Part - 1

Spring and Transaction

Ye Win

12/08/2014

Contents

- Who This Slide Is For?
- Introduction
- Local Vs Global Transactions
- Programmatic Vs Declarative
- Spring Transaction Abstractions
- Programmatic & Declarative (Practical)
- Conclusion

- Ps. Slide notes from Spring Transaction Abstractions Section are soul for this slide.

Who This Slide Is For?

- This session will let you know how to manage transactions in Spring. You can understand the programmatic as well as declarative transaction management in Spring Framework.

- You can take the source codes of the examples given in this slide from links available at

https://drive.google.com/open?id=0B72s1dEbEPzBLS00M0JNZDd0X3c&authuser=0

Introduction

Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as **ACID**:

1. **Atomicity:** A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.

2. **Consistency:** This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

3. **Isolation:** There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.

4. **Durability:** Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring supports both programmatic and declarative transaction management and Spring transaction management can be implemented without a need of application server.

Local Vs Global Tansactions

- Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

- Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic Vs Declarative

- Spring supports two types of transaction management:

  1. **Programmatic transaction management:** This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

  2. **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

- Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Spring Transaction Abstractions

We have to implements mainly three transaction interface.

*1)org.springframework.transaction.**PlatformTransactionManager***

2)org.springframework.transaction.**TransactionDefinition**

3)org.springframework.transaction.**TransactionStatus**

(1) Interface PlatformTransactionManager

Method Summary	
Modifier and Type	Method and Description
void	commit(TransactionStatus status) Commit the given transaction, with regard to its status.
TransactionStatus	getTransaction(TransactionDefinition definition) Return a currently active transaction or create a new one, according to the specified propagation behavior.
void	rollback(TransactionStatus status) Perform a rollback of the given transaction.

(2) Interface TransactionDefinition

Field Summary

Fields	
Modifier and Type	Field and Description
static int	ISOLATION_DEFAULT Use the default isolation level of the underlying datastore. Eg. READ COMMITTED is the default isolation level for the Microsoft SQL Server Database Engine.
static int	ISOLATION_READ_COMMITTED Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
static int	ISOLATION_READ_UNCOMMITTED Indicates that dirty reads, non-repeatable reads and phantom reads can occur.

(2) Interface TransactionDefinition

static int	**ISOLATION_REPEATABLE_READ** **Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.**
static int	ISOLATION_SERIALIZABLE Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.
static int	PROPAGATION_MANDATORY Support a current transaction; throw an exception if no current transaction exists.

(2) Interface TransactionDefinition

static int	**PROPAGATION_NESTED** **Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else.**
static int	PROPAGATION_NEVER Do not support a current transaction; throw an exception if a current transaction exists.
static int	PROPAGATION_NOT_SUPPORTED Do not support a current transaction; rather always execute non-transactionally.
static int	PROPAGATION_REQUIRED Support a current transaction; create a new one if none exists.

(2) Interface TransactionDefinition

static int	**PROPAGATION_REQUIRES_NEW** **Create a new transaction, suspending the current transaction if one exists.**
static int	PROPAGATION_SUPPORTS Support a current transaction; execute non-transactionally if none exists.
static int	TIMEOUT_DEFAULT Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

(2) Interface TransactionDefinition

Method Summary

All MethodsInstance MethodsAbstract Methods	
Modifier and Type	Method and Description
int	getIsolationLevel() Return the isolation level.
String	getName() Return the name of this transaction.
int	getPropagationBehavior() Return the propagation behavior.
int	getTimeout() Return the transaction timeout.
boolean	isReadOnly() Return whether to optimize as a read-only transaction.

(3) Interface TransactionStatus

Method Summary

Modifier and Type	Method and Description
void	flush() Flush the underlying session to the datastore, if applicable: for example, all affected Hibernate/JPA sessions.
boolean	hasSavepoint() Return whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint.
boolean	isCompleted() Return whether this transaction is completed, that is, whether it has already been committed or rolled back.
boolean	isNewTransaction() Return whether the present transaction is new (else participating in an existing transaction, or potentially not running in an actual transaction in the first place).
boolean	isRollbackOnly() Return whether the transaction has been marked as rollback-only (either by the application or by the transaction infrastructure).
void	setRollbackOnly() Set the transaction rollback-only.

Conclusion

- The Spring transaction management mechanism is very powerful.
- It can integrate any persistence framework
- A future post Part – 2 will explain more details of Programmatic and Declarative Transaction.

Do you have Questions ?

Please Discuss :


THANK
YOU