是Mybatis还是JPA又或者是JOOQ等框架，凡是涉及到事务操作最终使用的都是Connection接口。而Connection接口有以下方法commit、rollback、setAutoCommit、commit（SavePoint）...，JDBC是默认开启自动提交的，相关的解释如下：

JDBC connections start out with auto-commit mode enabled, where each SQL statement is implicitly demarcated with a transaction.
Users who wish to execute multiple statements per transaction must turn auto-commit off.
Changing the auto-commit mode triggers a commit of the current transaction (if one is active).
Connection.setTransactionIsolation() may be invoked anytime if auto-commit is enabled.
If auto-commit is disabled, Connection.setTransactionIsolation() may only be invoked before or after a transaction. Invoking it in the middle of a transaction leads to undefined behavior.
Sources:

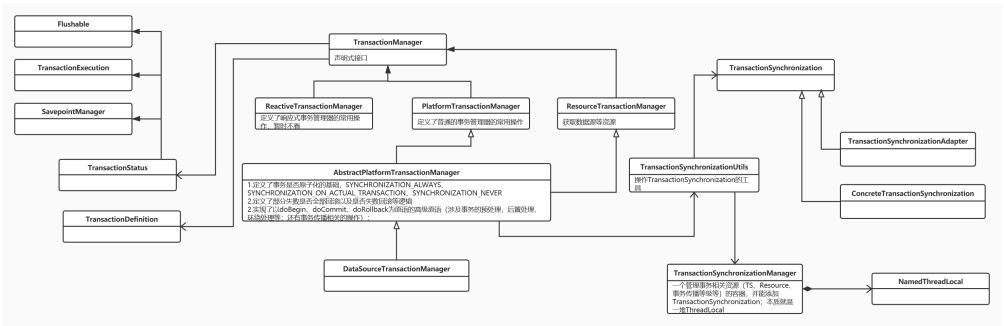
Javadoc
JDBC Tutorial

下图是JpaTransaction最终如何实现的，通过Session委托Connection实现的事务行为



3.事务管理器与事务传播级别

在Spring中，为了实现事务的集成，Spring实现了事务模块，而其中和事务操作相关的就是TransactionManager接口，这个接口是Spring事务模块的核心接口，整体大概的类图如下：



其中TransactionManager本身只是一个声明式接口，真正常用的接口是PlatformTransactionManager，与之对应的另一个接口是ReactiveTransactionManager。前者是在非响应式下使用，后者在响应式下使用，其中JDBC的实现是阻塞的，因此关系型数据库基本都不支持响应式事务管理器，我们需要关注的是站台式事务管理器。这里有三个方法，根据事务定义拿到事务状态，提交或者回滚。

```
public interface PlatformTransactionManager extends TransactionManager {
    TransactionStatus getTransaction(@Nullable TransactionDefinition var1) throws TransactionException;

    void commit(TransactionStatus var1) throws TransactionException;

    void rollback(TransactionStatus var1) throws TransactionException;
}
```

其中事务管理器的简单使用可以查看TransactionTemplate中的使用方式，这样就能对事务管理器的使用有具体的认知，（这里使用了Callback的思想，可以对比Listener与abstarct方法体会各自留下的拓展点）

```
@Nullable
public <T> T execute(TransactionCallback<T> action) throws TransactionException {
    Assert.state( expression: this.transactionManager != null,  message: "No PlatformTransactionManager set");
    if (this.transactionManager instanceof CallbackPreferringPlatformTransactionManager) {
        return ((CallbackPreferringPlatformTransactionManager)this.transactionManager).execute( transactionDefinition: this, action);
    } else {
        TransactionStatus status = this.transactionManager.getTransaction( transactionDefinition: this);

        Object result;
        try {
            result = action.doInTransaction(status);
        } catch (Error | RuntimeException var5) {
            this.rollbackOnException(status, var5);
            throw var5;
        } catch (Throwable var6) {
            this.rollbackOnException(status, var6);
            throw new UndeclaredThrowableException(var6, "TransactionCallback threw undeclared checked exception");
        }

        this.transactionManager.commit(status);
        return result;
    }
}
```

