## 12. Transaction Management

### 12.1 Introduction to Spring Framework transaction management

### 12.2 Advantages of the Spring Framework’s transaction support model

### 12.3 Understanding the Spring Framework transaction abstraction

Spring事务原理，事务策略是通过org.springframework.transaction.PlatformTransactionManager接口。

**public** **interface** PlatformTransactionManager {

TransactionStatus getTransaction(

TransactionDefinition definition) **throws** TransactionException;

**void** commit(TransactionStatus status) **throws** TransactionException;

**void** rollback(TransactionStatus status) **throws** TransactionException;

}

TransactionException会被任何PlatformTransactionManager接口方法在运行状态下丢出异常，

getTranscation(…)方法会返回一个TransactionStatus依托于TransactionDefintion。

TransactionDefintion接口定了以下内容：

* Isolation（独立性）：
* Propageation（传播性）：所有的代码都可以在一个事务的局域内运行
* Timeout（逾期性）：在事务逾期时间范围前，事务能够常态地运行
* Read\_only status（只读）：一个只读事务状态，只读状态是一种优化措施，详见Hibernate。

通过TransactionStatus接口提供极其简单的方式帮我们控制事务的执行以及查询事务的状态，这种场景似曾相识，让我们查看下常见的事务的API

**public** **interface** TransactionStatus **extends** SavepointManager {

**boolean** isNewTransaction();

**boolean** hasSavepoint();

**void** setRollbackOnly();

**boolean** isRollbackOnly();

**void** flush();

**boolean** isCompleted();

}

在JDBC, JTA, Hibernate等上面，PlatformTransactionManager的实现是最为普遍的，

假设你定义JDBC DataSource

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

关联到PlatformTransactionManager上通过成员变量DataSource

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

假设你采用JTA方式获取DataSource

通过JNDI配合JtaTransactionManager结合

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jee

http://www.springframework.org/schema/jee/spring-jee.xsd">

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

*<!-- other <bean/> definitions here -->*

</beans>

JtaTransactionManager不需要知道关于DataSource，因为它用了全局事务管理架构

假设你采用的是Hibernate方式

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="mappingResources">

<list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>

hibernate.dialect=${hibernate.dialect}

</value>

</property>

</bean>

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory" />

</bean>

### 12.4 Synchronizing resources with transactions

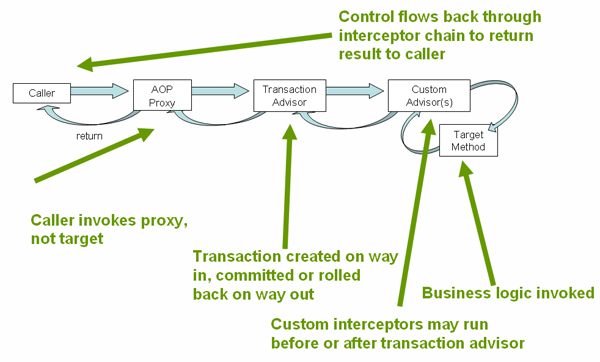
#### 12.4.1 High-level synchronization approach

Spring提倡使用ORM api或者采用JDBCTemplate

#### 12.4.2 Low-level synchronization approach

### 12.5 Declarative transaction management（声明式事务管理）

#### 12.5.1 Understanding the Spring Framework’s declarative transaction implementation



#### 12.5.2 Example of declarative transaction implementation

**package** x.y.service;

**public** **interface** FooService {

Foo getFoo(String fooName);

Foo getFoo(String fooName, String barName);

**void** insertFoo(Foo foo);

**void** updateFoo(Foo foo);

}

**public** **class** DefaultFooService **implements** FooService {

**public** Foo getFoo(String fooName) {

**throw** **new** UnsupportedOperationException();

}

**public** Foo getFoo(String fooName, String barName) {

**throw** **new** UnsupportedOperationException();

}

**public** **void** insertFoo(Foo foo) {

**throw** **new** UnsupportedOperationException();

}

**public** **void** updateFoo(Foo foo) {

**throw** **new** UnsupportedOperationException();

}

}

假设FooService接口，在事务的情况下，get方法在只读的场景中，其它的方法是读写的场景下。

*<!-- from the file 'context.xml' -->*

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

*<!-- this is the service object that we want to make transactional -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->*

<tx:advice id="txAdvice" transaction-manager="txManager">

*<!-- the transactional semantics... -->*

<tx:attributes>

*<!-- all methods starting with 'get' are read-only -->*

<tx:method name="get\*" read-only="true"/>

*<!-- other methods use the default transaction settings (see below) -->*

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

*<!-- ensure that the above transactional advice runs for any execution*

*of an operation defined by the FooService interface -->*

<aop:config>

<aop:pointcut id="fooServiceOperation" expression="execution(\* x.y.service.FooService.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>

