

TransactionSynchronizationManager initSynchronization 什么作 , 什么?  
 DefaultTransactionStatus 中 储 事务  
 spring 何创 事务 ?  
 中 事务 spring中 事务 什么 别  
 Connection 事务之 关  
 事务 connection ThreadLocal之 关  
 事务

先 代

```

19 public void test() {
20     TransactionSynchronizationManager.initSynchronization();
21     TransactionSynchronizationManager.isSynchronizationActive();
22     DruidDataSource dataSource; dataSource = "{\n\tcreateTime:"2021-02-08 14:55:33",\n\tActiveCount:0,\n\tPoolingCount:0,\n\t\tCre
23     dataSource = new DruidDataSource();
24
25     // DataSourceTransactionManager dataSourceTransactionManager = new DataSourceTransactionManager(dataSource);
26     // TransactionTemplate transactionTemplate = new TransactionTemplate(dataSourceTransactionManager, transactionDefinition);
27     TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager, transactionDefinition);
28     NamedParameterJdbcTemplate jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
29     Object result = transactionTemplate.execute(new TransactionCallback<Object>() { transactionTemplate: "PROPAGATI
30
31     public Object doInTransaction(TransactionStatus status){
32
33         if (true) {
34             throw new RuntimeException();
35         }
36         return jdbcTemplate.update( sql: "update union_student set name= zhangsan where oid=1", Maps.newHashMap());
37     }
38     });
39     System.out.println(result);
40     TransactionSynchronizationManager.isActualTransactionActive();
41     TransactionStatus transactionStatus2 = getTransactionStatus(transactionManager, jdbcTemplate);
42
43 }

```

, TransactionSynchronizationManager initSynchronization 什么作 , 实 原 什么?

```

/**
 * Activate transaction synchronization for the current thread.
 * Called by a transaction manager on transaction begin.
 * @throws IllegalStateException if synchronization is already active
 */
public static void initSynchronization() throws IllegalStateException {
    if (isSynchronizationActive()) {
        throw new IllegalStateException("Cannot activate transaction synchronization - already active");
    }
    logger.trace("Initializing transaction synchronization");
    synchronizations.set(new LinkedHashSet<>());
}

```

前 事务 事务 事务 以 为 为 前 创 个

TransactionSynchronization manager

if中 了isSynchronizationActive 内 Synchronizations 个ThreadLocal , if中 判 个  
 ThreadLocal 中 值, 值 active, 值

```
/**
 * Return if transaction synchronization is active for the current thread.
 * Can be called before register to avoid unnecessary instance creation.
 * @see #registerSynchronization
 */
```

```
public static boolean isSynchronizationActive() {
    return (synchronizations.get() != null);
}
```

```
private static final ThreadLocal<Set<TransactionSynchronization>> synchronizations =
    new NamedThreadLocal<>("Transaction synchronizations");
```

initSynchronizationActive 中, 先 isSynchronizationActive 前  
synchronizations 个ThreadLocal

### 么在Spring中 如何使 TransactionSynchronizationManager initSynchronization ?

们 Spring中事务 入 AbstractPlatformManager getTransaction , getTransaction 中会 两  
况

- (1) 事务-- 传 决 创 事务
- (2) 前 不 事务, 创 个 事务

```
@Override
public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
    Object transaction = doGetTransaction();

    if (isExistingTransaction(transaction)) {
        // Existing transaction found -> check propagation behavior to find out how to behave.
        return handleExistingTransaction(definition, transaction, debugEnabled);
    }

    else if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRED ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_MANDATORY ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRED_NEW ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRES_NEW) {
        SuspendedResourceHolder suspendedResourceHolder = suspend(transaction);

        if (debugEnabled) {
            logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);
        }

        try {
            boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
            DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, newTransaction: true, newSynchronization, debugEnabled, suspendedResources);
            prepareSynchronization(status, definition);
            return status;
        } catch (RuntimeException ex) {
            resume(transaction: null, suspendedResources);
            throw ex;
        }
    }
}
```

们先不 何判 事务 个  
们 关 不 事务则 创 个事务, 了prepareSynchronization , prepare 中  
了TransactionSynchronizationManager initSynchronization

```

} /**
 * Initialize transaction synchronization as appropriate.
 */
protected void prepareSynchronization(DefaultTransactionStatus status, TransactionDefinition definition) {
    if (status.isNewSynchronization()) {
        TransactionSynchronizationManager.setActualTransactionActive(status.hasTransaction());
        TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(
            definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT ?
                definition.getIsolationLevel() : null);
        TransactionSynchronizationManager.setCurrentTransactionReadOnly(definition.isReadOnly());
        TransactionSynchronizationManager.setCurrentTransactionName(definition.getName());
        TransactionSynchronizationManager.initSynchronization();
    }
}
}