回调函数，只有doInTransaction方法

开启事务，TransactionTemplate本身是TransactionDefinition的继承类，可以设置传播及隔离级别

回滚事务与异常处理

提交事务

看一下PlatformTransactionManager接口，可以看到后面两个是回滚和提交，但多了一个得到事务状态的接口，为什么多了这样的一个接口呢，为什么要拿到事务的状态信息呢？进入AbstractPlatformTransactionManager发现这个接口就是为了实现事务的传播级别功能，事务的传播级别就是在这里实现（换而言之，事务传播级别跟提供事务功能的服务端无关，是Spring通过事务管理器进行拓展得到的功能，接口与实现分离，模板方法与桥梁模式），根据当前的事务定义以及当前是否存在事务来决定是抛出异常还是加入当前事务或者创建新事务等行为。常用的事务传播级别有默认级别，假如当前不存在事务创建事务，如果存在则加入当前事务（Required）；还有当前不存在事务则不加入事务，存在事务则加入（Support），类似的还有（Required_New）以及在事务中执行跑异常等级别

```
public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
    TransactionDefinition def = definition != null ? definition : TransactionDefinition.withDefaults();
    Object transaction = this.doGetTransaction();
    boolean debugEnabled = this.logger.isDebugEnabled();
    if (this.isExistingTransaction(transaction)) {
        return this.handleExistingTransaction(def, transaction, debugEnabled);
    } else if (def.getTimeout() < -1) {
        throw new InvalidTimeoutException("Invalid transaction timeout", def.getTimeout());
    } else if (def.getPropagationBehavior() == 2) {
        throw new IllegalTransactionStateException("No existing transaction found for transaction marked with propagation 'mandatory'");
    } else if (def.getPropagationBehavior() != 0 && def.getPropagationBehavior() != 3 && def.getPropagationBehavior() != 6) {
        if (def.getIsolationLevel() != -1 && this.logger.isWarnEnabled()) {
            this.logger.warn( o: "Custom isolation level specified but no actual transaction initiated; isolation level will effectively be ignored: " + def);
        }

        boolean newSynchronization = this.getTransactionSynchronization() == 0;
        return this.prepareTransactionStatus(def, (Object)null,  newTransaction: true, newSynchronization, debugEnabled, (Object)null);
    } else {
        AbstractPlatformTransactionManager.SuspendedResourcesHolder suspendedResources = this.suspend((Object)null);
        if (debugEnabled) {
            this.logger.debug( o: "Creating new transaction with name [" + def.getName() + "]: " + def);
        }

        try {
            boolean newSynchronization = this.getTransactionSynchronization() != 2;
            DefaultTransactionStatus status = this.newTransactionStatus(def, transaction,  newTransaction: true, newSynchronization, debugEnabled, suspendedResources);
            this.doBegin(transaction, def);
            this.prepareSynchronization(status, def);
            return status;
        } catch (Error | RuntimeException var8) {
            this.resume((Object)null, suspendedResources);
            throw var8;
        }
    }
}
```

当前存在事务

事务传播级别要求当前必须存在事务才能运行，否则抛出异常

NEVER,SUPPORTS等级别，支持无事务不在事务中运行

创建新事务

事务管理器抽象类，其中有几个抽象方法未实现，就是doCommit，doRollback，doGetTransaction。首先是doCommit等，先看commit和processCommit方法实现了一些异常状态判断以及事务原子管理器的相关功能，这几个未实现是因为不同的事务管理器具体的操作是不一样的。

DataSourceTransactionManager需要的是JDBC的Connection对象进行事务操作

```
protected Object doGetTransaction() {
    DataSourceTransactionManager.DataSourceTransactionObject txObject = new DataSourceTransactionManager.DataSourceTransactior
    txObject.setSavepointAllowed(this.isNestedTransactionAllowed());
    ConnectionHolder conHolder = (ConnectionHolder)TransactionSynchronizationManager.getResource(this.obtainDataSource());
    txObject.setConnectionHolder(conHolder,  newConnectionHolder: false);
    return txObject;
}
```

下面是RabbitTransactionManager中拿到的事务对象，需要的是Connetion、Channel等对象进行事务操作

```
protected Object doGetTransaction() {
    RabbitTransactionManager.RabbitTransactionObject txObject = new RabbitTransactionManager.RabbitTransactionObject();
    txObject.setResourceHolder((RabbitResourceHolder)TransactionSynchronizationManager.getResource(this.getConnectionFactory())
    return txObject;
}
```

