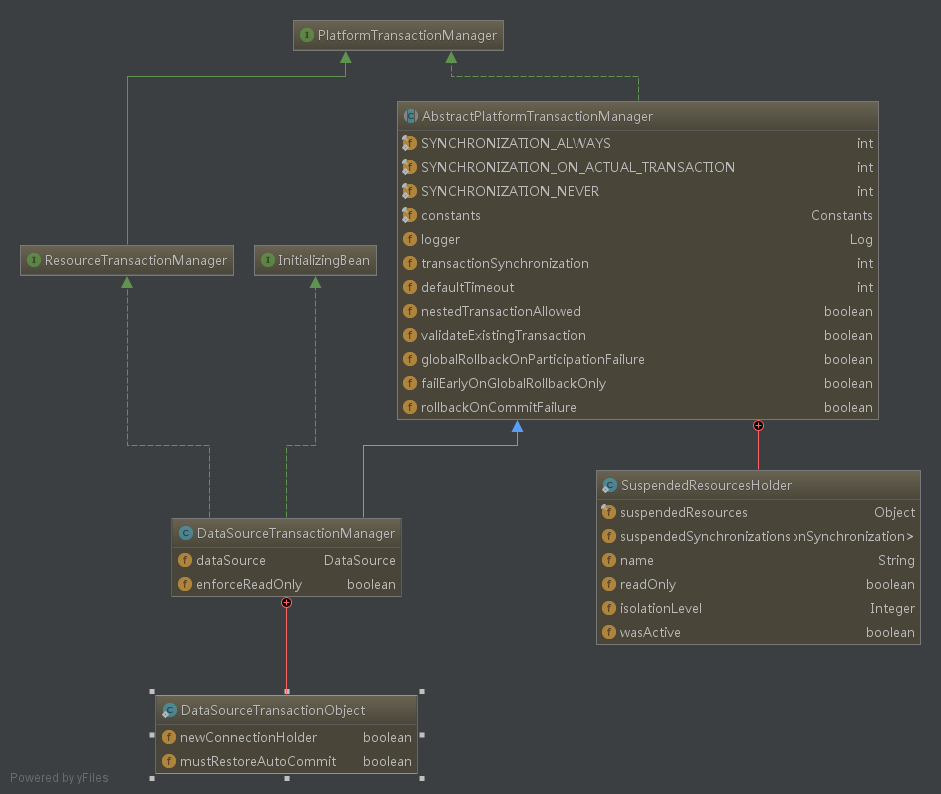
**Spring事务之RuleBasedTransactionAttribute**

**类图DataSourceTransactionManager**



**1. 接口PlatformTransactionManager**

该接口是整个Spring事务基础服务的核心的接口。

*/\*\*  
 \* 根据传播行为，返回一个已经存在的事务或者创建一个新的事务。  
 \*/*TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  
  
*/\*\*  
 \* 根据事务的状态提交事务.   
 \* 如果事务被程序标记为 仅回滚状态，执行回滚。  
 \* 如果当前的事务不是新建的事务，忽略本次提交。  
 \* 如果之前的事务被挂起然后又创建了一个新的事务，提交新创建的事务然后重新开始之前的事务。  
 \*  
 \* 如果事务提交的操作完成，不管正常提交还是抛出了异常，事务将会完全的完成并且清理掉。这种异常的情况，不会产生任何回滚。  
 \*   
 \* 如果该方法抛出异常且不是TransactionException,那么一些之前提交的错误会导致这次提交失败。  
 \*/*void commit(TransactionStatus status) throws TransactionException;  
  
*/\*\*  
 \* 回滚当前的事务。  
 \*/*void rollback(TransactionStatus status) throws TransactionException;

**2. 抽象类TransactionAspectSupport**

该类是一个抽象基类，实现了Spring标准事务工作流，具体的事务管理器需要实现该类，比如 DataSourceTransactionManager和JtaTransactionManager。

该类提供了以下工作流处理：

# 判断是否已经存在事务。

# 使用合适的事务传播行为。

# 挂起或者重新继续一个事务。

# 在提交的时候检查是否是一个 仅提交的事务。

# 回滚的时候，使用合适的修改。

# 触发已注册的同步机制的提交。

*/\*\*  
 \* 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;  
  
  
*/\*\* Constants instance for AbstractPlatformTransactionManager \*/*private static final Constants *constants* = new Constants(AbstractPlatformTransactionManager.class);  
  
//线程绑定事务同步  
private int transactionSynchronization = *SYNCHRONIZATION\_ALWAYS*;  
  
private int defaultTimeout = TransactionDefinition.*TIMEOUT\_DEFAULT*;  
  
private boolean nestedTransactionAllowed = false;  
  
private boolean validateExistingTransaction = false;  
  
private boolean globalRollbackOnParticipationFailure = true;  
  
private boolean failEarlyOnGlobalRollbackOnly = false;  
  
private boolean rollbackOnCommitFailure = false;  
  
public final void setTransactionSynchronizationName(String constantName) {  
 setTransactionSynchronization(*constants*.asNumber(constantName).intValue());  
}public final void setTransactionSynchronization(int transactionSynchronization) {  
 this.transactionSynchronization = transactionSynchronization;  
}public final int getTransactionSynchronization() {  
 return this.transactionSynchronization;  
}  
  
*/\*\*  
 \* 设置事务超时时间。  
 \*/*public final void setDefaultTimeout(int defaultTimeout) {  
 if (defaultTimeout < TransactionDefinition.*TIMEOUT\_DEFAULT*) {  
 throw new InvalidTimeoutException("Invalid default timeout", defaultTimeout);  
 }  
 this.defaultTimeout = defaultTimeout;  
}public final int getDefaultTimeout() {  
 return this.defaultTimeout;  
}  
  
