# 名词解释

JPA

全称[Java](http://lib.csdn.net/base/17) Persistence API。

JPA通过JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到[数据库](http://lib.csdn.net/base/14)中。

JTA

即Java Transaction API，译为Java事务API。

JTA允许应用程序执行分布式事务处理——在两个或多个网络计算机资源上访问并且更新数据。JDBC驱动程序的JTA支持极大地增强了数据访问能力。

JMS

即[Java消息服务](http://baike.baidu.com/view/3292569.htm)（Java Message Service）。

JMS（Java Messaging Service）是[Java](http://baike.baidu.com/view/29.htm)平台上有关面向消息中间件的技术规范，它便于消息系统中的Java[应用程序](http://baike.baidu.com/view/330120.htm)进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发，翻译为Java消息服务。

# 事物的传播特性和隔离级别

## 什么是事物的传播特性？

在我们用SSH开发项目的时候，我们一般都是将事务设置在Service层 那么当我们调用Service层的一个方法的时候它能够保证我们的这个方法中执行的所有的对数据库的更新操作保持在一个事务中，在事务层里面调用的这些方法要么全部成功，要么全部失败。那么事务的传播特性也是从这里说起的。

如 果你在你的Service层的这个方法中，除了调用了Dao层的方法之外，还调用了本类的其他的Service方法，那么在调用其他的Service方法 的时候，这个事务是怎么规定的呢，我必须保证我在我方法里掉用的这个方法与我本身的方法处在同一个事务中，否则如果保证事物的一致性。事务的传播特性就是 解决这个问题的，“事务是会传播的”在Spring中有针对传播特性的多种配置我们大多数情况下只用其中的一 种:PROPGATION\_REQUIRED：这个配置项的意思是说当我调用service层的方法的时候开启一个事务(具体调用那一层的方法开始创建事 务，要看你的aop的配置),那么在调用这个service层里面的其他的方法的时候,如果当前方法产生了事务就用当前方法产生的事务，否则就创建一个新 的事务。这个工作使由Spring来帮助我们完成的。

## 传播特性分类

* PROPAGATION\_REQUIRED: 如果存在一个事务，则支持当前事务。如果没有事务则开启
* PROPAGATION\_SUPPORTS: 如果存在一个事务，使用当前事务。如果没有事务，则非事务的执行
* PROPAGATION\_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。
* PROPAGATION\_REQUIRES\_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。
* PROPAGATION\_NOT\_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务。
* PROPAGATION\_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常
* PROPAGATION\_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行

## 隔离级别

数据库事务的隔离级别有4个，由低到高依次为Read uncommitted 、Read committed 、Repeatable read 、Serializable ，这四个级别可以逐个解决脏读 、不可重复读 、幻读 这几类问题。

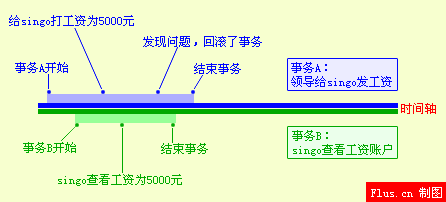
|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| Read uncommitted | √ | √ | √ |
| Read committed | × | √ | √ |
| Repeatable read | × | × | √ |
| Serializable | × | × | × |

√: 可能出现    ×: 不会出现

注意：我们讨论隔离级别的场景，主要是在多个事务并发 的情况下，因此，接下来的讲解都围绕事务并发。

### Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而singo正好去查看账户，发现工资已经到账，是5000元整，非常高 兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有 2000元，singo空欢喜一场。



出现上述情况，即我们所说的脏读 ，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted 时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

### Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好在网上转账，把singo工资卡的2000元转到另一账户，并在 singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有钱，扣款失败，singo十分纳闷，明明卡里有钱，为 何......

出现上述情况，即我们所说的不可重复读 ，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed 时，避免了脏读，但是可能会造成不可重复读。

大多数数据库的默认级别就是Read committed，比如Sql Server , Oracle。如何解决不可重复读这一问题，请看下一个隔离级别。

### Repeatable read 重复读

当隔离级别设置为Repeatable read 时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read避免了不可重复读，但还有可能出现幻读 。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费记录。有一天，她正在查询到singo当月信用卡的总消费金额 （select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出 现了幻觉，幻读就这样产生了。

注：Mysql的默认隔离级别就是Repeatable read。

### Serializable 序列化

Serializable 是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

# jta

1. Transaction 分两种，Local Transaction 和 Global Transaction。   
涉及到一个Connection的Commit，称为Local Transaction。   
涉及到多个Connection的Commit，称为Global Transaction。    
  
2. Global Transaction 需要XA接口（包括在JTA里面）的支持。   
import javax.sql.XAConnection;   
import javax.transaction.xa.Xid;   
import javax.transaction.xa.XAResource;   
import javax.transaction.xa.XAException;   
import javax.transaction.Transaction;   
import javax.transaction.TransactionManager;   
  
其中的   
javax.sql.XAConnection;   
javax.transaction.xa.Xid;   
javax.transaction.xa.XAResource;   
  
这些XA接口的实现，需要数据库的JDBC提供。  
数据库本身要支持XA。数据库的JDBC也要提供XA的实现。  
  
Oracle, Sybase, DB2, SQL Server等大型数据库才支持XA, 支持Global Transaction。   
My SQL 连Local Transaction都支持不好，更别说Global Transation了。   
  
3. XA需要两阶段提交 -- prepare 和 commit.   
假设有两个Connection, con1, con2, 大体的过程如下，

[复制代码](javascript:void(0);)

con1 = XAResouce1.getConnection...

con2 = XAResouce2.getConnection...

con1 do some thing.

con2 do some thing.

after they finish.