```

们 了 创 个 事务 保 前 ThreadLocal synchronizations 了值

事务已 存在 况

但 们 关 事务 况, 为 事务 候也 创 个 事务  
 事务 候会 handleExistingTransaction 会 事务 传 决 何

```

397 /**
398  * Create a TransactionStatus for an existing transaction.
399  */
400 private TransactionStatus handleExistingTransaction(
401     TransactionDefinition definition, Object transaction, boolean debugEnabled)
402     throws TransactionException {
403
404     if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_NEVER) {
405         throw new IllegalStateException(
406             "Existing transaction found for transaction marked with propagation 'never'");
407     }
408
409     if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_NOT_SUPPORTED) {
410         if (debugEnabled) {
411             logger.debug("Suspending current transaction");
412         }
413         Object suspendedResources = suspend(transaction);
414         boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
415         return prepareTransactionStatus(
416             definition, null, false, newSynchronization, debugEnabled, suspendedResources);
417     }
418
419     if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRES_NEW) {
420         if (debugEnabled) {
421             logger.debug("Suspending current transaction, creating new transaction with name [" +
422                 definition.getName() + "]");
423         }
424         SuspendedResourcesHolder suspendedResources = suspend(transaction);
425         try {
426             boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
427             DefaultTransactionStatus status = newTransactionStatus(
428                 definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
429             doBegin(transaction, definition);
430             prepareSynchronization(status, definition);
431             return status;
432         } catch (RuntimeException | Error beginEx) {
433             resumeAfterBeginException(transaction, suspendedResources, beginEx);
434             throw beginEx;
435         }
436     }
437
438     if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_NESTED) {
439         if (!isNestedTransactionAllowed()) {
440             throw new NestedTransactionNotSupportedException(
441                 "Transaction manager does not allow nested transactions by default - " +
442                 "specify 'nestedTransactionAllowed' property with value 'true'");
443         }
444         if (debugEnabled) {
445             logger.debug("Creating nested transaction with name [" + definition.getName() + "]");
446         }
447         if (useSavepointForNestedTransaction()) {
448             // Create savepoint within existing Spring-managed transaction,
449             // through the SavepointManager API implemented by TransactionStatus.
450             // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.
451             DefaultTransactionStatus status =
452                 prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);
453             status.createAndHoldSavepoint();
454             return status;
455         } else {
456             // Nested transaction through nested begin and commit/rollback calls.
457             // Usually only for JTA: Spring synchronization might get activated here
458             // in case of a pre-existing JTA transaction.
459             boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
460             DefaultTransactionStatus status = newTransactionStatus(
461                 definition, transaction, true, newSynchronization, debugEnabled, null);
462             doBegin(transaction, definition);
463             prepareSynchronization(status, definition);
464             return status;
465         }
466     }
467 }

```

Page 5 of 14

File - D:\repository\org\springframework\spring-tx\5.0.10.RELEASE\spring-tx-5.0.10.RELEASE-sources.jar\org\springframework\transaction\support\AbstractPlatformTx

```

471 // Assumably PROPAGATION_SUPPORTS or PROPAGATION_REQUIRED.
472 if (debugEnabled) {
473     logger.debug("Participating in existing transaction");
474 }
475 if (isValidExistingTransaction()) {
476     if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT) {
477         Integer currentIsolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();
478         if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {
479             Constants isoConstants = DefaultTransactionDefinition.constants;
480             throw new IllegalStateException("Participating transaction with definition [" +
481                 definition + "] specifies isolation level which is incompatible with existing transaction: " +
482                 (currentIsolationLevel != null ?
483                     isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.PREFIX_ISOLATION) :
484                     "(unknown)"));
485         }
486     }
487     if (!definition.isReadOnly()) {
488         if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
489             throw new IllegalStateException("Participating transaction with definition [" +
490                 definition + "] is not marked as read-only but existing transaction is");
491         }
492     }
493 }
494 boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
495 return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null);
496 }
497
498 /**

```

上代中们到传为require\_new会创一个事务,上newTransactionStatus候传为true且创了一个事务之了prepareSynchronization,中保了TransactionSynchronizationManager中TreadLocal

## 2: DefaultTransactionStatus

DefaultTransactionStatus status = newTransactionStatus(  
definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);

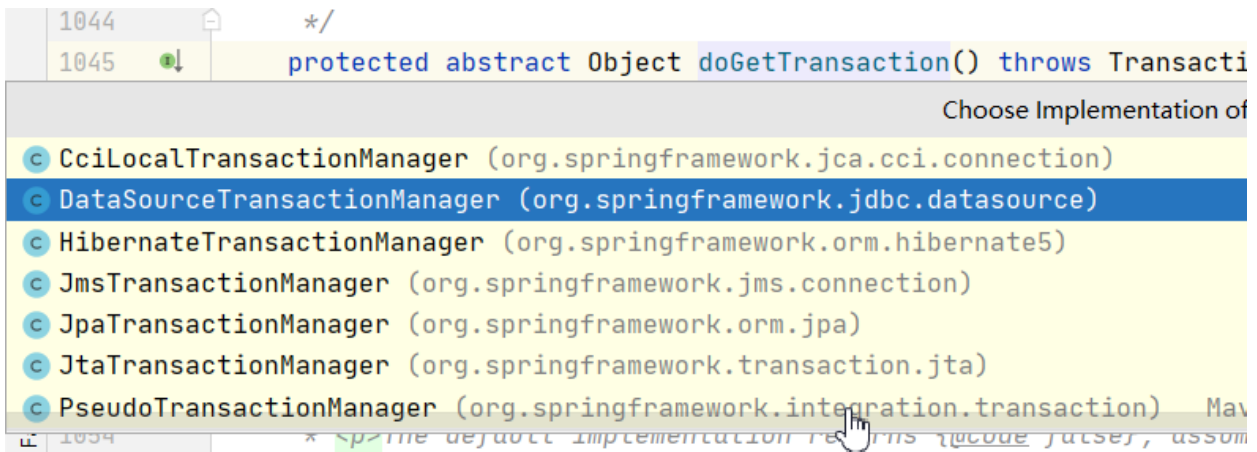
```
public class DefaultTransactionStatus extends AbstractTransactionStatus {  
  
    @Nullable  
    private final Object transaction;  
  
    private final boolean newTransaction;  
  
    private final boolean newSynchronization;  
  
    private final boolean readOnly;  
  
    private final boolean debug;  
  
    @Nullable  
    private final Object suspendedResources;  
}
```

仅当newTransaction属为true候一个事务  
且DefaultTransactionStatus中保了事务,们前事务信以  
TransactionStatus

## 2: 我们意到在Spring中事务对!使了Object,为什么不个口型?

getTransaction中先了doGetTransaction分,事务候们前  
事务,不则创,则会前事务为什么getTransaction先了  
doGetTransaction?

```
@Override  
public final TransactionStatus getTransaction(@Nullable Tra  
Object transaction = doGetTransaction();
```