*/\*\*  
 \* 设置是否允许嵌套事务，默认是false。  
 \*/*public final void setNestedTransactionAllowed(boolean nestedTransactionAllowed) {  
 this.nestedTransactionAllowed = nestedTransactionAllowed;  
}public final boolean isNestedTransactionAllowed() {  
 return this.nestedTransactionAllowed;  
}  
  
*/\*\*  
 \* 在加入到当前事务之前，可以设置当前事务是否有效。  
 \*/*public final void setValidateExistingTransaction(boolean validateExistingTransaction) {  
 this.validateExistingTransaction = validateExistingTransaction;  
}public final boolean isValidateExistingTransaction() {  
 return this.validateExistingTransaction;  
}  
  
*/\*\*  
 \* 加入到一个事务失败之后，是否全局的标记当前事务是 仅回滚状态。  
 \* <p>Default is "true": If a participating transaction (e.g. with  
 \* PROPAGATION\_REQUIRES or PROPAGATION\_SUPPORTS encountering an existing  
 \* transaction) fails, the transaction will be globally marked as rollback-only.  
 \* The only possible outcome of such a transaction is a rollback: The  
 \* transaction originator <i>cannot</i> make the transaction commit anymore.  
 \* <p>Switch this to "false" to let the transaction originator make the rollback  
 \* decision. If a participating transaction fails with an exception, the caller  
 \* can still decide to continue with a different path within the transaction.  
 \* However, note that this will only work as long as all participating resources  
 \* are capable of continuing towards a transaction commit even after a data access  
 \* failure: This is generally not the case for a Hibernate Session, for example;  
 \* neither is it for a sequence of JDBC insert/update/delete operations.  
 \* <p><b>Note:</b>This flag only applies to an explicit rollback attempt for a  
 \* subtransaction, typically caused by an exception thrown by a data access operation  
 \* (where TransactionInterceptor will trigger a {****@code*** *PlatformTransactionManager.rollback()}  
 \* call according to a rollback rule). If the flag is off, the caller can handle the exception  
 \* and decide on a rollback, independent of the rollback rules of the subtransaction.  
 \* This flag does, however, <i>not</i> apply to explicit {****@code*** *setRollbackOnly}  
 \* calls on a {****@code*** *TransactionStatus}, which will always cause an eventual  
 \* global rollback (as it might not throw an exception after the rollback-only call).  
 \* <p>The recommended solution for handling failure of a subtransaction  
 \* is a "nested transaction", where the global transaction can be rolled  
 \* back to a savepoint taken at the beginning of the subtransaction.  
 \* PROPAGATION\_NESTED provides exactly those semantics; however, it will  
 \* only work when nested transaction support is available. This is the case  
 \* with DataSourceTransactionManager, but not with JtaTransactionManager.  
 \** ***@see*** *#setNestedTransactionAllowed  
 \** ***@see*** *org.springframework.transaction.jta.JtaTransactionManager  
 \*/*public final void setGlobalRollbackOnParticipationFailure(boolean globalRollbackOnParticipationFailure) {  
 this.globalRollbackOnParticipationFailure = globalRollbackOnParticipationFailure;  
}  
  
*/\*\*  
 \* Return whether to globally mark an existing transaction as rollback-only  
 \* after a participating transaction failed.  
 \*/*public final boolean isGlobalRollbackOnParticipationFailure() {  
 return this.globalRollbackOnParticipationFailure;  
}  
  
*/\*\*  
 \* Set whether to fail early in case of the transaction being globally marked  
 \* as rollback-only.  
 \* <p>Default is "false", only causing an UnexpectedRollbackException at the  
 \* outermost transaction boundary. Switch this flag on to cause an  
 \* UnexpectedRollbackException as early as the global rollback-only marker  
 \* has been first detected, even from within an inner transaction boundary.  
 \* <p>Note that, as of Spring 2.0, the fail-early behavior for global  
 \* rollback-only markers has been unified: All transaction managers will by  
 \* default only cause UnexpectedRollbackException at the outermost transaction  
 \* boundary. This allows, for example, to continue unit tests even after an  
 \* operation failed and the transaction will never be completed. All transaction  
 \* managers will only fail earlier if this flag has explicitly been set to "true".  
 \** ***@see*** *org.springframework.transaction.UnexpectedRollbackException  
 \*/*public final void setFailEarlyOnGlobalRollbackOnly(boolean failEarlyOnGlobalRollbackOnly) {  
 this.failEarlyOnGlobalRollbackOnly = failEarlyOnGlobalRollbackOnly;  
}  
  
*/\*\*  
 \* Return whether to fail early in case of the transaction being globally marked  
 \* as rollback-only.  
 \*/*public final boolean isFailEarlyOnGlobalRollbackOnly() {  
 return this.failEarlyOnGlobalRollbackOnly;  
}  
  
*/\*\*  
 \* 方法doCommit()执行失败，是否需要执行方法doRollback().  
 \*/*public final void setRollbackOnCommitFailure(boolean rollbackOnCommitFailure) {  
 this.rollbackOnCommitFailure = rollbackOnCommitFailure;  
}public final boolean isRollbackOnCommitFailure() {  
 return this.rollbackOnCommitFailure;  
}  
  
  
//---------------------------------------------------------------------  
// Implementation of PlatformTransactionManager  
//---------------------------------------------------------------------  
  
*/\*\*  
 \* 处理传播行为，返回事务状态。  
 \*/*@Override  
public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {  
 Object transaction = doGetTransaction();  
  
 // Cache debug flag to avoid repeated checks.  
 boolean debugEnabled = logger.isDebugEnabled();  
  
 if (definition == null) {  
 // Use defaults if no transaction definition given.  
 definition = new DefaultTransactionDefinition();  
 }

//判断当前否存在事务。  
 if (isExistingTransaction(transaction)) {  
 // Existing transaction found -> check propagation behavior to find out how to behave.  
 //当前存在事务

return handleExistingTransaction(definition, transaction, debugEnabled);  
 }

//当前不存在事务  
 // Check definition settings for new transaction.  
 if (definition.getTimeout() < TransactionDefinition.*TIMEOUT\_DEFAULT*) {  
 throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());  
 }