pre1 = XAResouce1.prepare();

pre2 = XAResouce2.prepare();

if( both pre1 and pre2 are OK）{

XAResouce1 and 2 commit

}else {

XAResouce1 and 2 rollback

}

[复制代码](javascript:void(0);)

前面有人讲了，在XAResouce1 and 2 commit的时候，   
可能XAResouce1 commit() 成功了，XAResouce2 commit()失败了。   
这时候，会抛出一个 “启发式异常”。程序可以处理这个异常。比如，XAResouce.recover()之类。   
但一般情况下，还真没别的办法，需要数据管理员根据数据操作日志 undo所有的操作，或者恢复数据备份。   
有的数据库在进行数据操作的时候，会生成一个“反操作”日志。比如，insert 对 delete, 等。   
  
4. TransactionManager的实现能够处理多个XAResouce(一个XAResouce list)的情况。   
比如Tyrex。或JBoss等EJB Server的Transaction实现代码

注释：XA：XA协议由Tuxedo首先提出的，并交给X/Open组织，作为资源 管理器（数据库）与事务管理器的接口标准。目前，Oracle、Informix、DB2和Sybase等各大数据库厂家都提供对XA的支持。XA协议采 用两阶段提交方式来管理分布式事务。XA接口提供资源管理器与事务管理器之间进行通信的标准接口。XA协议包括两套函数，以xa\_开头的及以ax\_开头 的。

以下的函数使事务管理器可以对资源管理器进行的操作：

1）xa\_open,xa\_close：建立和关闭与资源管理器的连接。

2）xa\_start,xa\_end：开始和结束一个本地事务。

3）xa\_prepare,xa\_commit,xa\_rollback：预提交、提交和回滚一个本地事务。

4）xa\_recover：回滚一个已进行预提交的事务。

5）ax\_开头的函数使资源管理器可以动态地在事务管理器中进行注册，并可以对XID(TRANSACTION IDS)进行操作。

6）ax\_reg,ax\_unreg；允许一个资源管理器在一个TMS(TRANSACTION MANAGER SERVER)中动态注册或撤消注册。

## JTA、JTS、XA

**JTA**

定义了一套接口，其中约定了几种主要的角色：TransactionManager、UserTransaction、 Transaction、XAResource，并定义了这些 角色之间需要遵守的规范，如Transaction的委托给 TransactionManager等。

**JTS**

 JTS也是一组规范，上面提到JTA中需要角色之间的交互，那应该如何交互？JTS就是约定了交互细节的规范。

 总体上来说JTA更多的是从框架的角度来约定程序角色的接口，而JTS则是从具体实现的角度来约定程序角色之间的接口，两者各司其职。

 因为JTA相对来说，更高层一些，我们主要关注JTA。

**XA**

 XA协议，规定事务管理器和资源管理器接口，采用二阶段提交协议。

JTA事务有效的屏蔽了底层事务资源，但是与本地事务相比，XA协议的系统开销大。

## 总结

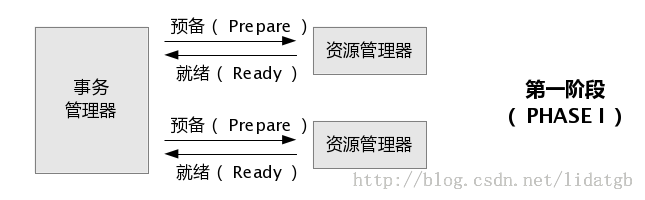
### 关键接口

 先介绍一下JTA的规范接口中，主要围绕以下几个接口：

* UserTransaction：编程人员接口---atomic 就是对其的一个实现
* TransactionManager：留给厂商实现的与事务管理有关的接口
* Transaction：留给厂商实现的事务
* XAResource：留给厂商实现的与持久化资源有关的接口

### 二阶段提交协议

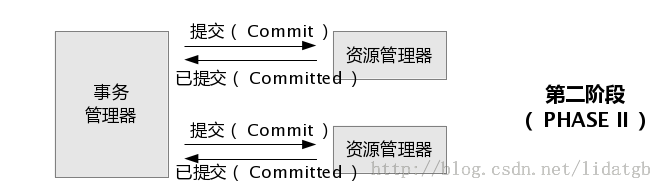
#### 第一阶段



应用程序调用了事务管理器的提交方法，第一阶段分为两个步骤：

* 事务管理器通知参与该事务的各个资源管理器，通知他们开始准备事务。
* 资源管理器接收到消息后开始准备阶段，写好事务日志并执行事务，但不提交，然后将是否就绪的消息返回给事务管理器（此时已经将事务的大部分事情做完，以后的内容耗时极小）。

#### 第二阶段



第二阶段也分为两个步骤：

* 事务管理器在接受各个消息后，开始分析，如果有任意其一失败，则发送回滚命令，否则发送提交命令。
* 各个资源管理器接收到命令后，执行（耗时很少），并将提交消息返回给事务管理器。

    事务管理器接受消息后，事务结束，应用程序继续执行。

#### 为什么需要两阶段提交

为什么要分两步执行？一是因为分两步，就有了事务管理器统一管理的机会；二尽可能晚地提交事务，让事务在提交前尽可能地完成所有能完成的工作，这样，最后的提交阶段将是耗时极短，耗时极短意味着操作失败的可能性也就降低。

    同时，二阶段提交协议为了保证事务的一致性，不管是事务管理器还是各个资源管理器，每执行一步操作，都会记录日志，为出现故障后的恢复准备依据。

#### 两阶段提交的弊端

二阶段提交协议的存在的弊端是阻塞，因为事务管理器要收集各个资源管理器的响应消息，如果其中一个或多个一直不返回消息，则事务管理器一直等待，应用程序也被阻塞，甚至可能永久阻塞。