8-7. Managing Transactions Declaratively with the @Transactional Annotation

Problem

Declaring transactions in the bean configuration file requires knowledge of AOP concepts such as pointcuts, advices, and advisors. Developers who lack this knowledge might find it hard to enable declarative transaction management.

Solution

In addition to declaring transactions in the bean configuration file with pointcuts, advices, and advisors, Spring allows you to declare transactions simply by annotating your transactional methods with `@Transactional` and enabling the `<tx:annotation-driven>` element. However, Java 1.5 or higher is required to use this approach.

How It Works

To define a method as transactional, you can simply annotate it with `@Transactional`. Note that you should only annotate public methods due to the proxy-based limitations of Spring AOP.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {

    @Transactional
    public void purchase(String isbn, String username) {
        ...
    }
}
```

You may apply the `@Transactional` annotation at the method level or the class level. When applying this annotation to a class, all of the public methods within this class will be defined as transactional. Although you can apply `@Transactional` to interfaces or method declarations in an interface, it's not recommended, as it may not work properly with class-based proxies (i.e., CGLIB proxies).

In the bean configuration file, you only have to enable the `<tx:annotation-driven>` element and specify a transaction manager for it. That's all you need to make it work. Spring will advise methods with `@Transactional`, or methods in a class with `@Transactional`, from beans declared in the IoC container. As a result, Spring can manage transactions for these methods.

```
<beans ...>
    <tx:annotation-driven transaction-manager="transactionManager" />
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.JdbcBookShop">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

In fact, you can omit the `transaction-manager` attribute in the `<tx:annotation-driven>` element if your transaction manager has the name `transactionManager`. This element will automatically detect a transaction manager with this name. You only have to specify a transaction manager when it has a different name.

```
<beans ...>
    <tx:annotation-driven />
    ...
</beans>
```

8-8. Setting the Propagation Transaction Attribute

Problem

When a transactional method is called by another method, it is necessary to specify how the transaction should be propagated. For example, the method may continue to run within the existing transaction, or it may start a new transaction and run within its own transaction.

Solution

A transaction's propagation behavior can be specified by the *propagation* transaction attribute. Spring defines seven propagation behaviors, as shown in Table 8-5. These behaviors are defined in the `org.springframework.transaction.TransactionDefinition` interface. Note that not all types of transaction managers support all of these propagation behaviors.

Table 8-5. *Propagation Behaviors Supported by Spring*

Propagation	Description
`REQUIRED`	If there's an existing transaction in progress, the current method should run within this transaction. Otherwise, it should start a new transaction and run within its own transaction.
`REQUIRES_NEW`	The current method must start a new transaction and run within its own transaction. If there's an existing transaction in progress, it should be suspended.
`SUPPORTS`	If there's an existing transaction in progress, the current method can run within this transaction. Otherwise, it is not necessary to run within a transaction.
`NOT_SUPPORTED`	The current method should not run within a transaction. If there's an existing transaction in progress, it should be suspended.
`MANDATORY`	The current method must run within a transaction. If there's no existing transaction in progress, an exception will be thrown.

Propagation	Description
NEVER	The current method should not run within a transaction. If there's an existing transaction in progress, an exception will be thrown.
NESTED	If there's an existing transaction in progress, the current method should run within the nested transaction (supported by the JDBC 3.0 save point feature) of this transaction. Otherwise, it should start a new transaction and run within its own transaction.

How It Works

Transaction propagation happens when a transactional method is called by another method. For example, suppose a customer would like to check out all books to purchase at the book shop cashier. To support this operation, you define the `Cashier` interface as follows:

```
package com.apress.springrecipes.bookshop;
...
public interface Cashier {

    public void checkout(List<String> isbns, String username);
}
```

You can implement this interface by delegating the purchases to a book shop bean by calling its `purchase()` method multiple times. Note that the `checkout()` method is made transactional by applying the `@Transactional` annotation.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {

    private BookShop bookShop;

    public void setBookShop(BookShop bookShop) {
        this.bookShop = bookShop;
    }

    @Transactional
    public void checkout(List<String> isbns, String username) {
        for (String isbn : isbns) {
            bookShop.purchase(isbn, username);
        }
    }
}
```

Then define a cashier bean in your bean configuration file and refer to the book shop bean for purchasing books.