4.事务管理器与事务原子管理器

事务原子管理器则是另一个在事务处理中及其重要的类，可以看到里面有一堆ThreadLocal，所以事务相关资源都是跟线程绑定的，通过该管理器可以拿到事务的名称，事务的隔离级别，事务的原子环绕处理器（重点），事务绑定的资源

```
public abstract class TransactionSynchronizationManager {
    private static final Log logger = LogFactory.getLog(TransactionSynchronizationManager.class);
    private static final ThreadLocal<Map<Object, Object>> resources = new NamedThreadLocal("Transactional resources");
    private static final ThreadLocal<Set<TransactionSynchronization>> synchronizations = new NamedThreadLocal("Transaction
    private static final ThreadLocal<String> currentTransactionName = new NamedThreadLocal("Current transaction name");
    private static final ThreadLocal<Boolean> currentTransactionReadOnly = new NamedThreadLocal("Current transaction read-
    private static final ThreadLocal<Integer> currentTransactionIsolationLevel = new NamedThreadLocal("Current transaction
    private static final ThreadLocal<Boolean> actualTransactionActive = new NamedThreadLocal("Actual transaction active");

    public TransactionSynchronizationManager() {
    }
```

其中事务的原子环绕处理器的主要作用有资源绑定（bindResource、unbindResource，释放是为了防止内存泄漏）与事务环绕处理（getSynchronizations、registerSynchronization），其中的逻辑都是在抽象站台事务管理器中实现的。

其中TransactionSynchronization实现原理如下:在抽象站台事务管理器中提交执行commit操作，会从commit到processCommit，其中processCommit里会触发相关的事务环绕对象，这些对象保存在TransactionSynchronizationManager里的ThreadLocal里.

commit触发processCommit

```java
public final void commit(TransactionStatus status) throws TransactionException {
    if (status.isCompleted()) {
        throw new IllegalTransactionStateException("Transaction is already completed - do not call commit or rollback more than once per transaction");
    } else {
        DefaultTransactionStatus defStatus = (DefaultTransactionStatus)status;
        if (defStatus.isLocalRollbackOnly()) {
            if (defStatus.isDebug()) {
                this.logger.debug( o: "Transactional code has requested rollback");
            }

            this.processRollback(defStatus,  unexpected: false);
        } else if (!this.shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {
            if (defStatus.isDebug()) {
                this.logger.debug( o: "Global transaction is marked as rollback-only but transactional code requested commit");
            }

            this.processRollback(defStatus,  unexpected: true);
        } else {
            this.processCommit(defStatus);
        }
    }
}
```

根据当前事务状态信息判断是提交还是回滚

processCommit触发相关环绕操作

```java
private void processCommit(DefaultTransactionStatus status) throws TransactionException {
    try {
        boolean beforeCompletionInvoked = false;

        try {
            boolean unexpectedRollback = false;
            this.prepareForCommit(status);
            this.triggerBeforeCommit(status);
            this.triggerBeforeCompletion(status);
            beforeCompletionInvoked = true;
            if (status.hasSavepoint()) {
                if (status.isDebug()) {
                    this.logger.debug( o: "Releasing transaction savepoint");
                }

                unexpectedRollback = status.isGlobalRollbackOnly();
                status.releaseHeldSavepoint();
            } else if (status.isNewTransaction()) {
                if (status.isDebug()) {
                    this.logger.debug( o: "Initiating transaction commit");
                }

                unexpectedRollback = status.isGlobalRollbackOnly();
                this.doCommit(status);
            } else if (this.isFailEarlyOnGlobalRollbackOnly()) {
                unexpectedRollback = status.isGlobalRollbackOnly();
            }

            if (unexpectedRollback) {
                throw new UnexpectedRollbackException("Transaction silently rolled back because it has been marked as rollback-only");
            }
        } catch (UnexpectedRollbackException var17) {
            this.triggerAfterCompletion(status,  completionStatus: 1);
            throw var17;
        } catch (TransactionException var18) {
            if (this.isRollbackOnCommitFailure()) {
                this.doRollbackOnCommitException(status, var18);
            } else {
                this.triggerAfterCompletion(status,  completionStatus: 2);
            }
```

触发BeforeCommit

提交操作，抽象方法，留给具体的事务管理器来实现

触发AfterCompletion

最终使用TransactionSynchronizationUtils触发对应的回调方法

```java
private void triggerAfterCompletion(DefaultTransactionStatus status, int completionStatus) {
    if (status.isNewSynchronization()) {
        List<TransactionSynchronization> synchronizations = TransactionSynchronizationManager.getSynchronizations();
        TransactionSynchronizationManager.clearSynchronization();
        if (status.hasTransaction() && !status.isNewTransaction()) {
            if (!synchronizations.isEmpty()) {
                this.registerAfterCompletionWithExistingTransaction(status.getTransaction(), synchronizations);
            }
        } else {
            if (status.isDebug()) {
                this.logger.trace( o: "Triggering afterCompletion synchronization");
            }

            this.invokeAfterCompletion(synchronizations, completionStatus);
        }
    }

}
```

```java
protected final void invokeAfterCompletion(List<TransactionSynchronization> synchronizations, int completionStatus) {
    TransactionSynchronizationUtils.invokeAfterCompletion(synchronizations, completionStatus);
}
```