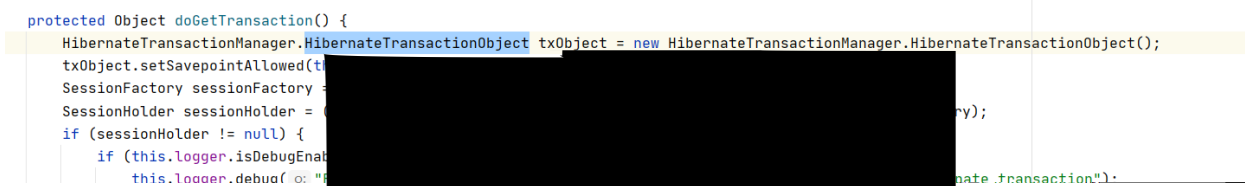
DataSourceTransactionManager 中 new 了 个 DataSourceTransactionObject 作为事务对象，他  
前 事务  
也 个 事务 以 个 Spring 中 事务 (DataSourceTransactionObject)，也 不  
DataSourceTransactionObject 会 事务 Spring 中 事务 中 事务不  
价



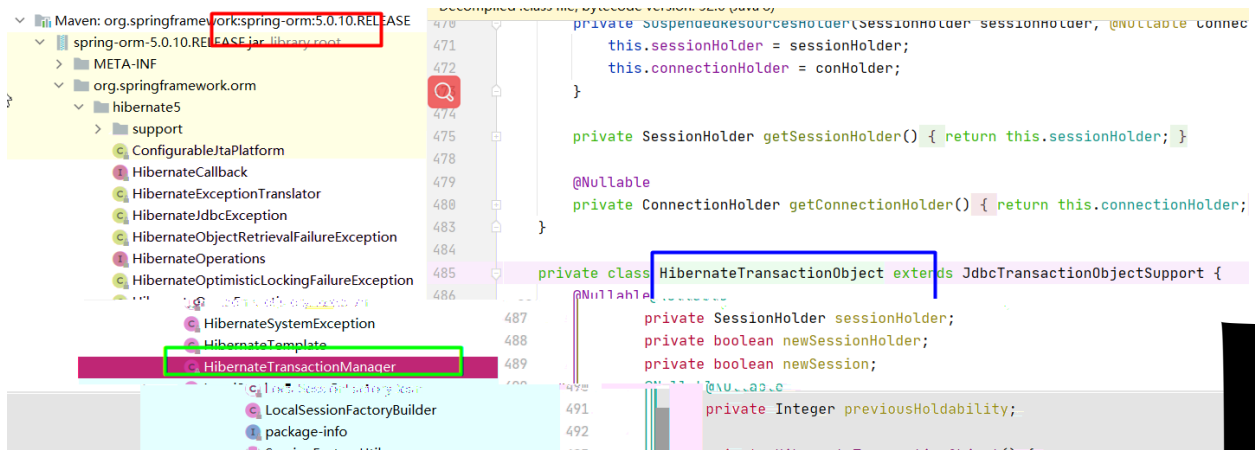
个 DataSourceTransactionObject spring-jdbc



HibernateTransactionManager 中，使 了 HibernateTransactionObject 作为事务



个事务 spring-orm



Jpa中JpaTransactionManager使用JpaTransactionObject作为事务，一个事务 spring-orm



我们 PlatformTransactionManager doGetTransaction 中 new了一个事务

事务 id不不他们不事务 id不 一个事务，一个事务 以一个事务中

你到了时候，两个事务互，么你何位两个事务？从上中们到了两个会创事务（准3个），三个以了



```

40 @Override
41 public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
42     Object transaction = doGetTransaction();
43
44     // Cache debug flag to avoid repeated checks.
45     boolean debugEnabled = logger.isDebugEnabled();
46
47     if (definition == null) {
48         // Use defaults if no transaction definition given.
49         definition = new DefaultTransactionDefinition();
50     }
51
52     if (isExistingTransaction(transaction)) {
53         // Existing transaction found -> check propagation behavior to find out how to behave.
54         return handleExistingTransaction(definition, transaction, debugEnabled);
55     }
56
57     // Check definition settings for new transaction.
58     if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
59         throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());
60     }
61
62     // No existing transaction found -> check propagation behavior to find out how to proceed.
63     if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_MANDATORY) {
64         throw new IllegalTransactionStateException(
65             "No existing transaction found for transaction marked with propagation 'mandatory'");
66     }
67     else if (definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRED ||
68             definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_REQUIRES_NEW ||
69             definition.getPropagationBehavior() == TransactionDefinition.PROPGATION_NESTED) {
70         SuspendedResourcesHolder suspendedResources = suspend(null);
71         if (debugEnabled) {
72             logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);
73         }
74         try {
75             boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
76             DefaultTransactionStatus status = newTransactionStatus(
77                 definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
78             prepareSynchronization(status, definition);
79             return status;
80         }
81         catch (RuntimeException | Error ex) {
82             resume(null, suspendedResources);
83             throw ex;
84         }
85     }
86     else {
87         // Create "empty" transaction: no actual transaction, but potentially synchronized.
88         if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT && logger.isWarnEnabled()) {
89             logger.warn("Custom isolation level specified but no actual transaction initiated; " +
90                 "isolation level will effectively be ignored: " + definition);
91         }
92         boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_ALWAYS);
93         return prepareTransactionStatus(definition, null, false, newSynchronization, debugEnabled, null);
94     }
95 }
96
97

```

handle方法中会根据传播属性  
考虑是否创建新的事务

第二个地方

第三个地方很少用到

(1) 不事务候创 TransactionStatus 候传 false, 但事事务前  
会个 TransactionStatus 事务 ( DataSourceTransactionObject)

```

return prepareTransactionStatus(
    definition, transaction: null, newTransaction: false, newSynchronization, debugEnabled, suspendedResources);

```

//

```

protected final DefaultTransactionStatus prepareTransactionStatus(
    TransactionDefinition definition, @Nullable Object transaction, boolean newTransaction,
    boolean newSynchronization, boolean debug, @Nullable Object suspendedResources) {

    DefaultTransactionStatus status = newTransactionStatus(
        definition, transaction, newTransaction, newSynchronization, debug, suspendedResources);
    prepareSynchronization(status, definition);
    return status;
}