```
<bean id="cashier"
    class="com.apress.springrecipes.bookshop.BookShopCashier">
    <property name="bookShop" ref="bookShop" />
</bean>
```

To illustrate the propagation behavior of a transaction, enter the data shown in Tables 8-6, 8-7, and 8-8 in your `bookshop` database.

Table 8-6. *Sample Data in the BOOK Table for Testing Propagation Behaviors*

ISBN	BOOK_NAME	PRICE
0001	The First Book	30
0002	The Second Book	50

Table 8-7. *Sample Data in the BOOK_STOCK Table for Testing Propagation Behaviors*

ISBN	STOCK
0001	10
0002	10

Table 8-8. *Sample Data in the ACCOUNT Table for Testing Propagation Behaviors*

USERNAME	BALANCE
user1	40

The REQUIRED Propagation Behavior

When the user `user1` checks out the two books from the cashier, the balance is sufficient to purchase the first book, but not the second.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        Cashier cashier = (Cashier) context.getBean("cashier");
        List<String> isbnList =
                Arrays.asList(new String[] { "0001", "0002" });
        cashier.checkout(isbnList, "user1");
    }
}
```

When the book shop's `purchase()` method is called by another transactional method, such as `checkout()`, it will run within the existing transaction by default. This default propagation behavior is called `REQUIRED`. That means there will be only one transaction whose boundary is the beginning and ending of the `checkout()` method. This transaction will only be committed at the end of the checkout() method. As a result, the user can purchase none of the books. Figure 8-2 illustrates the REQUIRED propagation behavior.

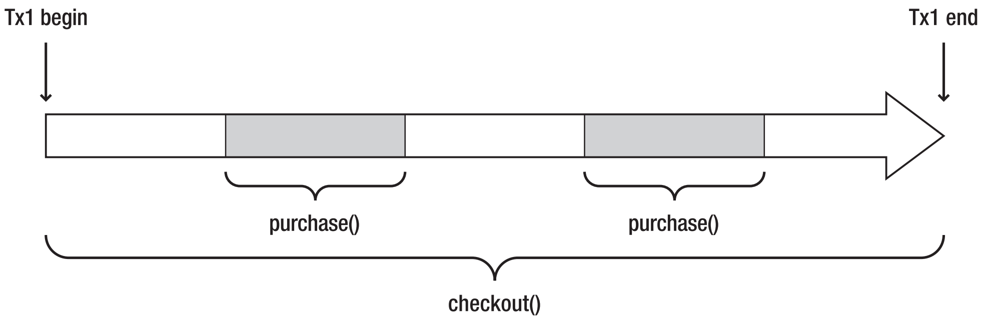

Figure 8-2. *The REQUIRED transaction propagation behavior*

However, if the purchase() method is called by a non-transactional method and there's no existing transaction in progress, it will start a new transaction and run within its own transaction.