另一个重要作用则在于资源绑定，一般资源绑定需要到是在getTransaction中，对于新的事务会有doBegin操作

```java
public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
    TransactionDefinition def = definition != null ? definition : TransactionDefinition.withDefaults();
    Object transaction = this.doGetTransaction();
    boolean debugEnabled = this.logger.isDebugEnabled();
    if (this.isExistingTransaction(transaction)) {
        return this.handleExistingTransaction(def, transaction, debugEnabled);
    } else if (def.getTimeout() < -1) {
        throw new InvalidTimeoutException("Invalid transaction timeout", def.getTimeout());
    } else if (def.getPropagationBehavior() == 2) {
        throw new IllegalTransactionStateException("No existing transaction found for transaction marked with propagation 'mandatory'");
    } else if (def.getPropagationBehavior() != 0 && def.getPropagationBehavior() != 3 && def.getPropagationBehavior() != 6) {
        if (def.getIsolationLevel() != -1 && this.logger.isWarnEnabled()) {
            this.logger.warn( o: "Custom isolation level specified but no actual transaction initiated; isolation level will effectively be ignored: " + def);
        }

        boolean newSynchronization = this.getTransactionSynchronization() == 0;
        return this.prepareTransactionStatus(def, (Object)null,  newTransaction: true, newSynchronization, debugEnabled, (Object)null);
    } else {
        AbstractPlatformTransactionManager.SuspendedResourcesHolder suspendedResources = this.suspend((Object)null);
        if (debugEnabled) {
            this.logger.debug( o: "Creating new transaction with name [" + def.getName() + "]: " + def);
        }

        try {
            boolean newSynchronization = this.getTransactionSynchronization() != 2;
            DefaultTransactionStatus status = this.newTransactionStatus(def, transaction,  newTransaction: true, newSynchronization, debugEnabled, suspendedResourc
            this.doBegin(transaction, def);
            this.prepareSynchronization(status, def);
            return status;
        } catch (Error | RuntimeException var8) {
            this.resume((Object)null, suspendedResources);
            throw var8;
        }
    }
```

如果是新的事务，则新建Status对象并doBegin

在doBegin中绑定事务可能用到的Resource，如果是第一次创建资源则会绑定在事务管理器里，方便后面使用（线程绑定），这里看的是DataSourceTransactionManager，绑定的是ConnectionHolder，里面有着JDBC的Connection对象

```java
protected void doBegin(Object transaction, TransactionDefinition definition) {
    DataSourceTransactionManager.DataSourceTransactionObject txObject = (DataSourceTransactionManager.DataSourceTransactionObject)transaction;
    Connection con = null;

    try {
        if (!txObject.hasConnectionHolder() || txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
            Connection newCon = this.obtainDataSource().getConnection();
            if (this.logger.isDebugEnabled()) {
                this.logger.debug( o: "Acquired Connection [" + newCon + "] for JDBC transaction");
            }

            txObject.setConnectionHolder(new ConnectionHolder(newCon),  newConnectionHolder: true);
        }

        txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
        con = txObject.getConnectionHolder().getConnection();
        Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);
        txObject.setPreviousIsolationLevel(previousIsolationLevel);
        txObject.setReadOnly(definition.isReadOnly());
        if (con.getAutoCommit()) {
            txObject.setMustRestoreAutoCommit(true);
            if (this.logger.isDebugEnabled()) {
                this.logger.debug( o: "Switching JDBC Connection [" + con + "] to manual commit");
            }

            con.setAutoCommit(false);
        }

        this.prepareTransactionalConnection(con, definition);
        txObject.getConnectionHolder().setTransactionActive(true);
        int timeout = this.determineTimeout(definition);
        if (timeout != -1) {
            txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
        }

        if (txObject.isNewConnectionHolder()) {
            TransactionSynchronizationManager.bindResource(this.obtainDataSource(), txObject.getConnectionHolder());
        }
```