//如果当前不存在事务就抛出异常，存在事务就加入事务。 这里当前是不存在事务的，所有就抛出异常。  
 // No existing transaction found -> check propagation behavior to find out how to proceed.  
 if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_MANDATORY*) {  
 throw new IllegalTransactionStateException(  
 "No existing transaction found for transaction marked with propagation 'mandatory'");  
 }else if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_REQUIRED* ||  
 definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_REQUIRES\_NEW* ||  
 definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NESTED*) {  
 SuspendedResourcesHolder suspendedResources = suspend(null);  
 if (debugEnabled) {  
 logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);  
 }  
 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 ex) {  
 resume(null, suspendedResources);  
 throw ex;  
 }  
 }else {  
 // Create "empty" transaction: no actual transaction, but potentially synchronization.  
 if (definition.getIsolationLevel() != TransactionDefinition.*ISOLATION\_DEFAULT* && logger.isWarnEnabled()) {  
 logger.warn("Custom isolation level specified but no actual transaction initiated; " +  
 "isolation level will effectively be ignored: " + definition);  
 }  
 boolean newSynchronization = (getTransactionSynchronization() == *SYNCHRONIZATION\_ALWAYS*);  
 return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);  
 }  
}  
  
*/\*\*  
 \* 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) {  
 resumeAfterBeginException(transaction, suspendedResources, beginEx);  
 throw beginEx;  
 }  
 }  
  
 if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NESTED*) {  
 if (!isNestedTransactionAllowed()) {  
 throw new NestedTransactionNotSupportedException(  
 "Transaction manager does not allow nested transactions by default - " +  
 "specify 'nestedTransactionAllowed' property with value 'true'");  
 }  
 if (debugEnabled) {  
 logger.debug("Creating nested transaction with name [" + definition.getName() + "]");  
 }  
 if (useSavepointForNestedTransaction()) {  
 // Create savepoint within existing Spring-managed transaction,  
 // through the SavepointManager API implemented by TransactionStatus.  
 // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.  
 DefaultTransactionStatus status =  
 prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);  
 status.createAndHoldSavepoint();  
 return status;  
 }  
 else {  
 // Nested transaction through nested begin and commit/rollback calls.  
 // Usually only for JTA: Spring synchronization might get activated here  
 // in case of a pre-existing JTA transaction.  
 boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  
 DefaultTransactionStatus status = newTransactionStatus(  
 definition, transaction, true, newSynchronization, debugEnabled, null);  
 doBegin(transaction, definition);  
 prepareSynchronization(status, definition);  
 return status;  
 }  
 }  
  
 // Assumably PROPAGATION\_SUPPORTS or PROPAGATION\_REQUIRED.  
 if (debugEnabled) {  
 logger.debug("Participating in existing transaction");  
 }  
 if (isValidateExistingTransaction()) {  
 if (definition.getIsolationLevel() != TransactionDefinition.*ISOLATION\_DEFAULT*) {  
 Integer currentIsolationLevel = TransactionSynchronizationManager.*getCurrentTransactionIsolationLevel*();  
 if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {  
 Constants isoConstants = DefaultTransactionDefinition.*constants*;  
 throw new IllegalTransactionStateException("Participating transaction with definition [" +  
 definition + "] specifies isolation level which is incompatible with existing transaction: " +  
 (currentIsolationLevel != null ?  
 isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.*PREFIX\_ISOLATION*) :  
 "(unknown)"));  
 }  
 }  
 if (!definition.isReadOnly()) {  
 if (TransactionSynchronizationManager.*isCurrentTransactionReadOnly*()) {  
 throw new IllegalTransactionStateException("Participating transaction with definition [" +  
 definition + "] is not marked as read-only but existing transaction is");  
 }  
 }  
 }  
 boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  
 return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null);  
}  
  
*/\*\*  
 \* Create a new TransactionStatus for the given arguments,  
 \* also initializing transaction synchronization as appropriate.  
 \** ***@see*** *#newTransactionStatus  
 \** ***@see*** *#prepareTransactionStatus  
 \*/*protected final DefaultTransactionStatus prepareTransactionStatus(  
 TransactionDefinition definition, Object transaction, boolean newTransaction,  
 boolean newSynchronization, boolean debug, Object suspendedResources) {  
  
 DefaultTransactionStatus status = newTransactionStatus(  
 definition, transaction, newTransaction, newSynchronization, debug, suspendedResources);  
 prepareSynchronization(status, definition);  
 return status;  
}  
  
*/\*\*  
 \* Create a TransactionStatus instance for the given arguments.  
 \*/*protected DefaultTransactionStatus newTransactionStatus(  
 TransactionDefinition definition, Object transaction, boolean newTransaction,  
 boolean newSynchronization, boolean debug, Object suspendedResources) {  
  
 boolean actualNewSynchronization = newSynchronization &&  
 !TransactionSynchronizationManager.*isSynchronizationActive*();  
 return new DefaultTransactionStatus(  
 transaction, newTransaction, actualNewSynchronization,  
 definition.isReadOnly(), debug, suspendedResources);  
}  
  
*/\*\*  
 \* 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*();  
 }  
}  
  
*/\*\*  
 \* Determine the actual timeout to use for the given definition.  
 \* Will fall back to this manager's default timeout if the  
 \* transaction definition doesn't specify a non-default value.  
 \** ***@param*** *definition the transaction definition  
 \** ***@return*** *the actual timeout to use  
 \** ***@see*** *org.springframework.transaction.TransactionDefinition#getTimeout()  
 \** ***@see*** *#setDefaultTimeout  
 \*/*protected int determineTimeout(TransactionDefinition definition) {  
 if (definition.getTimeout() != TransactionDefinition.*TIMEOUT\_DEFAULT*) {  
 return definition.getTimeout();  
 }  
 return this.defaultTimeout;  
}  
  