The propagation transaction attribute can be defined in the @Transactional annotation. For example, you can set the REQUIRED behavior for this attribute as follows. In fact, this is unnecessary, as it's the default behavior.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void checkout(List<String> isbns, String username) {
        ...
    }
}
```

The REQUIRES_NEW Propagation Behavior

Another common propagation behavior is `REQUIRES_NEW`. It indicates that the method must start a new transaction and run within its new transaction. If there's an existing transaction in progress, it should be suspended first.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

In this case, there will be three transactions started in total. The first transaction is started by the `checkout()` method, but when the first `purchase()` method is called, the first transaction will be suspended and a new transaction will be started. At the end of the first `purchase()` method, the new transaction completes and commits. When the second `purchase()` method is called, another new transaction will be started. However, this transaction will fail and roll back. As a result, the first book will be purchased successfully while the second will not. Figure 8-3 illustrates the `REQUIRES_NEW` propagation behavior.

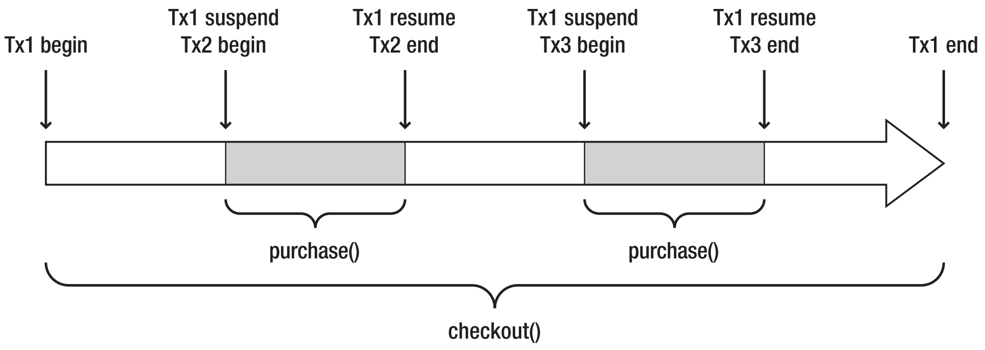

Figure 8-3. *The REQUIRES_NEW transaction propagation behavior*

Setting the Propagation Attribute in Transaction Advices, Proxies, and APIs

In a Spring 2.x transaction advice, the propagation transaction attribute can be specified in the `<tx:method>` element as follows:

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..."
```

```
            propagation="REQUIRES_NEW" />
    </tx:attributes>
</tx:advice>
```

In classic Spring AOP, the propagation transaction attribute can be specified in the transaction attributes of `TransactionInterceptor` and `TransactionProxyFactoryBean` as follows:

```
<property name="transactionAttributes">
    <props>
        <prop key="...">PROPAGATION_REQUIRES_NEW</prop>
    </props>
</property>
```

In Spring's transaction management API, the propagation transaction attribute can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
```

8-9. Setting the Isolation Transaction Attribute

Problem

When multiple transactions of the same application or different applications are operating concurrently on the same dataset, many unexpected problems may arise. You must specify how you expect your transactions to be isolated from one another.

Solution

The problems caused by concurrent transactions can be categorized into three types:

Dirty read: For two transactions T1 and T2, T1 reads a field that has been updated by T2 but not yet committed. Later, if T2 rolls back, the field read by T1 will be temporary and invalid.

Non-repeatable read: For two transactions T1 and T2, T1 reads a field and then T2 updates the field. Later, if T1 reads the same field again, the value will be different.

Phantom read: For two transactions T1 and T2, T1 reads some rows from a table and then T2 inserts new rows into the table. Later, if T1 reads the same table again, there will be additional rows.

In theory, transactions should be completely isolated from each other (i.e., serializable) to avoid all the mentioned problems. However, this isolation level will have great impact on performance because transactions have to run in serial order. In practice, transactions can run in lower isolation levels in order to improve performance.

A transaction's isolation level can be specified by the *isolation* transaction attribute. Spring supports five isolation levels, as shown in Table 8-9. These levels are defined in the `org.springframework.transaction.TransactionDefinition` interface.

Table 8-9. *Isolation Levels Supported by Spring*

Isolation	Description
`DEFAULT`	Uses the default isolation level of the underlying database. For most databases, the default isolation level is `READ_COMMITTED`.
`READ_UNCOMMITTED`	Allows a transaction to read uncommitted changes by other transactions. The dirty read, non-repeatable read, and phantom read problems may occur.
`READ_COMMITTED`	Allows a transaction to read only those changes that have been committed by other transactions. The dirty read problem can be avoided, but the non-repeatable read and phantom read problems may still occur.
`REPEATABLE_READ`	Ensures that a transaction can read identical values from a field multiple times. For the duration of this transaction, updates made by other transactions to this field are prohibited. The dirty read and non-repeatable read problems can be avoided, but the phantom read problem may still occur.
`SERIALIZABLE`	Ensures that a transaction can read identical rows from a table multiple times. For the duration of this transaction, inserts, updates, and deletes made by other transactions to this table are prohibited. All the concurrency problems can be avoided, but the performance will be low.

Note that transaction isolation is supported by the underlying database engine but not an application or a framework. However, not all database engines support all these isolation levels. You can change the isolation level of a JDBC connection by calling the `setTransactionIsolation()` method.

How It Works

To illustrate the problems caused by concurrent transactions, let's add two new operations to your book shop for increasing and checking the book stock.