```

//

(2) newTransaction 之前代 :

```
boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
```



```

/**
 * Always activate transaction synchronization, even for "empty" transactions
 * that result from PROPAGATION_SUPPORTS with no existing backend transaction.
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_SUPPORTS
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_NOT_SUPPORTED
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_NEVER
 */
public static final int SYNCHRONIZATION_ALWAYS = 0;

/**
 * Activate transaction synchronization only for actual transactions,
 * that is, not for empty ones that result from PROPAGATION_SUPPORTS with
 * no existing backend transaction.
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_REQUIRED
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_MANDATORY
 * @see org.springframework.transaction.TransactionDefinition#PROPAGATION_REQUIRES_NEW
 */
public static final int SYNCHRONIZATION_ON_ACTUAL_TRANSACTION = 1;

/**
 * Never active transaction synchronization, not even for actual transactions.
 */
public static final int SYNCHRONIZATION_NEVER = 2;

```

默认值

getTransactionSynchronization 值 0, 况下newSynchronization 为true,  
从 DefaultTransactionStatus newSynchronization 为true

们 下 个newSynchronization 使 到, 下 些 中 了isNewSynchronization

```

✓ * m AbstractPlatformTransactionManager.prepareSynchronization(DefaultTransactionStatus
> m AbstractPlatformTransactionManager.triggerAfterCommit(DefaultTransactionStatus) (c
> m AbstractPlatformTransactionManager.triggerBeforeCompletion(DefaultTransactionStatus)
> m AbstractPlatformTransactionManager.triggerBeforeCommit(DefaultTransactionStatus)
> m AbstractPlatformTransactionManager.cleanupAfterCompletion(DefaultTransactionStatus)
> m AbstractPlatformTransactionManager.triggerAfterCompletion(DefaultTransactionStatus)
m UOWActionAdapter in WebSphereUowTransactionManager.run() (org.springframework

```

prepareSynchronization newTransaction newTransaction为true 事务 doBegin之  
prepareSynchronization

```

/**
 * Initialize transaction synchronization as appropriate.
 */
protected void prepareSynchronization(DefaultTransactionStatus status, TransactionDefinition definition) {
    if (status.isNewSynchronization()) {
        TransactionSynchronizationManager.setActualTransactionActive(status.hasTransaction());
        TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(
            definition.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT ?
            definition.getIsolationLevel() : null);
        TransactionSynchronizationManager.setCurrentTransactionReadOnly(definition.isReadOnly());
        TransactionSynchronizationManager.setCurrentTransactionName(definition.getName());
        TransactionSynchronizationManager.initSynchronization();
    }
}

```

triggerAfterCommit中会 isNewSynchronization, 了Synchronization, 则

```

/**
 * Trigger {@code afterCommit} callbacks.
 * @param status object representing the transaction
 */
private void triggerAfterCommit(DefaultTransactionStatus status) {
    if (status.isNewSynchronization()) {
        if (status.isDebugEnabled()) {
            logger.trace("Triggering afterCommit synchronization");
        }
        TransactionSynchronizationUtils.triggerAfterCommit();
    }
}

```

TransactionSynchronizationUtils triggerAfterCommit 会 一个TransactionSynchronization afterCommit

```

/**
 * public static void invokeAfterCommit(@Nullable List<TransactionSynchronization> synchronizations) {
 *     if (synchronizations != null) {
 *         for (TransactionSynchronization synchronization : synchronizations) {
 *             synchronization.afterCommit();
 *         }
 *     }
 * }

```

： new了 TransactionStatus之后 会做什么 ？

事务了， 么 们 下TransactionStatus中 newTransaction 何 使 ？

```

DefaultTransactionStatus status = new TransactionStatus(connection, definition, transaction, newTransaction: true, newSynchronization, debugEnabled, suspendedRe
sources);
doBegin(transaction, definition);
prepareSynchronization(status, definition);

```

doBegin 创 TransactionStatus 且 newTransaction为true 候 会 ， 也

doBegin 会 中 个事务

们 MySQL中 事务 使

- **BEGIN** 个事务
- **ROLLBACK** 事务
- **COMMIT** 事务

么在doBegin中 否 具体 sql 句 ？ 事实上JDBC 式事务中我们 取了Connection, autoCommit为false, commit, 并 到具体 库 begin 句 代 在 PlatformTransactionManager 具体实 如DataSourceTransactionManager doBegin 中我们 到 取connection autoCommit属 否为只 属 ， 到 库 begin 句开启事务

```

1 Connection conn = DriverManager.getConnection(...);
2 try{
3     con.setAutoCommit(false);
4     Statement stmt = con.createStatement();
5     //1 or more queries or updates
6     con.commit();
7 }catch(Exception e){
8     con.rollback();
9 }finally{
1    con.close();
10 }

```

```

/**
 * Begin a new transaction with semantics according to the given transaction
 * definition. Does not have to care about applying the propagation behavior,
 * as this has already been handled by this abstract manager.
 * <p>This method gets called when the transaction manager has decided to actually
 * start a new transaction. Either there wasn't any transaction before, or the
 * previous transaction has been suspended.
 * <p>A special scenario is a nested transaction without savepoint: If
 * {@code useSavepointForNestedTransaction()} returns "false", this method
 * will be called to start a nested transaction when necessary. In such a context,
 * there will be an active transaction: The implementation of this method has
 * to detect this and start an appropriate nested transaction.
 * @param transaction transaction object returned by {@code doGetTransaction}
 * @param definition TransactionDefinition instance, describing propagation
 * behavior, isolation level, read-only flag, timeout, and transaction name
 * @throws TransactionException in case of creation or system errors
 */
protected abstract void doBegin(Object transaction, TransactionDefinition definition)
    throws TransactionException;

