根究底事务是如何开启的?

我们知 开启事务可以使用 start transaction， 么我 常好奇

（1）开启事务 提交事务 是在spring中执 的 是在jdbc 动中执 的?

（2）spring中或 jdbc的 动中 在哪 执 了start transaction? 什么时候执 的? getTransaction的时候 是begin的时候?

（3）在ConnectionImpl的commit方法实现中 会同 ConnectionImpl对 内 持有的NativeSession对 执 来发 条 commit 句来实现实物的提交

但是在ConnectionImpl中没有 starTransaction 样的方法开启 个事务

开启事务

：在spring中开启 个事务使用了startTransaction， 方法内 用了doBegin方法开启 个事务，在doBegin方法中，开启事务 个 义是如何体现的? doBegin方法中会执 start transaction sql 句吗?

```
/**
 * Start a new transaction.
 */
private TransactionStatus startTransaction(TransactionDefinition definition, Object transaction,
        boolean debugEnabled, @Nullable SuspendedResourcesHolder suspendedResources) {

    boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
    org.springframework.transaction.support.DefaultTransactionStatus status = newTransactionStatus(
            definition, transaction, newTransaction: true, newSynchronization, debugEnabled, suspendedResources);
    doBegin(transaction, definition);
    /**
     * 注意startTransaction方法中调用了prepareSynchronization 方法
     */
    prepareSynchronization(status, definition);
    return status;
}
```

DataSourceTransactionManager的doBegin实现

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

        try {
            if (!txObject.hasConnectionHolder() ||
                    txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                /**
                 *
                 * [illegible]connection
                 */
                Connection newCon = obtainDataSource().getConnection();
                if (logger.isDebugEnabled()) {
                    logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");
                }
                txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
            }

            txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
            con = txObject.getConnectionHolder().getConnection();

            /**
             *
             * [illegible]
             *  con.setTransactionIsolation(definition.getIsolationLevel());
             *
             *  [illegible]
             *
             */
            Integer previousIsolationLevel = org.springframework.jdbc.datasource.DataSourceUtils.prepareConnectionForTransaction(con, definition);
            txObject.setPreviousIsolationLevel(previousIsolationLevel);
            txObject.setReadOnly(definition.isReadOnly());

            // Switch to manual commit if necessary. This is very expensive in some JDBC drivers,
            // so we don't want to do it unnecessarily (for example if we've explicitly
            // configured the connection pool to set it already).
            // [illegible]JDBC [illegible]
          // [illegible]
          // [illegible]。
            /**
             * [illegible]
             *
             */
            if (con.getAutoCommit()) {
                txObject.setMustRestoreAutoCommit(true);
                if (logger.isDebugEnabled()) {
                    logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
                }
                con.setAutoCommit(false);
            }

            prepareTransactionalConnection(con, definition);
            txObject.getConnectionHolder().setTransactionActive(true);

            int timeout = determineTimeout(definition);
            if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
            }

            // Bind the connection holder to the thread.
            if (txObject.isNewConnectionHolder()) {
                TransactionSynchronizationManager.bindResource(obtainDataSource(), txObject.getConnectionHolder());
            }
        } catch (Throwable ex) {
            if (txObject.isNewConnectionHolder()) {
                org.springframework.jdbc.datasource.DataSourceUtils.releaseConnection(con, obtainDataSource());
                txObject.setConnectionHolder(null, false);
            }
            throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);
        }
    }
```

在 个方法中我们看到 先是 取Connection，然后 autoCommit为false， 在后 的方法prepareTransactionConnection方法中 判断是否是只 ，如果只 事务为只

```java
protected void prepareTransactionalConnection(Connection con, TransactionDefinition definition)
        throws SQLException {