```
package com.apress.springrecipes.bookshop;

public interface BookShop {
    ...
    public void increaseStock(String isbn, int stock);
    public int checkStock(String isbn);
}
```

Then you implement these operations as follows. Note that these two operations should also be declared as transactional.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional
    public void increaseStock(String isbn, int stock) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + " - Prepare to increase book stock");

        getJdbcTemplate().update(
                "UPDATE BOOK_STOCK SET STOCK = STOCK + ? " +
                "WHERE ISBN = ?",
                new Object[] { stock, isbn });

        System.out.println(threadName + " - Book stock increased by " + stock);
        sleep(threadName);

        System.out.println(threadName + " - Book stock rolled back");
        throw new RuntimeException("Increased by mistake");
    }

    @Transactional
    public int checkStock(String isbn) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + " - Prepare to check book stock");

        int stock = getJdbcTemplate().queryForInt(
                "SELECT STOCK FROM BOOK_STOCK WHERE ISBN = ?",
                new Object[] { isbn });

        System.out.println(threadName + " - Book stock is " + stock);
        sleep(threadName);

        return stock;
    }

    private void sleep(String threadName) {
        System.out.println(threadName + " - Sleeping");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {}
        System.out.println(threadName + " - Wake up");
    }
}
```

To simulate concurrency, your operations need to be executed by multiple threads. You can know the current status of the operations through the `println` statements. For each operation, you print a couple of messages to the console around the SQL statement's execution. The messages should include the thread name for you to know which thread is currently executing the operation.

After each operation executes the SQL statement, you ask the thread to sleep for 10 seconds. As you know, the transaction will be committed or rolled back immediately once the operation completes. Inserting a sleep statement can help to postpone the commit or rollback.

For the `increase()` operation, you eventually throw a `RuntimeException` to cause the transaction to roll back.

Before you start with the isolation level examples, enter the data from Tables 8-10 and 8-11 into your `bookshop` database.

Table 8-10. *Sample Data in the BOOK Table for Testing Isolation Levels*

ISBN	BOOK_NAME	PRICE
0001	The First Book	30

Table 8-11. *Sample Data in the BOOK_STOCK Table for Testing Isolation Levels*

ISBN	STOCK
0001	10

The READ_UNCOMMITTED and READ_COMMITTED Isolation Levels

`READ_UNCOMMITTED` is the lowest isolation level that allows a transaction to read uncommitted changes made by other transactions. You can set this isolation level in the `@Transaction` annotation of your `checkStock()` method.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}
```

You can create some threads to experiment on this transaction isolation level. In the following `Main` class, there are two threads you are going to create. Thread 1 increases the book stock while thread 2 checks the book stock. Thread 1 starts 5 seconds before thread 2.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 1");

        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 2");

        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}
        thread2.start();
    }
}
```

If you run the application, you will get the following result:

```
Thread 1 - Prepare to increase book stock
Thread 1 - Book stock increased by 5
Thread 1 - Sleeping
Thread 2 - Prepare to check book stock
Thread 2 - Book stock is 15
Thread 2 - Sleeping
Thread 1 - Wake up
Thread 1 - Book stock rolled back
Thread 2 - Wake up
```

First, thread 1 increased the book stock and then went to sleep. At that time, thread 1's transaction had not yet been rolled back. While thread 1 was sleeping, thread 2 started and attempted to read the book stock. With the `READ_UNCOMMITTED` isolation level, thread 2 would be able to read the stock value that had been updated by an uncommitted transaction.