</aop:config>

*<!-- don't forget the DataSource -->*

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>

<property name="username" value="scott"/>

<property name="password" value="tiger"/>

</bean>

*<!-- similarly, don't forget the PlatformTransactionManager -->*

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

*<!-- other <bean/> definitions here -->*

</beans>

#### 12.5.3 Rolling back a declarative transaction

在上一章节中，我们谈到声名式事务如何配置，那么接下来谈下在roll-back后，异常信息的捕获，默认场景下，spring基础设置只提供了运行期的unchecked异常。

你能够配置checked异常，通过下面的方式

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true" rollback-for="NoProductInStockException"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

你也能指定roll-back异常条件

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="\*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>

</tx:attributes>

</tx:advice>

#### 12.5.4 Configuring different transactional semantics for different beans

为不同的bean配置不同的事务场景

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<aop:config>

<aop:pointcut id="serviceOperation"

expression="execution(\* x.y.service..\*Service.\*(..))"/>

<aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

</aop:config>

*<!-- these two beans will be transactional... -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<bean id="barService" class="x.y.service.extras.SimpleBarService"/>

*<!-- ... and these two beans won't -->*

<bean id="anotherService" class="org.xyz.SomeService"/> *<!-- (not in the right package) -->*

<bean id="barManager" class="x.y.service.SimpleBarManager"/> *<!-- (doesn't end in 'Service') -->*

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

*<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->*

</beans>

不同的bean的事务配置情况

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<aop:config>

<aop:pointcut id="defaultServiceOperation"

expression="execution(\* x.y.service.\*Service.\*(..))"/>

<aop:pointcut id="noTxServiceOperation"

expression="execution(\* x.y.service.ddl.DefaultDdlManager.\*(..))"/>

<aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

<aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

</aop:config>

*<!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- this bean will also be transactional, but with totally different transactional settings -->*

<bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

<tx:advice id="defaultTxAdvice">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<tx:advice id="noTxAdvice">

<tx:attributes>

<tx:method name="\*" propagation="NEVER"/>

</tx:attributes>

</tx:advice>

*<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->*

</beans>

#### 12.5.5 <tx:advice/> settings

#### 12.5.6 Using @Transactional

在同个类的情况，事务的不同配置情况

*@Transactional(readOnly = true)*

**public** **class** DefaultFooService **implements** FooService {

**public** Foo getFoo(String fooName) {

*// do something*

}

*// these settings have precedence for this method*

*@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)*

**public** **void** updateFoo(Foo foo) {

*// do something*

}

}

使用@Transactional配置不同事物管理方式

**public** **class** TransactionalService {

*@Transactional("order")*

**public** **void** setSomething(String name) { ... }

*@Transactional("account")*

**public** **void** doSomething() { ... }

}

<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="order"/>

</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="account"/>

</bean>

##### 定制事务的注解

*@Target({ElementType.METHOD, ElementType.TYPE})*

*@Retention(RetentionPolicy.RUNTIME)*

*@Transactional("order")*

**public** *@interface* OrderTx {

}

*@Target({ElementType.METHOD, ElementType.TYPE})*

*@Retention(RetentionPolicy.RUNTIME)*

*@Transactional("account")*

**public** *@interface* AccountTx {

}

#### 12.5.7 Transaction propagation

一、Propagation取值：

REQUIRED（默认值）：在有transaction状态下执行；如当前没有transaction，则创建新的transaction；

SUPPORTS：如当前有transaction，则在transaction状态下执行；如果当前没有transaction，在无transaction状态下执行；

MANDATORY：必须在有transaction状态下执行，如果当前没有transaction，则抛出异常IllegalTransactionStateException；

REQUIRES\_NEW：创建新的transaction并执行；如果当前已有transaction，则将当前transaction挂起；

NOT\_SUPPORTED：在无transaction状态下执行；如果当前已有transaction，则将当前transaction挂起；

NEVER：在无transaction状态下执行；如果当前已有transaction，则抛出异常IllegalTransactionStateException。

二、REQUIRED与REQUIRED\_NEW

上面描述的6种propagation属性配置中，最难以理解，并且容易在transaction设计时出现问题的是REQUIRED和REQURED\_NEW这两者的区别。当程序在某些情况下抛出异常时，如果对于这两者不够了解，就可能很难发现而且解决问题。

下面我们给出三个场景进行分析：

场景一：

ServiceA.java:

public class ServiceA {  
    @Transactional  
    public void callB() {  
        serviceB.doSomething();  
    }  
}

ServiceB.java

public class ServiceB {  
    @Transactional  
    public void doSomething() {  
        throw new RuntimeException("B throw exception");  
    }  
}