  
*/\*\*  
 \* Suspend the given transaction. Suspends transaction synchronization first,  
 \* then delegates to the {****@code*** *doSuspend} template method.  
 \** ***@param*** *transaction the current transaction object  
 \* (or {****@code*** *null} to just suspend active synchronizations, if any)  
 \** ***@return*** *an object that holds suspended resources  
 \* (or {****@code*** *null} if neither transaction nor synchronization active)  
 \** ***@see*** *#doSuspend  
 \** ***@see*** *#resume  
 \*/*protected final SuspendedResourcesHolder suspend(Object transaction) throws TransactionException {  
 if (TransactionSynchronizationManager.*isSynchronizationActive*()) {  
 List<TransactionSynchronization> suspendedSynchronizations = doSuspendSynchronization();  
 try {  
 Object suspendedResources = null;  
 if (transaction != null) {  
 suspendedResources = doSuspend(transaction);  
 }  
 String name = TransactionSynchronizationManager.*getCurrentTransactionName*();  
 TransactionSynchronizationManager.*setCurrentTransactionName*(null);  
 boolean readOnly = TransactionSynchronizationManager.*isCurrentTransactionReadOnly*();  
 TransactionSynchronizationManager.*setCurrentTransactionReadOnly*(false);  
 Integer isolationLevel = TransactionSynchronizationManager.*getCurrentTransactionIsolationLevel*();  
 TransactionSynchronizationManager.*setCurrentTransactionIsolationLevel*(null);  
 boolean wasActive = TransactionSynchronizationManager.*isActualTransactionActive*();  
 TransactionSynchronizationManager.*setActualTransactionActive*(false);  
 return new SuspendedResourcesHolder(  
 suspendedResources, suspendedSynchronizations, name, readOnly, isolationLevel, wasActive);  
 }  
 catch (RuntimeException | Error ex) {  
 // doSuspend failed - original transaction is still active...  
 doResumeSynchronization(suspendedSynchronizations);  
 throw ex;  
 }  
 }  
 else if (transaction != null) {  
 // Transaction active but no synchronization active.  
 Object suspendedResources = doSuspend(transaction);  
 return new SuspendedResourcesHolder(suspendedResources);  
 }  
 else {  
 // Neither transaction nor synchronization active.  
 return null;  
 }  
}  
  
*/\*\*  
 \* Resume the given transaction. Delegates to the {****@code*** *doResume}  
 \* template method first, then resuming transaction synchronization.  
 \** ***@param*** *transaction the current transaction object  
 \** ***@param*** *resourcesHolder the object that holds suspended resources,  
 \* as returned by {****@code*** *suspend} (or {****@code*** *null} to just  
 \* resume synchronizations, if any)  
 \** ***@see*** *#doResume  
 \** ***@see*** *#suspend  
 \*/*protected final void resume(Object transaction, SuspendedResourcesHolder resourcesHolder)  
 throws TransactionException {  
  
 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);  
 }  
 }  
}  
  
*/\*\*  
 \* Resume outer transaction after inner transaction begin failed.  
 \*/*private void resumeAfterBeginException(  
 Object transaction, SuspendedResourcesHolder suspendedResources, Throwable beginEx) {  
  
 String exMessage = "Inner transaction begin exception overridden by outer transaction resume exception";  
 try {  
 resume(transaction, suspendedResources);  
 }  
 catch (RuntimeException | Error resumeEx) {  
 logger.error(exMessage, beginEx);  
 throw resumeEx;  
 }  
}  
  
*/\*\*  
 \* Suspend all current synchronizations and deactivate transaction  
 \* synchronization for the current thread.  
 \** ***@return*** *the List of suspended TransactionSynchronization objects  
 \*/*private List<TransactionSynchronization> doSuspendSynchronization() {  
 List<TransactionSynchronization> suspendedSynchronizations =  
 TransactionSynchronizationManager.*getSynchronizations*();  
 for (TransactionSynchronization synchronization : suspendedSynchronizations) {  
 synchronization.suspend();  
 }  
 TransactionSynchronizationManager.*clearSynchronization*();  
 return suspendedSynchronizations;  
}  
  
*/\*\*  
 \* Reactivate transaction synchronization for the current thread  
 \* and resume all given synchronizations.  
 \** ***@param*** *suspendedSynchronizations List of TransactionSynchronization objects  
 \*/*private void doResumeSynchronization(List<TransactionSynchronization> suspendedSynchronizations) {  
 TransactionSynchronizationManager.*initSynchronization*();  
 for (TransactionSynchronization synchronization : suspendedSynchronizations) {  
 synchronization.resume();  
 TransactionSynchronizationManager.*registerSynchronization*(synchronization);  
 }  
}  
  
  
*/\*\*  
 \* This implementation of commit handles participating in existing  
 \* transactions and programmatic rollback requests.  
 \* Delegates to {****@code*** *isRollbackOnly}, {****@code*** *doCommit}  
 \* and {****@code*** *rollback}.  
 \** ***@see*** *org.springframework.transaction.TransactionStatus#isRollbackOnly()  
 \** ***@see*** *#doCommit  
 \** ***@see*** *#rollback  
 \*/*@Override  
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");  
 }  
  
 DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;  
 if (defStatus.isLocalRollbackOnly()) {  
 if (defStatus.isDebug()) {  
 logger.debug("Transactional code has requested rollback");  
 }  
 processRollback(defStatus, false);  
 return;  
 }  
  
 if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {  
 if (defStatus.isDebug()) {  
 logger.debug("Global transaction is marked as rollback-only but transactional code requested commit");  
 }  
 processRollback(defStatus, true);  
 return;  
 }  
  
 processCommit(defStatus);  
}  
  
*/\*\*  
 \* Process an actual commit.  
 \* Rollback-only flags have already been checked and applied.  
 \** ***@param*** *status object representing the transaction  
 \** ***@throws*** *TransactionException in case of commit failure  
 \*/*private void processCommit(DefaultTransactionStatus status) throws TransactionException {  
 try {  
 boolean beforeCompletionInvoked = false;  
  
 try {  
 boolean unexpectedRollback = false;  
 prepareForCommit(status);  
 triggerBeforeCommit(status);  
 triggerBeforeCompletion(status);  
 beforeCompletionInvoked = true;  
  
 if (status.hasSavepoint()) {  
 if (status.isDebug()) {  
 logger.debug("Releasing transaction savepoint");  
 }  
 unexpectedRollback = status.isGlobalRollbackOnly();  
 status.releaseHeldSavepoint();  
 }  
 else if (status.isNewTransaction()) {  
 if (status.isDebug()) {  
 logger.debug("Initiating transaction commit");  
 }  
 unexpectedRollback = status.isGlobalRollbackOnly();  
 doCommit(status);  
 }  
 else if (isFailEarlyOnGlobalRollbackOnly()) {  
 unexpectedRollback = status.isGlobalRollbackOnly();  
 }  
  
 // Throw UnexpectedRollbackException if we have a global rollback-only  
 // marker but still didn't get a corresponding exception from commit.  
 if (unexpectedRollback) {  
 throw new UnexpectedRollbackException(  
 "Transaction silently rolled back because it has been marked as rollback-only");  
 }  
 }  
 catch (UnexpectedRollbackException ex) {  
 // can only be caused by doCommit  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_ROLLED\_BACK*);  
 throw ex;  
 }  
 catch (TransactionException ex) {  
 // can only be caused by doCommit  
 if (isRollbackOnCommitFailure()) {  
 doRollbackOnCommitException(status, ex);  
 }  
 else {  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_UNKNOWN*);  
 }  
 throw ex;  
 }  
 catch (RuntimeException | Error ex) {  
 if (!beforeCompletionInvoked) {  
 triggerBeforeCompletion(status);  
 }  
 doRollbackOnCommitException(status, ex);  
 throw ex;  
 }  
  
 // Trigger afterCommit callbacks, with an exception thrown there  
 // propagated to callers but the transaction still considered as committed.  
 try {  
 triggerAfterCommit(status);  
 }  
 finally {  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_COMMITTED*);  
 }  
  
 }  
 finally {  
 cleanupAfterCompletion(status);  
 }  
}  
  
*/\*\*  
 \* This implementation of rollback handles participating in existing  
 \* transactions. Delegates to {****@code*** *doRollback} and  
 \* {****@code*** *doSetRollbackOnly}.  
 \** ***@see*** *#doRollback  
 \** ***@see*** *#doSetRollbackOnly  
 \*/*@Override  
public final void rollback(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");  
 }  
  
 DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;  
 processRollback(defStatus, false);  
}  
  
*/\*\*  
 \* Process an actual rollback.  
 \* The completed flag has already been checked.  
 \** ***@param*** *status object representing the transaction  
 \** ***@throws*** *TransactionException in case of rollback failure  
 \*/*private void processRollback(DefaultTransactionStatus status, boolean unexpected) {  
 try {  
 boolean unexpectedRollback = unexpected;  
  
 try {  
 triggerBeforeCompletion(status);  
  
 if (status.hasSavepoint()) {  
 if (status.isDebug()) {  
 logger.debug("Rolling back transaction to savepoint");  
 }  
 status.rollbackToHeldSavepoint();  
 }  
 else if (status.isNewTransaction()) {  
 if (status.isDebug()) {  
 logger.debug("Initiating transaction rollback");  
 }  
 doRollback(status);  
 }  
 else {  
 // Participating in larger transaction  
 if (status.hasTransaction()) {  
 if (status.isLocalRollbackOnly() || isGlobalRollbackOnParticipationFailure()) {  
 if (status.isDebug()) {  
 logger.debug("Participating transaction failed - marking existing transaction as rollback-only");  
 }  
 doSetRollbackOnly(status);  
 }  
 else {  
 if (status.isDebug()) {  
 logger.debug("Participating transaction failed - letting transaction originator decide on rollback");  
 }  
 }  
 }  
 else {  
 logger.debug("Should roll back transaction but cannot - no transaction available");  
 }  
 // Unexpected rollback only matters here if we're asked to fail early  
 if (!isFailEarlyOnGlobalRollbackOnly()) {  
 unexpectedRollback = false;  
 }  
 }  
 }  
 catch (RuntimeException | Error ex) {  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_UNKNOWN*);  
 throw ex;  
 }  
  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_ROLLED\_BACK*);  
  
 // Raise UnexpectedRollbackException if we had a global rollback-only marker  
 if (unexpectedRollback) {  
 throw new UnexpectedRollbackException(  
 "Transaction rolled back because it has been marked as rollback-only");  
 }  
 }  
 finally {  
 cleanupAfterCompletion(status);  
 }  
}  
  
*/\*\*  
 \* Invoke {****@code*** *doRollback}, handling rollback exceptions properly.  
 \** ***@param*** *status object representing the transaction  
 \** ***@param*** *ex the thrown application exception or error  
 \** ***@throws*** *TransactionException in case of rollback failure  
 \** ***@see*** *#doRollback  
 \*/*private void doRollbackOnCommitException(DefaultTransactionStatus status, Throwable ex) throws TransactionException {  
 try {  
 if (status.isNewTransaction()) {  
 if (status.isDebug()) {  
 logger.debug("Initiating transaction rollback after commit exception", ex);  
 }  
 doRollback(status);  
 }  
 else if (status.hasTransaction() && isGlobalRollbackOnParticipationFailure()) {  
 if (status.isDebug()) {  
 logger.debug("Marking existing transaction as rollback-only after commit exception", ex);  
 }  
 doSetRollbackOnly(status);  
 }  
 }  
 catch (RuntimeException | Error rbex) {  
 logger.error("Commit exception overridden by rollback exception", ex);  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_UNKNOWN*);  
 throw rbex;  
 }  
 triggerAfterCompletion(status, TransactionSynchronization.*STATUS\_ROLLED\_BACK*);  
}  
  