However, when thread 1 wakes up, its transaction will be rolled back due to a `RuntimeException`, so the value read by thread 2 is temporary and invalid. This problem is known as *dirty read*, as a transaction may read values that are "dirty."

To avoid the dirty read problem, you should raise the isolation level of `checkStock()` to `READ_COMMITTED`.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;
```

```
public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_COMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}
```

If you run the application again, thread 2 won't be able to read the book stock until thread 1 has rolled back the transaction. In this way, the dirty read problem can be avoided by preventing a transaction from reading a field that has been updated by another uncommitted transaction.

```
Thread 1 - Prepare to increase book stock
Thread 1 - Book stock increased by 5
Thread 1 - Sleeping
Thread 2 - Prepare to check book stock
Thread 1 - Wake up
Thread 1 - Book stock rolled back
Thread 2 - Book stock is 10
Thread 2 - Sleeping
Thread 2 - Wake up
```

In order that the underlying database can support the `READ_COMMITTED` isolation level, it may acquire an *update lock* on a row that was updated but not yet committed. Then other transactions must wait to read that row until the update lock is released, which happens when the locking transaction commits or rolls back.

The REPEATABLE_READ Isolation Level

Now let's restructure the threads to demonstrate another concurrency problem. Swap the tasks of the two threads so that thread 1 checks the book stock before thread 2 increases the book stock.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 1");
        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 2");

        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}
        thread2.start();
    }
}
```

If you run the application, you will get the following result:

```
Thread 1 - Prepare to check book stock
Thread 1 - Book stock is 10
Thread 1 - Sleeping
Thread 2 - Prepare to increase book stock
Thread 2 - Book stock increased by 5
Thread 2 - Sleeping
Thread 1 - Wake up
Thread 2 - Wake up
Thread 2 - Book stock rolled back
```

First, thread 1 read the book stock and then went to sleep. At that time, thread 1's transaction had not yet been committed. While thread 1 was sleeping, thread 2 started and attempted to increase the book stock. With the READ_COMMITTED isolation level, thread 2 would be able to update the stock value that was read by an uncommitted transaction.

However, if thread 1 reads the book stock again, the value will be different from its first read. This problem is known as *non-repeatable read* because a transaction may read different values for the same field.

To avoid the non-repeatable read problem, you should raise the isolation level of checkStock() to REPEATABLE_READ.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public int checkStock(String isbn) {
        ...
    }
}
```

If you run the application again, thread 2 won't be able to update the book stock until thread 1 has committed the transaction. In this way, the non-repeatable read problem can be avoided by preventing a transaction from updating a value that has been read by another uncommitted transaction.

```
Thread 1 - Prepare to check book stock
Thread 1 - Book stock is 10
Thread 1 - Sleeping
Thread 2 - Prepare to increase book stock
Thread 1 - Wake up
Thread 2 - Book stock increased by 5
Thread 2 - Sleeping
Thread 2 - Wake up
Thread 2 - Book stock rolled back
```

In order that the underlying database can support the REPEATABLE_READ isolation level, it may acquire a *read lock* on a row that was read but not yet committed. Then other transactions must wait to update the row until the read lock is released, which happens when the locking transaction commits or rolls back.

The SERIALIZABLE Isolation Level

After a transaction has read several rows from a table, another transaction inserts new rows into the same table. If the first transaction reads the same table again, it will find additional rows that are different from the first read. This problem is known as *phantom read*. Actually, phantom read is very similar to non-repeatable read, but involves multiple rows.

To avoid the phantom read problem, you should raise the isolation level to the highest: SERIALIZABLE. Notice that this isolation level is the slowest, as it may acquire a read lock on the full table. In practice, you should always choose the lowest isolation level that can satisfy your requirements.

Setting the Isolation Level Attribute in Transaction Advices, Proxies, and APIs

In a Spring 2.x transaction advice, the isolation level can be specified in the `<tx:method>` element as follows:

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..."
            isolation="REPEATABLE_READ" />
    </tx:attributes>
</tx:advice>
```

In classic Spring AOP, the isolation level can be specified in the transaction attributes of TransactionInterceptor and TransactionProxyFactoryBean as follows:

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, ISOLATION_REPEATABLE_READ
        </prop>
    </props>
</property>
```

In Spring's transaction management API, the isolation level can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);
```

8-10. Setting the Rollback Transaction Attribute

Problem

By default, only unchecked exceptions (i.e., of type `RuntimeException` and `Error`) will cause a transaction to roll back, while checked exceptions will not. Sometimes you may wish to break this rule and set your own exceptions for rolling back.