绑定资源ConnectionHolder

在需要用的时候，可以很容易的获取

```java
protected Object doGetTransaction() {
    DataSourceTransactionManager.DataSourceTransactionObject txObject = new DataSourceTransactionManager.DataSourceTransactionObject();
    txObject.setSavepointAllowed(this.isNestedTransactionAllowed());
    ConnectionHolder conHolder = (ConnectionHolder)TransactionSynchronizationManager.getResource(this.obtainDataSource());
    txObject.setConnectionHolder(conHolder,  newConnectionHolder: false);
    return txObject;
}
```

最终使用时，拿Status里的Transaction对象，最终拿到资源，操作资源对象来进行提交、回滚等操作

```
protected void doCommit(DefaultTransactionStatus status) {
    DataSourceTransactionManager.DataSourceTransactionObject txObject = (DataSourceTransactionManager.DataSourceTransactionObject)status.getTransaction();
    Connection con = txObject.getConnectionHolder().getConnection();
    if (status.isDebug()) {
        this.logger.debug( o: "Committing JDBC transaction on Connection [" + con + "]");
    }

    try {
        con.commit();
    } catch (SQLException var5) {
        throw new TransactionSystemException("Could not commit JDBC transaction", var5);
    }
}

protected void doRollback(DefaultTransactionStatus status) {
    DataSourceTransactionManager.DataSourceTransactionObject txObject = (DataSourceTransactionManager.DataSourceTransactionObject)status.getTransaction();
    Connection con = txObject.getConnectionHolder().getConnection();
    if (status.isDebug()) {
        this.logger.debug( o: "Rolling back JDBC transaction on Connection [" + con + "]");
    }

    try {
        con.rollback();
    } catch (SQLException var5) {
        throw new TransactionSystemException("Could not roll back JDBC transaction", var5);
    }
}
```

这里可以梳理下重要对象的关系，首先通过TransactionDefinition得到TransactionStatus对象（通过事务管理器的方法），一般事务对象也在TransactionStatus之中，我们通过事务管理器和TransactionStatus操作事务行为。

在事务管理器的getTransaction方法中，会根据传播级别决定是抛异常还是加入当前事务或者新建事务等（传播行为）

如果是新的事务，会触发doBegin方法，在doBegin方法中会把操作事务所需要的对象绑定在原子管理器，最终设置到TransactionStatus的Transaction对象里

在其他的do方法里，则通过status拿到transaction对象最终拿到绑定的资源（DataSourceTransaction是TransactionStatus->DataSourceTransactionObject->ConnectionHolder->Connection，Jpa里是TransactionStatus->JpaTransactionObject->EntityManagerHolder->Session->Connection,Rabbit里是TransactionStatus->RabbitTransactionObject->RabbitResourceHolder->Connection、Channel，Kafka里是TransactionStatus->KafkaTransactionObject<K, V>->KafkaResourceHolder->Producer<K, V>,Redisson里是TransactionStatus->RedissonTransactionObject->RTransaction）来进行事务操作

而在抽象事务管理器里，执行commit，rollback等方法时又会触发TransactionSynchronizationManager注册的TransactionSynchronization对象，从而进行事务的环绕操作

5.事务支持之Redis

一般情况下，一些第三方框架或者Spring的其他模块如果要支持事务都会实现事务管理器用于集成Spring.Transaction，但是在Spring.data.redis里却找不到AbstractTransactionManager的实现类，但是我们知道Redis是支持事务的，那么spring.data.redis是如何实现事务的呢