  
*/\*\*  
 \* Trigger {****@code*** *beforeCommit} callbacks.  
 \** ***@param*** *status object representing the transaction  
 \*/*protected final void triggerBeforeCommit(DefaultTransactionStatus status) {  
 if (status.isNewSynchronization()) {  
 if (status.isDebug()) {  
 logger.trace("Triggering beforeCommit synchronization");  
 }  
 TransactionSynchronizationUtils.*triggerBeforeCommit*(status.isReadOnly());  
 }  
}  
  
*/\*\*  
 \* Trigger {****@code*** *beforeCompletion} callbacks.  
 \** ***@param*** *status object representing the transaction  
 \*/*protected final void triggerBeforeCompletion(DefaultTransactionStatus status) {  
 if (status.isNewSynchronization()) {  
 if (status.isDebug()) {  
 logger.trace("Triggering beforeCompletion synchronization");  
 }  
 TransactionSynchronizationUtils.*triggerBeforeCompletion*();  
 }  
}  
  
*/\*\*  
 \* Trigger {****@code*** *afterCommit} callbacks.  
 \** ***@param*** *status object representing the transaction  
 \*/*private void triggerAfterCommit(DefaultTransactionStatus status) {  
 if (status.isNewSynchronization()) {  
 if (status.isDebug()) {  
 logger.trace("Triggering afterCommit synchronization");  
 }  
 TransactionSynchronizationUtils.*triggerAfterCommit*();  
 }  
}  
  
*/\*\*  
 \* Trigger {****@code*** *afterCompletion} callbacks.  
 \** ***@param*** *status object representing the transaction  
 \** ***@param*** *completionStatus completion status according to TransactionSynchronization constants  
 \*/*private void triggerAfterCompletion(DefaultTransactionStatus status, int completionStatus) {  
 if (status.isNewSynchronization()) {  
 List<TransactionSynchronization> synchronizations = TransactionSynchronizationManager.*getSynchronizations*();  
 TransactionSynchronizationManager.*clearSynchronization*();  
 if (!status.hasTransaction() || status.isNewTransaction()) {  
 if (status.isDebug()) {  
 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);  
 }  
 }  
}  
  
*/\*\*  
 \* Actually invoke the {****@code*** *afterCompletion} methods of the  
 \* given Spring TransactionSynchronization objects.  
 \* <p>To be called by this abstract manager itself, or by special implementations  
 \* of the {****@code*** *registerAfterCompletionWithExistingTransaction} callback.  
 \** ***@param*** *synchronizations List of TransactionSynchronization objects  
 \** ***@param*** *completionStatus the completion status according to the  
 \* constants in the TransactionSynchronization interface  
 \** ***@see*** *#registerAfterCompletionWithExistingTransaction(Object, java.util.List)  
 \** ***@see*** *TransactionSynchronization#STATUS\_COMMITTED  
 \** ***@see*** *TransactionSynchronization#STATUS\_ROLLED\_BACK  
 \** ***@see*** *TransactionSynchronization#STATUS\_UNKNOWN  
 \*/*protected final void invokeAfterCompletion(List<TransactionSynchronization> synchronizations, int completionStatus) {  
 TransactionSynchronizationUtils.*invokeAfterCompletion*(synchronizations, completionStatus);  
}  
  
*/\*\*  
 \* Clean up after completion, clearing synchronization if necessary,  
 \* and invoking doCleanupAfterCompletion.  
 \** ***@param*** *status object representing the transaction  
 \** ***@see*** *#doCleanupAfterCompletion  
 \*/*private void cleanupAfterCompletion(DefaultTransactionStatus status) {  
 status.setCompleted();  
 if (status.isNewSynchronization()) {  
 TransactionSynchronizationManager.*clear*();  
 }  
 if (status.isNewTransaction()) {  
 doCleanupAfterCompletion(status.getTransaction());  
 }  
 if (status.getSuspendedResources() != null) {  
 if (status.isDebug()) {  
 logger.debug("Resuming suspended transaction after completion of inner transaction");  
 }  
 resume(status.getTransaction(), (SuspendedResourcesHolder) status.getSuspendedResources());  
 }  
}  
  
  
//---------------------------------------------------------------------  
// Template methods to be implemented in subclasses  
//---------------------------------------------------------------------  
  
*/\*\*  
 \* Return a transaction object for the current transaction state.  
 \* <p>The returned object will usually be specific to the concrete transaction  
 \* manager implementation, carrying corresponding transaction state in a  
 \* modifiable fashion. This object will be passed into the other template  
 \* methods (e.g. doBegin and doCommit), either directly or as part of a  
 \* DefaultTransactionStatus instance.  
 \* <p>The returned object should contain information about any existing  
 \* transaction, that is, a transaction that has already started before the  
 \* current {****@code*** *getTransaction} call on the transaction manager.  
 \* Consequently, a {****@code*** *doGetTransaction} implementation will usually  
 \* look for an existing transaction and store corresponding state in the  
 \* returned transaction object.  
 \** ***@return*** *the current transaction object  
 \** ***@throws*** *org.springframework.transaction.CannotCreateTransactionException  
 \* if transaction support is not available  
 \** ***@throws*** *TransactionException in case of lookup or system errors  
 \** ***@see*** *#doBegin  
 \** ***@see*** *#doCommit  
 \** ***@see*** *#doRollback  
 \** ***@see*** *DefaultTransactionStatus#getTransaction  
 \*/*protected abstract Object doGetTransaction() throws TransactionException;  
  