Solution

The exceptions that cause a transaction to roll back or not can be specified by the *rollback* transaction attribute. Any exceptions not explicitly specified in this attribute will be handled by the default rollback rule (i.e., rolling back for unchecked exceptions and not rolling back for checked exceptions).

How It Works

A transaction's rollback rule can be defined in the `@Transactional` annotation via the `rollbackFor` and `noRollbackFor` attributes. These two attributes are declared as `Class[]`, so you can specify more than one exception for each attribute.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
            propagation = Propagation.REQUIRES_NEW,
            rollbackFor = IOException.class,
            noRollbackFor = ArithmeticException.class)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

In a Spring 2.x transaction advice, the rollback rule can be specified in the `<tx:method>` element. You can separate the exceptions with commas if there's more than one exception.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..."
            rollback-for="java.io.IOException"
            no-rollback-for="java.lang.ArithmeticException" />
        ...
    </tx:attributes>
</tx:advice>
```

In classic Spring AOP, the rollback rule can be specified in the transaction attributes of `TransactionInterceptor` and `TransactionProxyFactoryBean`. The minus sign indicates an exception to cause a transaction to roll back, while the plus sign indicates an exception to cause a transaction to commit.

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, -java.io.IOException,
            +java.lang.ArithmeticException
        </prop>
    </props>
</property>
```

In Spring's transaction management API, the rollback rule can be specified in a `RuleBasedTransactionAttribute` object. As it implements the `TransactionDefinition` interface, it can be passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
RuleBasedTransactionAttribute attr = new RuleBasedTransactionAttribute();
attr.getRollbackRules().add(
    new RollbackRuleAttribute(IOException.class));
attr.getRollbackRules().add(
    new NoRollbackRuleAttribute(SendFailedException.class));
```

8-11. Setting the Timeout and Read-Only Transaction Attributes

Problem

As a transaction may acquire locks on rows and tables, a long transaction will tie up resources and have an impact on overall performance. Besides, if a transaction only reads but does not update data, the database engine could optimize this transaction. You can specify these attributes to increase the performance of your application.

Solution

The *timeout* transaction attribute indicates how long your transaction can survive before it is forced to roll back. This can prevent a long transaction from tying up resources. The *read-only* attribute indicates that this transaction will only read but not update data. That can help the database engine optimize the transaction.

How It Works

The timeout and read-only transaction attributes can be defined in the `@Transactional` annotation. Note that timeout is measured in seconds.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
            isolation = Isolation.REPEATABLE_READ,
            timeout = 30,
            readOnly = true)
    public int checkStock(String isbn) {
        ...
    }
}
```

In a Spring 2.x transaction advice, the timeout and read-only transaction attributes can be specified in the `<tx:method>` element.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..."
            timeout="30"
            read-only="true" />
    </tx:attributes>
</tx:advice>
```

In classic Spring AOP, the timeout and read-only transaction attributes can be specified in the transaction attributes of `TransactionInterceptor` and `TransactionProxyFactoryBean`.

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, timeout_30, readOnly
        </prop>
    </props>
</property>
```