```

根据给定的食物语definition，开启一个新的事务（注意是新的事务，数据库层面的事务）。不需要关心应用传播行为，因为这已经被这个抽象管理器处理了

当事务管理器决定实际调用该方法时启动一个新的事务。要么之前没有任何交易，要么先前的交易已被暂停

一个特殊的场景是没有保存点的嵌套事务: If {@code useSavepointForNestedTransaction()} 返回“false”，这个方法会在必要时被调用来启动一个嵌套事务。在这种情况下，会有一个活动的事务: 这个方法的实现有检测并启动一个适当的嵌套事务。

doBegin PlatformTransactionManager中 个 ，具体 决于不 TransactionManager  
DataSourceTransactionManager中 具体

```

152  * THIS IMPLEMENTATION SETS THE ISOLATION LEVEL BUT IGNORES THE TIMEOUT.
153  */
154  @Override
155  protected void doBegin(Object transaction, TransactionDefinition definition) {
156      DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
157      Connection con = null;
158
159      try {
160          if (!txObject.hasConnectionHolder() ||
161              txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
162              Connection newCon = obtainDataSource().getConnection();
163              if (logger.isDebugEnabled()) {
164                  logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");
165              }
166              txObject.setConnectionHolder(new ConnectionHolder(newCon, true));
167          }
168      }

```

获取具体的connection对象

Page 2 of 4

```

169      txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
170      con = txObject.getConnectionHolder().getConnection();
171
172      Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);
173      txObject.setPreviousIsolationLevel(previousIsolationLevel);
174
175      // Switch to manual commit if necessary. This is very expensive in some JDBC drivers,
176      // so we don't want to do it unnecessarily (for example if we've explicitly
177      // configured the connection pool to set it already).
178      if (con.getAutoCommit()) {
179          txObject.setMustRestoreAutoCommit(true);
180          if (logger.isDebugEnabled()) {
181              logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
182          }
183          con.setAutoCommit(false);
184      }
185
186      prepareTransactionalConnection(con, definition);
187      txObject.getConnectionHolder().setTransactionActive(true);
188
189      int timeout = determineTimeout(definition);
190      if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
191          txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
192      }
193
194      // Bind the connection holder to the thread.
195      if (txObject.isNewConnectionHolder()) {
196          ThreadLocal<ConnectionHolder> threadLocal = DataSourceUtils.findThreadLocal();
197          if (threadLocal == null) {
198              threadLocal = new ThreadLocal<>();
199          }
200          threadLocal.set(txObject.getConnectionHolder());
201      }
202
203      catch (Throwable ex) {
204          if (txObject.isNewConnectionHolder()) {
205              DataSourceUtils.releaseConnection(con, obtainDataSource());
206              txObject.setConnectionHolder(null, false);
207          }
208
209          throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);
210      }
211  }

```

将 DataSource 和对应的connection放置到ThreadLocal中， ThreadLocal的value是一个Map， key为DataSource， value为connection

```

*/
protected void prepareTransactionalConnection(Connection con, TransactionDefinition definition) {
    Statement stmt = con.createStatement();
    stmt.executeUpdate("SET TRANSACTION READ ONLY");
}

```

上 doBegin 了bindResource, bindResource 中 DataSource connection作为 value 储到ThreadLocal中, 个ThreadLocal value 个Map, key为 DataSource, value为connection bindResource 具体 到了 个 为resources ThreadLocal中

private static final ThreadLocal<Map<Object, Object>> resources =  
new NamedThreadLocal<>("Transactional resources");

```

private static final ThreadLocal<Map<Object, Object>> resources =
    new NamedThreadLocal<>("Transactional resources");
*/
public static void bindResource(Object key, Object value) throws IllegalStateException {
    Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
    Assert.notNull(value, message: "Value must not be null");
    Map<Object, Object> map = resources.get();
}

```

## 事务 和 复

上 分 中 们 到了doBegin , 么 们 分 下 了事务 , 事务 传  
 为require\_new, spring 何创 事务, 何 事务, 事务 交 何 到 事务 , 事  
 事务之 关 何保 ?

```

/**
 * Create a TransactionStatus for an existing transaction.
 */
private TransactionStatus handleExistingTransaction(
    TransactionDefinition definition, Object transaction, boolean debugEnabled)
    throws TransactionException {

    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NEVER) {
        throw new IllegalTransactionStateException(
            "Existing transaction found for transaction marked with propagation 'never'");
    }

    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NOT_SUPPORTED) {
        if (debugEnabled) {
            logger.debug("Suspending current transaction");
        }
        Object suspendedResources = suspend(transaction);
        boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
        return prepareTransactionStatus(
            definition, null, false, newSynchronization, debugEnabled, suspendedResources);
    }

    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW) {
        if (debugEnabled) {
            logger.debug("Suspending current transaction, creating new transaction with name [" +
                definition.getName() + "]");
        }
        SuspendedResourcesHolder suspendedResources = suspend(transaction);
        try {
            boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
            DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
            doBegin(transaction, definition);
            prepareSynchronization(status, definition);
            return status;
        } catch (RuntimeException | Error beginEx) {
            // resumeAfterBegin
            throw beginEx;
        }
    }
}

```

handleExistingTransaction 中判 别 requires\_new则会创 事务 先 前事务

```

1  */
2  @Nullable
3  protected final SuspendedResourcesHolder suspend(@Nullable Object transaction) throws TransactionException {
4      if (TransactionSynchronizationManager.isSynchronizationActive()) {
5          List<TransactionSynchronization> suspendedSynchronizations = doSuspendSynchronization();
6          try {
7              Object suspendedResources = null;
8              if (transaction != null) {
9                  suspendedResources = doSuspend(transaction);
10             }
11             String name = TransactionSynchronizationManager.getCurrentTransactionName();
12             TransactionSynchronizationManager.setCurrentTransactionName(null);
13             boolean readOnly = TransactionSynchronizationManager.isCurrentTransactionReadOnly();
14             TransactionSynchronizationManager.setCurrentTransactionReadOnly(false);
15             Integer isolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();
16             TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(null);
17             boolean wasActive = TransactionSynchronizationManager.isActualTransactionActive();
18             TransactionSynchronizationManager.setActualTransactionActive(false);
19             return new SuspendedResourcesHolder(
20                 suspendedResources, suspendedSynchronizations, name, readOnly, isolationLevel, wasActive);
21         }
22         catch (RuntimeException | Error ex) {
23             // doSuspend failed - original transaction is still active...
24             doResumeSynchronization(suspendedSynchronizations);
25             throw ex;
26         }
27     }
28     else if (transaction != null) {
29         // Transaction active but no synchronization active.
30         Object suspendedResources = doSuspend(transaction);
31         return new SuspendedResourcesHolder(suspendedResources);
32     }
33 }