*/\*\*  
 \* Check if the given transaction object indicates an existing transaction  
 \* (that is, a transaction which has already started).  
 \* <p>The result will be evaluated according to the specified propagation  
 \* behavior for the new transaction. An existing transaction might get  
 \* suspended (in case of PROPAGATION\_REQUIRES\_NEW), or the new transaction  
 \* might participate in the existing one (in case of PROPAGATION\_REQUIRED).  
 \* <p>The default implementation returns {****@code*** *false}, assuming that  
 \* participating in existing transactions is generally not supported.  
 \* Subclasses are of course encouraged to provide such support.  
 \** ***@param*** *transaction transaction object returned by doGetTransaction  
 \** ***@return*** *if there is an existing transaction  
 \** ***@throws*** *TransactionException in case of system errors  
 \** ***@see*** *#doGetTransaction  
 \*/*protected boolean isExistingTransaction(Object transaction) throws TransactionException {  
 return false;  
}  
  
*/\*\*  
 \* Return whether to use a savepoint for a nested transaction.  
 \* <p>Default is {****@code*** *true}, which causes delegation to DefaultTransactionStatus  
 \* for creating and holding a savepoint. If the transaction object does not implement  
 \* the SavepointManager interface, a NestedTransactionNotSupportedException will be  
 \* thrown. Else, the SavepointManager will be asked to create a new savepoint to  
 \* demarcate the start of the nested transaction.  
 \* <p>Subclasses can override this to return {****@code*** *false}, causing a further  
 \* call to {****@code*** *doBegin} - within the context of an already existing transaction.  
 \* The {****@code*** *doBegin} implementation needs to handle this accordingly in such  
 \* a scenario. This is appropriate for JTA, for example.  
 \** ***@see*** *DefaultTransactionStatus#createAndHoldSavepoint  
 \** ***@see*** *DefaultTransactionStatus#rollbackToHeldSavepoint  
 \** ***@see*** *DefaultTransactionStatus#releaseHeldSavepoint  
 \** ***@see*** *#doBegin  
 \*/*protected boolean useSavepointForNestedTransaction() {  
 return true;  
}  
  
*/\*\*  
 \* 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;  
  
*/\*\*  
 \* Suspend the resources of the current transaction.  
 \* Transaction synchronization will already have been suspended.  
 \* <p>The default implementation throws a TransactionSuspensionNotSupportedException,  
 \* assuming that transaction suspension is generally not supported.  
 \** ***@param*** *transaction transaction object returned by {****@code*** *doGetTransaction}  
 \** ***@return*** *an object that holds suspended resources  
 \* (will be kept unexamined for passing it into doResume)  
 \** ***@throws*** *org.springframework.transaction.TransactionSuspensionNotSupportedException  
 \* if suspending is not supported by the transaction manager implementation  
 \** ***@throws*** *TransactionException in case of system errors  
 \** ***@see*** *#doResume  
 \*/*protected Object doSuspend(Object transaction) throws TransactionException {  
 throw new TransactionSuspensionNotSupportedException(  
 "Transaction manager [" + getClass().getName() + "] does not support transaction suspension");  
}  
  
*/\*\*  
 \* Resume the resources of the current transaction.  
 \* Transaction synchronization will be resumed afterwards.  
 \* <p>The default implementation throws a TransactionSuspensionNotSupportedException,  
 \* assuming that transaction suspension is generally not supported.  
 \** ***@param*** *transaction transaction object returned by {****@code*** *doGetTransaction}  
 \** ***@param*** *suspendedResources the object that holds suspended resources,  
 \* as returned by doSuspend  
 \** ***@throws*** *org.springframework.transaction.TransactionSuspensionNotSupportedException  
 \* if resuming is not supported by the transaction manager implementation  
 \** ***@throws*** *TransactionException in case of system errors  
 \** ***@see*** *#doSuspend  
 \*/*protected void doResume(Object transaction, Object suspendedResources) throws TransactionException {  
 throw new TransactionSuspensionNotSupportedException(  
 "Transaction manager [" + getClass().getName() + "] does not support transaction suspension");  
}  
  
*/\*\*  
 \* Return whether to call {****@code*** *doCommit} on a transaction that has been  
 \* marked as rollback-only in a global fashion.  
 \* <p>Does not apply if an application locally sets the transaction to rollback-only  
 \* via the TransactionStatus, but only to the transaction itself being marked as  
 \* rollback-only by the transaction coordinator.  
 \* <p>Default is "false": Local transaction strategies usually don't hold the rollback-only  
 \* marker in the transaction itself, therefore they can't handle rollback-only transactions  
 \* as part of transaction commit. Hence, AbstractPlatformTransactionManager will trigger  
 \* a rollback in that case, throwing an UnexpectedRollbackException afterwards.  
 \* <p>Override this to return "true" if the concrete transaction manager expects a  
 \* {****@code*** *doCommit} call even for a rollback-only transaction, allowing for  
 \* special handling there. This will, for example, be the case for JTA, where  
 \* {****@code*** *UserTransaction.commit} will check the read-only flag itself and  
 \* throw a corresponding RollbackException, which might include the specific reason  
 \* (such as a transaction timeout).  
 \* <p>If this method returns "true" but the {****@code*** *doCommit} implementation does not  
 \* throw an exception, this transaction manager will throw an UnexpectedRollbackException  
 \* itself. This should not be the typical case; it is mainly checked to cover misbehaving  
 \* JTA providers that silently roll back even when the rollback has not been requested  
 \* by the calling code.  
 \** ***@see*** *#doCommit  
 \** ***@see*** *DefaultTransactionStatus#isGlobalRollbackOnly()  
 \** ***@see*** *DefaultTransactionStatus#isLocalRollbackOnly()  
 \** ***@see*** *org.springframework.transaction.TransactionStatus#setRollbackOnly()  
 \** ***@see*** *org.springframework.transaction.UnexpectedRollbackException  
 \** ***@see*** *javax.transaction.UserTransaction#commit()  
 \** ***@see*** *javax.transaction.RollbackException  
 \*/*protected boolean shouldCommitOnGlobalRollbackOnly() {  
 return false;  
}  
  