关键在于RedisTemplate，其中RedisTemplate可以设置事务是否开启，那么RedisTemplate是如何实现事务管理的呢？

首先，我们知道redis为了实现事务拥有四个命令,MULTI用来组装一个事务；EXEC用来执行一个事务；DISCARD用来取消一个事务；WATCH类似于乐观锁机制里的版本号，通过这些基本命令封装出commit和rollback的逻辑（实际在Spring.data叫exec与discard）;其次操作Redis是基于ConnectionFactory创建的Connection操作的，spring.data.redis实现事务是在doGetConnection中实现了事务相关的逻辑，每次获取connection时根据是否支持事务调用不同的方法，如果支持事务则注册TransactionSynchronization在事务afterCompletion时，根据事务状态决定是exec或者discard。大部分spring集成其他模块实现事务管理都是实现自己的事务管理器，而在Spring.data.redis却找不到对应的实现类，经过一般探索最终在RedisTemplate中找到了原理，就是基于TransactionSynchronization

```
public static RedisConnection doGetConnection(RedisConnectionFactory factory, boolean allowCreate, boolean bind, boolean enable
    Assert.notNull(factory,  message: "No RedisConnectionFactory specified");
    RedisConnectionUtils.RedisConnectionHolder connHolder = (RedisConnectionUtils.RedisConnectionHolder)TransactionSynchronizat
    if (connHolder != null) {
        if (enableTransactionSupport) {
            potentiallyRegisterTransactionSynchronisation(connHolder, factory);
        }

        return connHolder.getConnection();
    } else if (!allowCreate) {
        throw new IllegalArgumentException("No connection found and allowCreate = false");
    } else {
        if (log.isDebugEnabled()) {
            log.debug( o: "Opening RedisConnection");
        }

        RedisConnection conn = factory.getConnection();
        if (bind) {
```

```
private static void potentiallyRegisterTransactionSynchronisation(RedisConnectionUtils.RedisConnectionHolder connHolder, RedisConnectionF
    if (isActualNonReadonlyTransactionActive() && !connHolder.isTransactionSyncronisationActive()) {
        connHolder.setTransactionSyncronisationActive(true);
        RedisConnection conn = connHolder.getConnection();
        conn.multi();
        TransactionSynchronizationManager.registerSynchronization(new RedisConnectionUtils.RedisTransactionSynchronizer(connHolder, conn,
    }

}
```

```
private static class RedisTransactionSynchronizer extends TransactionSynchronizationAdapter {
    private final RedisConnectionUtils.RedisConnectionHolder connHolder;
    private final RedisConnection connection;
    private final RedisConnectionFactory factory;

    public void afterCompletion(int status) {
        try {
            switch(status) {
            case 0:
                this.connection.exec();
                break;
            case 1:
            case 2:
            default:
                this.connection.discard();
            }
        } finally {
            if (RedisConnectionUtils.log.isDebugEnabled()) {
                RedisConnectionUtils.log.debug( o: "Closing bound connection after transaction completed with " + statu
            }

            this.connHolder.setTransactionSyncronisationActive(false);
            this.connection.close();
            TransactionSynchronizationManager.unbindResource(this.factory);
        }
```

这里需要注意spring.data.redis必须运作在事务里才能触发TransactionSynchronization，可是redis模块没有，因此只能在其他事务管理器的声明式事务里触发(比如DataSourceTransactionManager)

6.事务支持之Redisson

Reddisson是一个基于Redis实现大量分布式组件的框架，包括分布式容器（List、Set、Map等）、分布式同步器（并发互斥锁、并发限流器、并发读写锁等）

其实现事务是基于ReddissonTransactionManager

需要注意的是使用Redisson的声明式事务，需要按照以下模式（具体原因是因为使用声明式事务时，事务的创建是在切面里的，你只要通过事务原子管理器才能拿到绑定的事务对象。。。。。。比较麻烦，不说了），

```
/**
 *  最终jpa提交，redisson提交
 */
@Transactional(rollbackFor = RuntimeException.class)
public void test2(){
    RedissonTransactionHolder holder=(RedissonTransactionHolder)TransactionSynchronizationManager
            .getResource(redissonClient);
    RSet<String> set=holder.getTransaction().getSet( s: "tran-test");
    set.add("rollbackRedis");
    repo.save(Model.builder().cusName("rollback").build());
}
```

7.其他

1.一般情况下，如果想要在一个声明式事务里同时使用多个服务端的事务（比如Redis+Mysql+RabbitMq），那么简单的可以考虑使用ChainedTransactionManager（全部提交或者全部回滚）组合多个事务管理器，如果使用了分库分表中间件，也需要注意使用对应的事务管理器，或者自己实现TransactionManager包装多个事务管理器，然后在@Transactional中标明.

2.如果想要了解某个中间件事务操作的方法，可以去找PlatformTransactionManager的实现类

3.拒绝默认传播级别的操作出现在事务环绕处理中的afterCommit和afterCompletion里，否则会出现操作失效且无任何报错的行为（事务信息绑定在线程里，进行后置处理时当前事务是已提交的，之后的操作不会提交，相当于无效操作），解决方法，异步执行或者改成Required_New级别

4.公司的异步事件处理就有一部分逻辑基于TransactionSynchronizationManager实现（在事务提交后生产事件到Rabbit，在事务执行完毕后清除Mysql的相关信息）

5.接口是要暴露的方法，抽象类实现了模板（留下拓展的部分，有时将执行策略留下拓展(RedisOperations与RedisTemplate)，有时确定执行策略但是不实现具体的操作（FrameworkServlet与DispatcherServlet），最复杂的就是AbstractQueuedSyncer了）