这种情况下，我们只需要在调用ServiceA.callB时捕获ServiceB中抛出的运行时异常，则transaction就会正常的rollback。

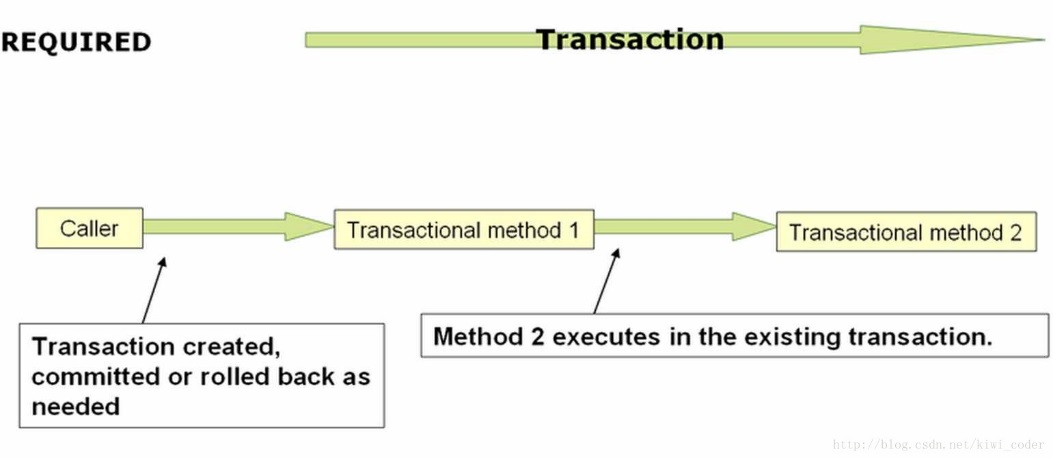
场景二

在保持场景一中ServiceB不变，在ServiceA中调用ServiceB的doSomething时去捕获这个异常，如下：

public class ServiceA {  
    @Transactional  
    public void callB() {  
        try {  
            serviceB.doSomething();  
        } catch (RuntimeException e) {  
            System.err.println(e.getMessage());  
        }  
    }  
}

这个时候，我们再调用ServiceA的callB。程序会抛出org.springframework.transaction.UnexpectedRollbackException: Transaction rolled back because it has been marked as rollback-only这样一个异常信息。原因是什么呢？

因为在ServiceA和ServiceB中的@Transactional propagation都采用的默认值：REQUREID。根据我们前面讲过的REQUIRED特性，当ServiceA调用ServiceB的时候，他们是处于同一个transaction中。如下图所示：



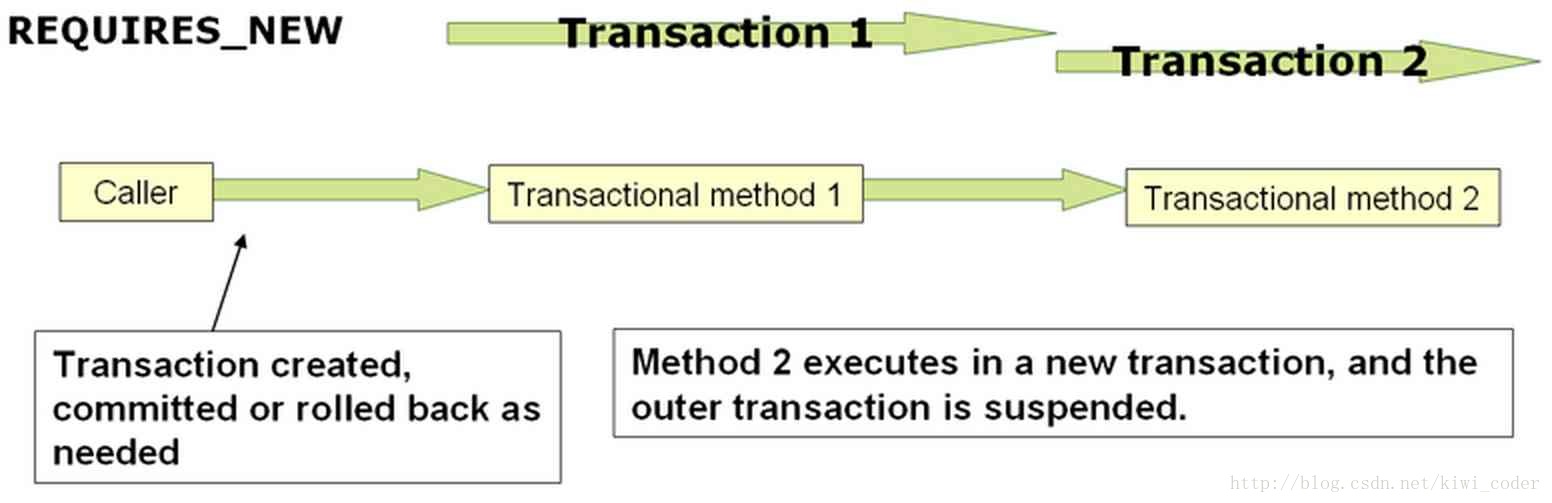
当ServiceB中抛出了一个异常以后，ServiceB会把当前的transaction标记为需要rollback。但是ServiceA中捕获了这个异常，并进行了处理，认为当前transaction应该正常commit。此时就出现了前后不一致，也就是因为这样，抛出了前面的UnexpectedRollbackException。