    if (isEnforceReadOnly() && definition.isReadOnly()) {
        try (Statement stmt = con.createStatement()) {
            stmt.executeUpdate( sql: "SET TRANSACTION READ ONLY");
        }
    }
}
```

doBegin方法中并没有 执 “start transaction” 样的sql 句来显示开启 个事务， 且在数据库 动（mysql-connector-java）的实现中也搜不到 start transaction 样的sql 句， 么事务到底是怎么开启的 ？

原来，当你 取到 接的时候， 情况下每执 条 句 会 动创建 个事务，当我们 了autoCommit为false的情况下，多个sql 句 动合并到 个事务中，只有你执 了commit的时候事务才会提交，因此不 使用start transaction 句 参 下文官方文档

在NativeCat中 模拟如下场景：

（1）新建查 ，打开 个connection

（2）执 ： set autocommit0

（3）执 update

（4）查看数据并没有发现修改后的值

（5）执 commit 后发现修改后的值

就 了开启事务不 使用start transaction ,只 autoCommit为false， 在DataSourceTransactionManager的doBegin方法中就将autoCommit 为false 示开启了 个事务

> https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, the proprietor will also want to update the total amount sold to date. However, the amount sold per week and the total amount sold should be updated at the same time; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.

有些时候， 另 条 句完成，否则您不希望 条 句生效 例如，当the Coffee Break的所有 更新每周售出的咖 数 时， 所有 将希望更新到目前为止售出的总数 但是，每周的 售 和总 售 应 同时更新;否则会导 数据不 确保两个操作 发生或两个操作 不发生的方法是使用事务 事务是作为 个单元执 的 个或多个 句的合，因此 么执 所有 句， 么不执 任何 句

This page covers the following topics

- Disabling Auto-Commit Mode
- Committing Transactions
- Using Transactions to Preserve Data Integrity
- Setting and Rolling Back to Savepoints
- Releasing Savepoints
- When to Call Method rollback

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. (To be more precise, the default is for a SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.)

创建 接时，它处于 动提交模式 意味着每个单独的SQL 句 为 个事务，并在执 后立即 动提交 (更准确地 ， 是在SQL 句完成时提交， 不是在执 时提交 当检索到 句的所有 果 和更新 数时， 句就完成了 然 ，在几乎所有情况下， 句 是在执 之后完成并提交的)

The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where `con` is an active connection:

允 将两个或多个 句分组到 个事务中的方法是禁用 动提交模式 下 的代码演示了 点，其中con是 个活动接:

```
con.setAutoCommit(false);
```

因此也就是说 实 上数据库的事务不是我们手动开启的， 认 况下每 条sql语句 会有 个事务，在实 应用中我们 要做的 不是开启事务，而是将多个语句放置到 个事务中执行，这个是 过 将autoCommit设置为false来实现的

The statement `con.setAutoCommit(true);` enables auto-commit mode, which means that each statement is once again committed automatically when it is completed. Then, you are back to the default state where you do not have to call the method `commit` yourself. It is advisable to disable the auto-commit mode only during the transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

声明con.setAutoCommit(真正的);启用 动提交模式， 意味着每条 句在完成时再次 动提交 然后，您就回到 状态，不必 己 用commit方法 建 只在事务模式期 禁用 动提交模式 种方式，您可以 免为多个 句持有数据库 ， 增加了与其他用户发生冲突的可 性

使用事务来保持数据完整性

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. For instance, imagine that an employee was supposed to enter new coffee prices in the table `COFFEES` but delayed doing it for a few days. In the meantime, prices rose, and today the owner is in the process of entering the higher prices. The employee finally gets around to entering the now outdated prices at the same time that the owner is trying to update the table. After inserting the outdated prices, the employee realizes that they are no longer valid and calls the `Connection` method `rollback` to undo their effects. (The method `rollback` aborts a transaction and restores values to what they were before the attempted update.) At the same time, the owner is executing a `SELECT` statement and printing the new prices. In this situation, it is possible that the owner will print a price that had been rolled back to its previous value, making the printed price incorrect.

This kind of situation can be avoided by using transactions, providing some level of protection against conflicts that arise when two users access data at the same time.

To avoid conflicts during a transaction, a DBMS uses locks, mechanisms for blocking access by others to the data that is being accessed by the transaction. (Note that in auto-commit mode, where each statement is a transaction, locks are held for only one statement.) After a lock is set, it remains in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed. The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a *dirty read* because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)

How locks are set is determined by what is called a transaction isolation level, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules.

One example of a transaction isolation level is `TRANSACTION_READ_COMMITTED`, which will not allow a value to be accessed until after it has been committed. In other words, if the transaction isolation level is set to `TRANSACTION_READ_COMMITTED`, the DBMS does not allow dirty reads to occur. The interface `Connection` includes five values that represent the transaction isolation levels you can use in JDBC:

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
`TRANSACTION_NONE`	Not supported	*Not applicable*	*Not applicable*	*Not applicable*
`TRANSACTION_READ_COMMITTED`	Supported	Prevented	Allowed	Allowed

TRANSACTION_READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	Supported	Prevented	Prevented	Prevented

A *non-repeatable read* occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.
A *phantom read* occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.
Usually, you do not need to do anything about the transaction isolation level; you can just use the default one for your DBMS. The default transaction isolation level depends on your DBMS. For example, for Java DB, it is `TRANSACTION_READ_COMMITTED`. JDBC allows you to find out what transaction isolation level your DBMS is set to (using the `Connection` method `getTransactionIsolation`) and also allows you to set it to another level (using the `Connection` method `setTransactionIsolation`).

: A JDBC driver might not support all transaction isolation levels. If a driver does not support the isolation level specified in an invocation of `setTransactionIsolation`, the driver can substitute a higher, more restrictive transaction isolation level. If a driver cannot substitute a higher transaction level, it throws a `SQLException`. Use the method `DatabaseMetaData.supportsTransactionIsolationLevel` to determine whether or not the driver supports a given level.

The method `Connection.setSavepoint`, sets a `Savepoint` object within the current transaction.
The `Connection.rollback` method is overloaded to take a `Savepoint` argument.
The following method, `CoffeesTable.modifyPricesByPercentage`, raises the price of a particular coffee by a percentage, `priceModifier`. However, if the new price is greater than a specified price, `maximumPrice`, then the price is reverted to the original price:

```
  public void modifyPricesByPercentage(
    String coffeeName,
    float priceModifier,
    float maximumPrice) throws SQLException {
    con.setAutoCommit(false);
    ResultSet rs = null;
    String priceQuery = "SELECT COF_NAME, PRICE FROM COFFEES " +
                        "WHERE COF_NAME = ?";
    String updateQuery = "UPDATE COFFEES SET PRICE = ? " +
                         "WHERE COF_NAME = ?";
    try (PreparedStatement getPrice = con.prepareStatement(priceQuery,
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
         PreparedStatement updatePrice = con.prepareStatement(updateQuery))
    {
      Savepoint save1 = con.setSavepoint();
      getPrice.setString(1, coffeeName);
      if (!getPrice.execute()) {
        System.out.println("Could not find entry for coffee named " + coffeeName);
```

```
      } else {
        rs = getPrice.getResultSet();
        rs.first();
        float oldPrice = rs.getFloat("PRICE");
        float newPrice = oldPrice + (oldPrice * priceModifier);
        System.out.printf("Old price of %s is $%.2f%n", coffeeName, oldPrice);
        System.out.printf("New price of %s is $%.2f%n", coffeeName, newPrice);
        System.out.println("Performing update...");
        updatePrice.setFloat(1, newPrice);
        updatePrice.setString(2, coffeeName);
        updatePrice.executeUpdate();
        System.out.println("\nCOFFEES table after update:");
        CoffeesTable.viewTable(con);
        if (newPrice > maximumPrice) {
          System.out.printf("The new price, $%.2f, is greater " +
                            "than the maximum price, $%.2f. " +
                            "Rolling back the transaction...%n",
                            newPrice, maximumPrice);
          con.rollback(save1);
          System.out.println("\nCOFFEES table after rollback:");
          CoffeesTable.viewTable(con);
        }
        con.commit();
      }
    } catch (SQLException e) {
      JDBCTutorialUtilities.printSQLException(e);
    } finally {
      con.setAutoCommit(true);
    }
  }
```

The following statement specifies that the cursor of the `ResultSet` object generated from the `getPrice` query is closed when the `commit` method is called. Note that if your DBMs does not
support `ResultSet.CLOSE_CURSORS_AT_COMMIT`, then this constant is ignored:

```
getPrice = con.prepareStatement(query, ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The method begins by creating a `Savepoint` with the following statement:

```
Savepoint save1 = con.setSavepoint();
```

The method checks if the new price is greater than the `maximumPrice` value. If so, the method rolls back the transaction with the following statement:

```
con.rollback(save1);
```

Consequently, when the method commits the transaction by calling the `Connection.commit` method, it will not commit any rows whose associated `Savepoint` has been rolled back; it will commit all the other updated rows.

The method `Connection.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.

After a savepoint has been released, attempting to reference it in a rollback operation causes a `SQLException` to be thrown. Any savepoints that have been created in a transaction are automatically released and become invalid when the

transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

As mentioned earlier, calling the method `rollback` terminates a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get a `SQLException`, call the method `rollback` to end the transaction and start the transaction all over again. That is the only way to know what has been committed and what has not been committed. Catching a `SQLException` tells you that something is wrong, but it does not tell you what was or was not committed. Because you cannot count on the fact that nothing was committed, calling the method `rollback` is the only way to be certain.
The method `CoffeesTable.updateCoffeeSales` demonstrates a transaction and includes a `catch` block that invokes the method `rollback`. If the application continues and uses the results of the transaction, this call to the `rollback` method in the `catch` block prevents the use of possibly incorrect data.

事务的提交

事务的提交是在ConnectionImpl的commit方法中，在 个方法中他最终 用了 NativeSession对 的execSQL方法，且执 的 句是commit； 我们如何确定事务是如何开启的？（1）确 ConnectionImpl中是否存在开启事务的方法 （2）使用断点拦截查看 NativeSession是否会执 start transaction 句