*/\*\*  
 \* Make preparations for commit, to be performed before the  
 \* {****@code*** *beforeCommit} synchronization callbacks occur.  
 \* <p>Note that exceptions will get propagated to the commit caller  
 \* and cause a rollback of the transaction.  
 \** ***@param*** *status the status representation of the transaction  
 \** ***@throws*** *RuntimeException in case of errors; will be <b>propagated to the caller</b>  
 \* (note: do not throw TransactionException subclasses here!)  
 \*/*protected void prepareForCommit(DefaultTransactionStatus status) {  
}  
  
*/\*\*  
 \* Perform an actual commit of the given transaction.  
 \* <p>An implementation does not need to check the "new transaction" flag  
 \* or the rollback-only flag; this will already have been handled before.  
 \* Usually, a straight commit will be performed on the transaction object  
 \* contained in the passed-in status.  
 \** ***@param*** *status the status representation of the transaction  
 \** ***@throws*** *TransactionException in case of commit or system errors  
 \** ***@see*** *DefaultTransactionStatus#getTransaction  
 \*/*protected abstract void doCommit(DefaultTransactionStatus status) throws TransactionException;  
  
*/\*\*  
 \* Perform an actual rollback of the given transaction.  
 \* <p>An implementation does not need to check the "new transaction" flag;  
 \* this will already have been handled before. Usually, a straight rollback  
 \* will be performed on the transaction object contained in the passed-in status.  
 \** ***@param*** *status the status representation of the transaction  
 \** ***@throws*** *TransactionException in case of system errors  
 \** ***@see*** *DefaultTransactionStatus#getTransaction  
 \*/*protected abstract void doRollback(DefaultTransactionStatus status) throws TransactionException;  
  
*/\*\*  
 \* Set the given transaction rollback-only. Only called on rollback  
 \* if the current transaction participates in an existing one.  
 \* <p>The default implementation throws an IllegalTransactionStateException,  
 \* assuming that participating in existing transactions is generally not  
 \* supported. Subclasses are of course encouraged to provide such support.  
 \** ***@param*** *status the status representation of the transaction  
 \** ***@throws*** *TransactionException in case of system errors  
 \*/*protected void doSetRollbackOnly(DefaultTransactionStatus status) throws TransactionException {  
 throw new IllegalTransactionStateException(  
 "Participating in existing transactions is not supported - when 'isExistingTransaction' " +  
 "returns true, appropriate 'doSetRollbackOnly' behavior must be provided");  
}  
  
*/\*\*  
 \* Register the given list of transaction synchronizations with the existing transaction.  
 \* <p>Invoked when the control of the Spring transaction manager and thus all Spring  
 \* transaction synchronizations end, without the transaction being completed yet. This  
 \* is for example the case when participating in an existing JTA or EJB CMT transaction.  
 \* <p>The default implementation simply invokes the {****@code*** *afterCompletion} methods  
 \* immediately, passing in "STATUS\_UNKNOWN". This is the best we can do if there's no  
 \* chance to determine the actual outcome of the outer transaction.  
 \** ***@param*** *transaction transaction object returned by {****@code*** *doGetTransaction}  
 \** ***@param*** *synchronizations List of TransactionSynchronization objects  
 \** ***@throws*** *TransactionException in case of system errors  
 \** ***@see*** *#invokeAfterCompletion(java.util.List, int)  
 \** ***@see*** *TransactionSynchronization#afterCompletion(int)  
 \** ***@see*** *TransactionSynchronization#STATUS\_UNKNOWN  
 \*/*protected void registerAfterCompletionWithExistingTransaction(  
 Object transaction, List<TransactionSynchronization> synchronizations) throws TransactionException {  
  
 logger.debug("Cannot register Spring after-completion synchronization with existing transaction - " +  
 "processing Spring after-completion callbacks immediately, with outcome status 'unknown'");  
 invokeAfterCompletion(synchronizations, TransactionSynchronization.*STATUS\_UNKNOWN*);  
}  
  
*/\*\*  
 \* Cleanup resources after transaction completion.  
 \* <p>Called after {****@code*** *doCommit} and {****@code*** *doRollback} execution,  
 \* on any outcome. The default implementation does nothing.  
 \* <p>Should not throw any exceptions but just issue warnings on errors.  
 \** ***@param*** *transaction transaction object returned by {****@code*** *doGetTransaction}  
 \*/*protected void doCleanupAfterCompletion(Object transaction) {  
}  
  
  
//---------------------------------------------------------------------  
// Serialization support  
//---------------------------------------------------------------------  
  
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {  
 // Rely on default serialization; just initialize state after deserialization.  
 ois.defaultReadObject();  
  
 // Initialize transient fields.  
 this.logger = LogFactory.*getLog*(getClass());  
}  
  
  
*/\*\*  
 \* Holder for suspended resources.  
 \* Used internally by {****@code*** *suspend} and {****@code*** *resume}.  
 \*/*protected static class SuspendedResourcesHolder {  
  
 private final Object suspendedResources;  
  
 private List<TransactionSynchronization> suspendedSynchronizations;  
  
 private String name;  
  
 private boolean readOnly;  
  
 private Integer isolationLevel;  
  
 private boolean wasActive;  
  
 private SuspendedResourcesHolder(Object suspendedResources) {  
 this.suspendedResources = suspendedResources;  
 }  
  
 private SuspendedResourcesHolder(  
 Object suspendedResources, List<TransactionSynchronization> suspendedSynchronizations,  
 String name, boolean readOnly, Integer isolationLevel, boolean wasActive) {  
 this.suspendedResources = suspendedResources;  
 this.suspendedSynchronizations = suspendedSynchronizations;  
 this.name = name;  
 this.readOnly = readOnly;  
 this.isolationLevel = isolationLevel;  
 this.wasActive = wasActive;  
 }  
}