```

DataSourceTransactionManager的doSuspend实现

```

@Override
protected Object doSuspend(Object transaction) {
    DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
    txObject.setConnectionHolder(null);
    return TransactionSynchronizationManager.unbindResource(observeOnDataSource());
}

```

```

/**
 * Unbind a resource for the given key from the current thread.
 * @param key the key to unbind (usually the resource factory)
 * @return the previously bound value (usually the active resource object)
 * @throws IllegalStateException if there is no value bound to the thread
 * @see ResourceTransactionManager#getResourceFactory()
 */
public static Object unbindResource(Object key) throws IllegalStateException {
    Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
    Object value = doUnbindResource(actualKey);
    if (value == null) {
        throw new IllegalStateException(
            "No value for key [" + actualKey + "] bound to thread [" + Thread.currentThread().getName() + "]);
    }
    return value;
}

```



```

/**
 * Actually remove the value of the resource that is bound for the given key.
 */
@Nullable
private static Object doUnbindResource(Object actualKey) {
    Map<Object, Object> map = resources.get();
    if (map == null) {
        return null;
    }
    Object value = map.remove(actualKey);
    // Remove entire ThreadLocal if empty...
    if (map.isEmpty()) {
        resources.remove();
    }
    // Transparently suppress a ResourceHolder that was marked as void...
    if (value instanceof ResourceHolder && ((ResourceHolder) value).isVoid()) {
        value = null;
    }
    if (value != null && logger.isTraceEnabled()) {
        logger.trace("Removed value [" + value + "] for key [" + actualKey + "] from thread [" +
            Thread.currentThread().getName() + "]);");
    }
    return value;
}

```

获取当前线程的 resources 这个ThreadLocal 中的value, 这个value是一个map  
根据DataSource作为key获取到Object value是一个connection

从上代中我们到doSuspend前事务, 也前事务 connection, 到  
SuspendResourceHolder中  
创建DefaultTransactionStatus 时候 一个SuspendResourceHolder传入  
一个事务 时候会 SuspendResourceHolder 到上一个事务  
DefaultTransactionStatus getSuspendResourceHolder 下, 从下 们以 到 commit  
rolla 到上 前从判 tSuspendResourceHol 下 中 们 到ndRs er J



```

    */
    protected final void resume(@Nullable Object transaction, @Nullable SuspendedResourcesHolder resourcesHolder)
        throws IllegalStateException {
        if (resourcesHolder != null) {
            Object suspendedResources = resourcesHolder.suspendedResources;
            if (suspendedResources != null) {
                doResume(transaction, suspendedResources);
            }
            List<TransactionSynchronization> suspendedSynchronizations = resourcesHolder.suspendedSynchronizations;
            if (suspendedSynchronizations != null) {
                TransactionSynchronizationManager.setActualTransactionActive(resourcesHolder.wasActive);
                TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(resourcesHolder.isolationLevel);
                TransactionSynchronizationManager.setCurrentTransactionReadOnly(resourcesHolder.readOnly);
                TransactionSynchronizationManager.setCurrentTransactionName(resourcesHolder.name);
                doResumeSynchronization(suspendedSynchronizations);
            }
        }
    }

    @Override
    protected void doResume(@Nullable Object transaction, Object suspendedResources) {
        TransactionSynchronizationManager.bindResource(obtainDataSource(), suspendedResources);
    }

    /**
     * @see ResourceTransactionManager#getResourceHolderKey()
     */
    public static void bindResource(Object key, Object value) throws IllegalStateException {
        Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
        Assert.notNull(value, message: "Value must not be null");
        Map<Object, Object> map = resources.get();
        // set ThreadLocal Map if none found
        if (map == null) {
            map = new HashMap<>();
            resources.set(map);
        }
        Object oldValue = map.put(actualKey, value);
        // Transparently suppress a ResourceHolder that was marked as void...
        if (oldValue instanceof ResourceHolder && ((ResourceHolder) oldValue).isVoid()) {
            oldValue = null;
        }
        if (oldValue != null) {
            throw new IllegalStateException("Already value [" + oldValue + "] for key [" +
                actualKey + "] bound to thread [" + Thread.currentThread().getName() + "]");
        }
        if (logger.isTraceEnabled()) {
            logger.trace("Bound value [" + value + "] for key [" + actualKey + "] to thread [" +
                Thread.currentThread().getName() + "]");
        }
    }
}

```

从上面代码中我们可以看到，在resume方法中，具体是如何将connection以DataSource作为key放入到resources这个ThreadLocal中的。下面我们来看一下：

- (1) 当前没有事务，则newTransactionStatus会创建一个connection，<dataSource,connection>值放入到resources这个ThreadLocal的value中。
- (2) 当前有事务，在事务情况下，先在resources中的value的map中找到DataSource，然后创建一个connection放入到resources中。
- (3) 如当前存在事务，但不是事务的情况下，应重用两个事务会共用一个connection。在mysql实现中：一个Connection对应一个事务，应重用Object transaction对同一个Connection则重用。有些transaction在同一个数据库事务中，一个jdbc connection对应一个事务，取一个connection就意味着创建了一个事务。在数据库中我们常使用库提供begin命令开启事务，但在应用层不创建connection就使用一个socket。

定 入 个JDBC Connection 对应 个事务,因 当开启事务 候 加你当前事务对应  
<dataSource,connection>从 resources ThreadLocal中 ,封 到SuspendResourceHolder中 交  
开启 事务, 开启 事务 完成之后 resume 复之前 事务, 就 SuspendResourceHolder中封  
<DataSource,Connection>再 入到resource ThreadLocal中