场景三

在保持场景二中ServiceA不变，修改ServiceB中方法的propagation配置为REQUIRES\_NEW，如下：

public class ServiceB {  
    @Transactional(propagation = Propagation.REQUIRES\_NEW)  
    public void doSomething() {  
        throw new RuntimeException("B throw exception");  
    }  
}

此时，程序可以正常的退出了，也没有抛出UnexpectedRollbackException。原因是因为当ServiceA调用ServiceB时，serviceB的doSomething是在一个新的transaction中执行的。如下图所示：



所以，当doSomething抛出异常以后，仅仅是把新创建的transaction rollback了，而不会影响到ServiceA的transaction。ServiceA就可以正常的进行commit。

当然这里把ServiceA和ServiceB放在两个独立的transaction是否成立，还需要再多多考虑你的业务需求。

#### 12.5.8 Advising transactional operations

通过ordered接口，进行切面优先顺序

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- the profiling advice -->*

<bean id="profiler" class="x.y.SimpleProfiler">

*<!-- execute before the transactional advice (hence the lower order number) -->*

\_\_<property name="order" value="1\_\_"/>

</bean>

<aop:config>

<aop:pointcut id="entryPointMethod" expression="execution(\* x.y..\*Service.\*(..))"/>

*<!-- will execute after the profiling advice (c.f. the order attribute) -->*

<aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod" \_\_order="2\_\_"/>

*<!-- order value is higher than the profiling aspect -->*

<aop:aspect id="profilingAspect" ref="profiler">

<aop:pointcut id="serviceMethodWithReturnValue"

expression="execution(!void x.y..\*Service.\*(..))"/>

<aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>

</aop:aspect>

</aop:config>

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

*<!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->*

</beans>

#### 12.5.9 Using @Transactional with AspectJ

### 12.6 Programmatic transaction management

Spring提供两种动态事务管理方式

* 使用TransactionTemplate
* 使用PlatformTransactionManager
* Spring推荐使用TransactionTemplate
* 推荐JTAUserTransaction API方式

#### 12.6.1 Using the TransactionTemplate

**public** **class** SimpleService **implements** Service {

*// single TransactionTemplate shared amongst all methods in this instance*

**private** **final** TransactionTemplate transactionTemplate;

*// use constructor-injection to supply the PlatformTransactionManager*

**public** SimpleService(PlatformTransactionManager transactionManager) {

Assert.notNull(transactionManager, "The *'transactionManager*' argument must not be null.");

**this**.transactionTemplate = **new** TransactionTemplate(transactionManager);

}

**public** Object someServiceMethod() {

**return** transactionTemplate.execute(**new** TransactionCallback() {

*// the code in this method executes in a transactional context*

**public** Object doInTransaction(TransactionStatus status) {

updateOperation1();

**return** resultOfUpdateOperation2();

}

});

}

}

如果你想通过匿名类获取无返回值的

transactionTemplate.execute(**new** **TransactionCallbackWithoutResult**() {

**protected** **void** doInTransactionWithoutResult(TransactionStatus status) {

updateOperation1();

updateOperation2();

}

});

如果你想rollback，可以通过TransactionStatus状态，然后调用setRollBackOnly

transactionTemplate.execute(**new** TransactionCallbackWithoutResult() {

**protected** **void** doInTransactionWithoutResult(TransactionStatus status) {

**try** {

updateOperation1();

updateOperation2();

} **catch** (SomeBusinessExeption ex) {

**status.setRollbackOnly();**

}

}

});

可以通过配置事务模板的属性

<bean id="sharedTransactionTemplate"

class="org.springframework.transaction.support.TransactionTemplate">

<property name="isolationLevelName" value="ISOLATION\_READ\_UNCOMMITTED"/>

<property name="timeout" value="30"/>

</bean>"

#### 12.6.2 Using the PlatformTransactionManager

DefaultTransactionDefinition def = **new** DefaultTransactionDefinition();

*// explicitly setting the transaction name is something that can only be done programmatically*

def.setName("SomeTxName");

def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);

**try** {

*// execute your business logic here*

}

**catch** (MyException ex) {

txManager.rollback(status);

**throw** ex;

}

txManager.commit(status);

### 12.7 Choosing between programmatic and declarative transaction management

选择动态和声名式事务管理，依据于你的业务场景，比如你的有少量的事务控制，可以选择动态事务处理，在web application中就比较常用。

### 12.8 Application server-specific integration