In Spring's transaction management API, the timeout and read-only transaction attributes can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setTimeout(30);
def.setReadOnly(true);
```

8-12. Managing Transactions with Load-Time Weaving

Problem

By default, Spring's declarative transaction management is enabled via its AOP framework. However, as Spring AOP can only advise public methods of beans declared in the IoC container, you are restricted to manage transactions within this scope using Spring AOP. Sometimes you may wish to manage transactions for non-public methods, or methods of objects created outside the Spring IoC container (e.g., domain objects).

Solution

Spring 2.5 also provides an AspectJ aspect named `AnnotationTransactionAspect` that can manage transactions for any methods of any objects, even if the methods are non-public or the objects are created outside the Spring IoC container. This aspect will manage transactions for any methods with the `@Transactional` annotation. You can choose either AspectJ's compile-time weaving or load-time weaving to enable this aspect.

How It Works

First of all, let's create a domain class `Book`, whose instances (i.e., domain objects) may be created outside the Spring IoC container.

```
package com.apress.springrecipes.bookshop;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.jdbc.core.JdbcTemplate;

@Configurable
public class Book {

    private String isbn;
    private String name;
    private int price;

    // Constructors, Getters and Setters
    ...

    private JdbcTemplate jdbcTemplate;
    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void purchase(String username) {
        jdbcTemplate.update(
                "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
                "WHERE ISBN = ?",
                new Object[] { isbn });

        jdbcTemplate.update(
                "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
                "WHERE USERNAME = ?",
                new Object[] { price, username });
    }
}
```

This domain class has a `purchase()`method that will deduct the current book instance's stock and the user account's balance from the database. To utilize Spring's powerful JDBC support features, you can inject the JDBC template via setter injection.

You can use Spring's load-time weaving support to inject a JDBC template into book domain objects. You have to annotate this class with `@Configurable` to declare that this type of object is configurable in the Spring IoC container. Moreover, you can annotate the JDBC template's setter method with `@Autowired` to have it auto-wired.

Spring includes an AspectJ aspect, `AnnotationBeanConfigurerAspect`, in its aspect library for configuring object dependencies even if these objects are created outside the IoC container. To enable this aspect, you just define the `<context:spring-configured>` element in your bean configuration file. To weave this aspect into your domain classes at load time, you also have to define `<context:load-time-weaver>`. Finally, to auto-wire the JDBC template into book domain objects via `@Autowired`, you need `<context:annotation-config>` also.

■ **Note** To use the Spring aspect library for AspectJ, you have to include `spring-aspects.jar` (located in the `dist/weaving` directory of the Spring installation) in your classpath.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:load-time-weaver />
    <context:annotation-config />

    <context:spring-configured />

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/bookshop;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

In this bean configuration file, you can define a JDBC template on a data source, and then it will be auto-wired into book domain objects for them to access the database.

Now you can create the following `Main` class to test this domain class. Of course, there's no transaction support at this moment.

```
package com.apress.springrecipes.bookshop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        Book book = new Book("0001", "My First Book", 30);
        book.purchase("user1");
    }
}
```

For a simple Java application, you can weave this aspect into your classes at load time with the Spring agent specified as a VM argument.

```
java -javaagent:c:/spring-framework-2.5/dist/weaving/spring-agent.jar➥
com.apress.springrecipes.bookshop.Main
```

To enable transaction management for a domain object's method, you can simply annotate it with `@Transactional`, just as you did for methods of Spring beans.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.transaction.annotation.Transactional;

@Configurable
public class Book {
    ...
    @Transactional
    public void purchase(String username) {
        ...
    }
}
```

Finally, to enable Spring's `AnnotationTransactionAspect` for transaction management, you just define the `<tx:annotation-driven>` element and set its mode to `aspectj`. Then the transaction aspect will automatically get enabled. You also have to provide a transaction manager for this aspect. By default, it will look for a transaction manager whose name is `transactionManager`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven mode="aspectj" />

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

8-13. Summary

In this chapter, you have learned the importance of transaction management for ensuring data integrity and consistency. Without transaction management, your data and resources may be corrupted and left in an inconsistent state.

Spring's core transaction management abstraction is `PlatformTransactionManager`. This encapsulates a set of technology-independent methods that allow you to manage transactions without having to know much about the underlying transaction APIs. Spring has several built-in transaction manager implementations for different transaction management APIs.