么 又 了, Connection对I 已 了事务, spring为什么 对MySQL 供  
org.springframework.jdbc.datasource.DataSourceTransactionManager.DataSourceTransactionObject 作为  
事务对I ?

```
/**
 * @Override
 * public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {
 *     Object transaction = doGetTransaction();
 *     @Override
 *     protected Object doGetTransaction() {
 *         DataSourceTransactionObject txObject = new DataSourceTransactionObject();
 *         txObject.setSavepointAllowed(isNestedTransactionAllowed());
 *         ConnectionHolder conHolder = DataSourceTransactionManager中的实现
 *             (ConnectionHolder) TransactionSynchronizationManager.getResource(obtainDataSource);
 *         txObject.setConnectionHolder(conHolder, newConnectionHolder: false);
 *         return txObject;
 *     }
 * }
```

```
/**
 * DataSource transaction object, representing a ConnectionHolder.
 * Used as transaction object by DataSourceTransactionManager.
 */
```

```
private static class DataSourceTransactionObject extends JdbcTransactionObjectSupport {
```

```
private boolean newConnectionHolder;
```

```
private boolean mustRestoreAutoCommit;
```

```
public void setConnectionHolder(@Nullable ConnectionHolder connectionHolder, boolean newConnectionHolder) {
    super.setConnectionHolder(connectionHolder);
    this.newConnectionHolder = newConnectionHolder; 接受connectionHolder, 这个holder中存在
                                                         Connection
}
```

```
public boolean isNewConnectionHolder() {
    return this.newConnectionHolder;
}
当时新创建的 Connection的时候 newConnectionHolder就为
true
因此 : 多个DataSourceTransactionObject对象可以持有相同的
ConnectionHolder, 也就是相同的Connection
```

```
public void setMustRestoreAutoCommit(boolean mustRestoreAutoCommit) {
    this.mustRestoreAutoCommit = mustRestoreAutoCommit;
}
```

```
public boolean isMustRestoreAutoCommit() { return this.mustRestoreAutoCommit; }
```

```
public void setRollbackOnly() { getConnectionHolder().setRollbackOnly(); }
```

```
@Override
public boolean isRollbackOnly() { return getConnectionHolder().isRollbackOnly(); }
```

```
@Override
```

```

/**
 * Convenient base class for JDBC-aware transaction objects. Can contain a
 * {@link ConnectionHolder} with a JDBC {@code Connection}, and implements the
 * {@link SavepointManager} interface based on that {@code ConnectionHolder}.
 *
 * <p>Allows for programmatic management of JDBC {@link java.sql.Savepoint Savepoints}.
 * Spring's {@link org.springframework.transaction.support.DefaultTransactionStatus}
 * automatically delegates to this, as it autodetects transaction objects which
 * implement the {@link SavepointManager} interface.
 *
 * @author Juergen Hoeller
 * @since 1.1
 * @see DataSourceTransactionManager
 */
public abstract class JdbcTransactionObjectSupport implements SavepointManager, TransactionObject {

    private static final Log logger = LoggerFactory.getLog(JdbcTransactionObjectSupport.class);

    @Nullable
    private ConnectionHolder connectionHolder;

    @Nullable
    private Integer previousIsolationLevel;

    private boolean savepointAllowed = false;
}

```

们了 到了spring中 供 DataSourceTransactionObject 主 Savepointmanager功

spring DataSourceTransactionManager 供了

HibernateTransactionManager 中使 HibernateTransactionObject作为事务 , 个  
 HibernateTransactionObject也 JdbcTransactionObjectSupport, 从 具 了  
 savePoint 功

```

protected Object doGetTransaction() {
    HibernateTransactionManager.HibernateTransactionObject txObject = new HibernateTransactionManager.HibernateTransactionObject();
    txObject.setSavepointAllowed(this.isNestedTransactionAllowed());

    private class HibernateTransactionObject extends JdbcTransactionObjectSupport {
        @Nullable
        private SessionHolder sessionHolder;
        private boolean newSessionHolder;
    }
}

```

JpaTransactionManager 使 JpaTransactionObject作为 事务 个 也  
 JdbcTransactionObjectSupport

```

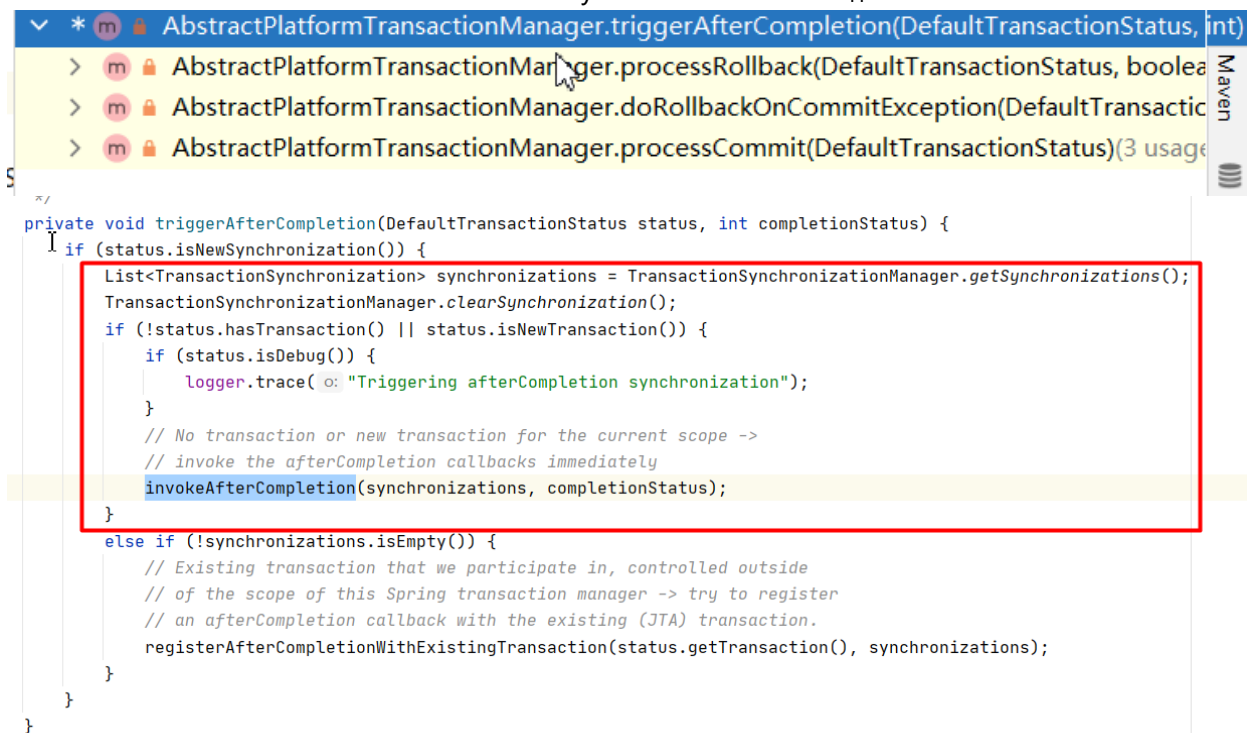
protected Object doGetTransaction() {
    JpaTransactionManager.JpaTransactionObject txObject = new JpaTransactionManager.JpaTransactionObject();
}

```

```
private class JpaTransactionObject extends JdbcTransactionObjectSupport {
    @Nullable
    private EntityManagerHolder entityManagerHolder;
    private boolean newEntityManagerHolder;
    @Nullable
    private Object transactionData;

    private JpaTransactionObject() {
    }
}
```

## TransactionSynchronization 作



```

> AbstractPlatformTransactionManager.triggerAfterCompletion(DefaultTransactionStatus, int)
> AbstractPlatformTransactionManager.processRollback(DefaultTransactionStatus, boolean)
> AbstractPlatformTransactionManager.doRollbackOnCommitException(DefaultTransactionStatus, boolean)
> AbstractPlatformTransactionManager.processCommit(DefaultTransactionStatus) (3 usages)

private void triggerAfterCompletion(DefaultTransactionStatus status, int completionStatus) {
    if (status.isNewSynchronization()) {
        List<TransactionSynchronization> synchronizations = TransactionSynchronizationManager.getSynchronizations();
        TransactionSynchronizationManager.clearSynchronization();
        if (!status.hasTransaction() || status.isNewTransaction()) {
            if (status.isDebugEnabled()) {
                logger.trace("Triggering afterCompletion synchronization");
            }
            // No transaction or new transaction for the current scope ->
            // invoke the afterCompletion callbacks immediately
            invokeAfterCompletion(synchronizations, completionStatus);
        }
    }
    else if (!synchronizations.isEmpty()) {
        // Existing transaction that we participate in, controlled outside
        // of the scope of this Spring transaction manager -> try to register
        // an afterCompletion callback with the existing (JTA) transaction.
        registerAfterCompletionWithExistingTransaction(status.getTransaction(), synchronizations);
    }
}

```

先到 TransactionSynchronization, 个 TransactionSynchronization  
afterCompletion

1 List

```

9  throw new IllegalStateException("Cannot deactivate transaction synchroni
    zation - not active");
10 }
11 logger.trace("Clearing transaction synchronization");
12 synchronizations.remove();
13 }
14

```

从上 代 们 到 了ThreadLocal remove synchronizations.remove();

ansactionSynchronizationManager.java × ThreadLocal.java × TransactionSynchro

```

* @since 1.5
*/
public void remove() {
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        m.remove(key: this);
}

```

在线程的ThreadLocalMap中 将自身作为key 移除指定的 key value

```

717 /**
718  * Process an actual commit.
719  * Rollback-only flags have already been checked and applied.
720  * @param status object representing the transaction
721  * @throws TransactionException in case of commit failure
722  */
723 private void processCommit(DefaultTransactionStatus status) throws TransactionException {
724     try {
725         boolean beforeCompletionInvoked = false;
726
727         try {
728             boolean unexpectedRollback = false;
729             prepareForCommit(status);
730             triggerBeforeCommit(status);
731             triggerBeforeCompletion(status);
732             beforeCompletionInvoked = true;
733
734             if (status.hasSavepoint()) {
735                 if (status.isDebugEnabled()) {
736                     logger.debug("Releasing transaction savepoint");
737                 }
738                 unexpectedRollback = status.isGlobalRollbackOnly();
739                 status.releaseHeldSavepoint();
740             }
741             else if (status.isNewTransaction()) {
742                 if (status.isDebugEnabled()) {
743                     logger.debug("Initiating transaction commit");
744                 }
745                 unexpectedRollback = status.isGlobalRollbackOnly();
746                 doCommit(status);
747             }
748             else if (isFailEarlyOnGlobalRollbackOnly()) {
749                 unexpectedRollback = status.isGlobalRollbackOnly();
750             }
751
752             // Throw UnexpectedRollbackException if we have a global rollback-only
753             // marker but still didn't get a corresponding exception from commit.
754             if (unexpectedRollback) {
755                 throw new UnexpectedRollbackException(
756                     "Transaction silently rolled back because it has been marked as rollback-only");
757             }
758         }
759         catch (UnexpectedRollbackException ex) {
760             // can only be caused by doCommit
761             triggerAfterCompletion(status, TransactionSynchronization.STATUS_ROLLED_BACK);
762             throw ex;
763         }
764         catch (TransactionException ex) {
765             // can only be caused by doCommit
766             if (isRollbackOnCommitFailure()) {
767                 doRollbackOnCommitException(status, ex);
768             }
769             else {
770                 triggerAfterCompletion(status, TransactionSynchronization.STATUS_UNKNOWN);
771             }
772             throw ex;
773         }
774         catch (RuntimeException | Error ex) {
775             if (beforeCompletionInvoked) {
776                 triggerBeforeCompletion(status);
777             }
778             doRollbackOnCommitException(status, ex);
779             throw ex;
780         }
781     }
782     // Trigger afterCommit callbacks, with an exception thrown there
783     // propagated to callers but the transaction still considered as committed

```

在doCommit过程中如果出现了异常  
会执行triggerAfterCompletion

```

784     try {
785         triggerAfterCommit(status);
786     }
787     finally {
788         triggerAfterCompletion(status, TransactionSynchronization.STATUS_COMMITTED);
789     }
790 }
791
792 finally {
793     cleanupAfterCompletion(status);
794 }
795 }

```

这一定是triggerAfterCommit