# 事务

# MySQL事务

### 一.什么是事务?

事务（Transaction）是数据库区别于文件系统的重要特性之一，事务会把数据库从一种一致性状态转换为另一种一致性状态。

在 MySQL 中只有使用了 Innodb 数据库引擎的数据库或表才支持事务。

事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。

事务用来管理 insert,update,delete 语句

### 二.事务的（ACID）特征

* 原子性(Atomicity)：整个事物的所有操作要么全部提交成功，要么全部失败回滚(不会出现部分执行的情况)。
* 一致性(Consistency)：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。
* 隔离性(Isolation)：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。
* 持久性(Durability): 一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

### 三.事务的隔离级别

**V表示会发生,X表示不会发生**



### 四.如何演示脏读,不可重复读,幻读

# 创建 demo01 数据库

create database demo01

use demo01

# 创建测试表

create table test01(

id int(3) not null primary key auto\_increment,

name varchar(64) default null,

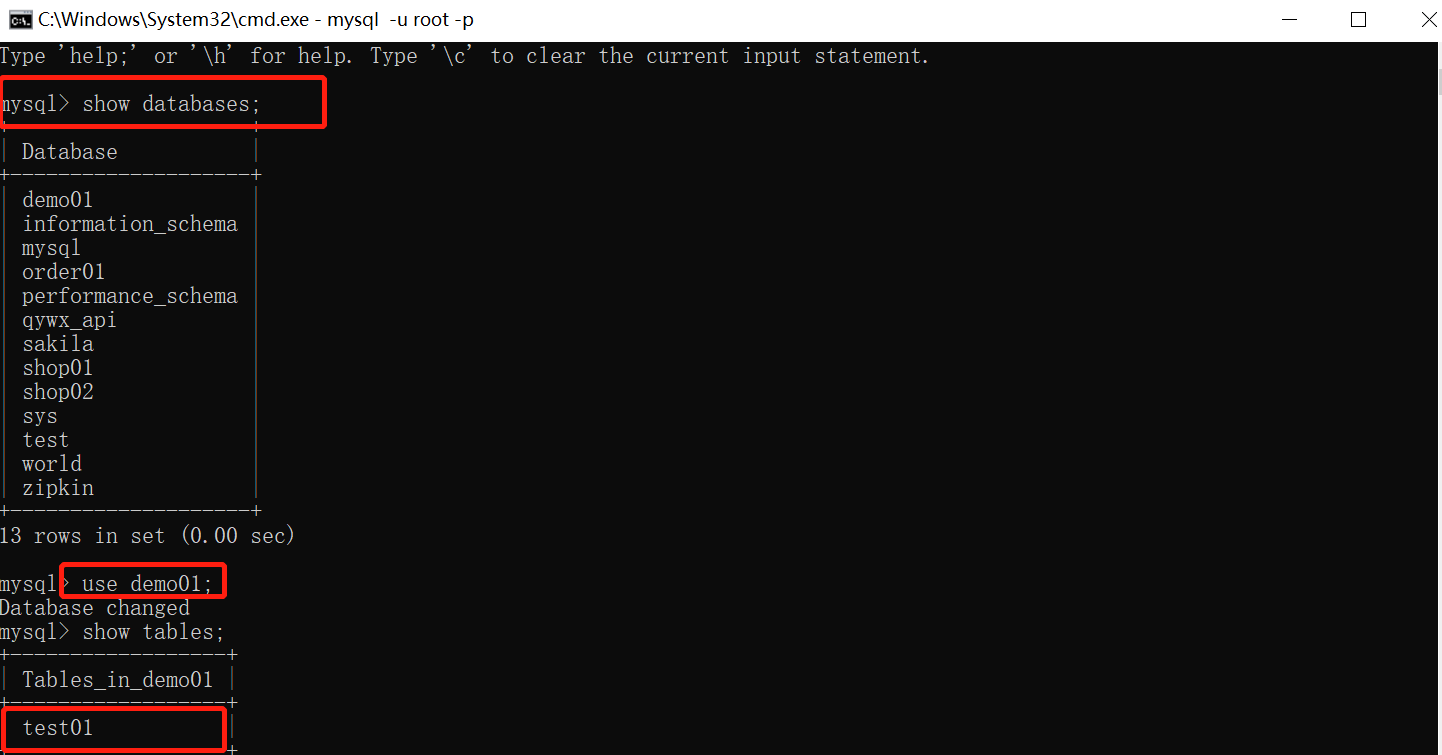
price int(7) default 0 # 这里不许有逗号

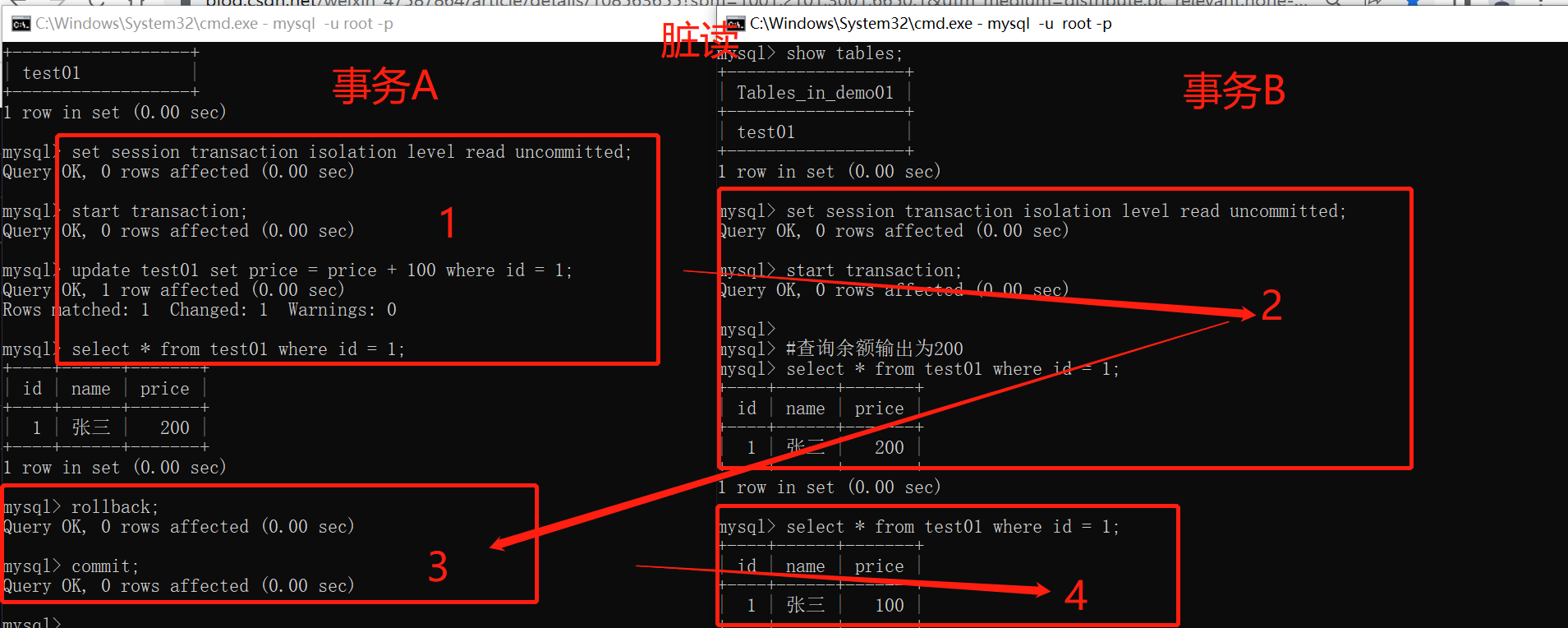
)ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8mb4;

insert into test01(name,price) values('张三',100),('李四',0);

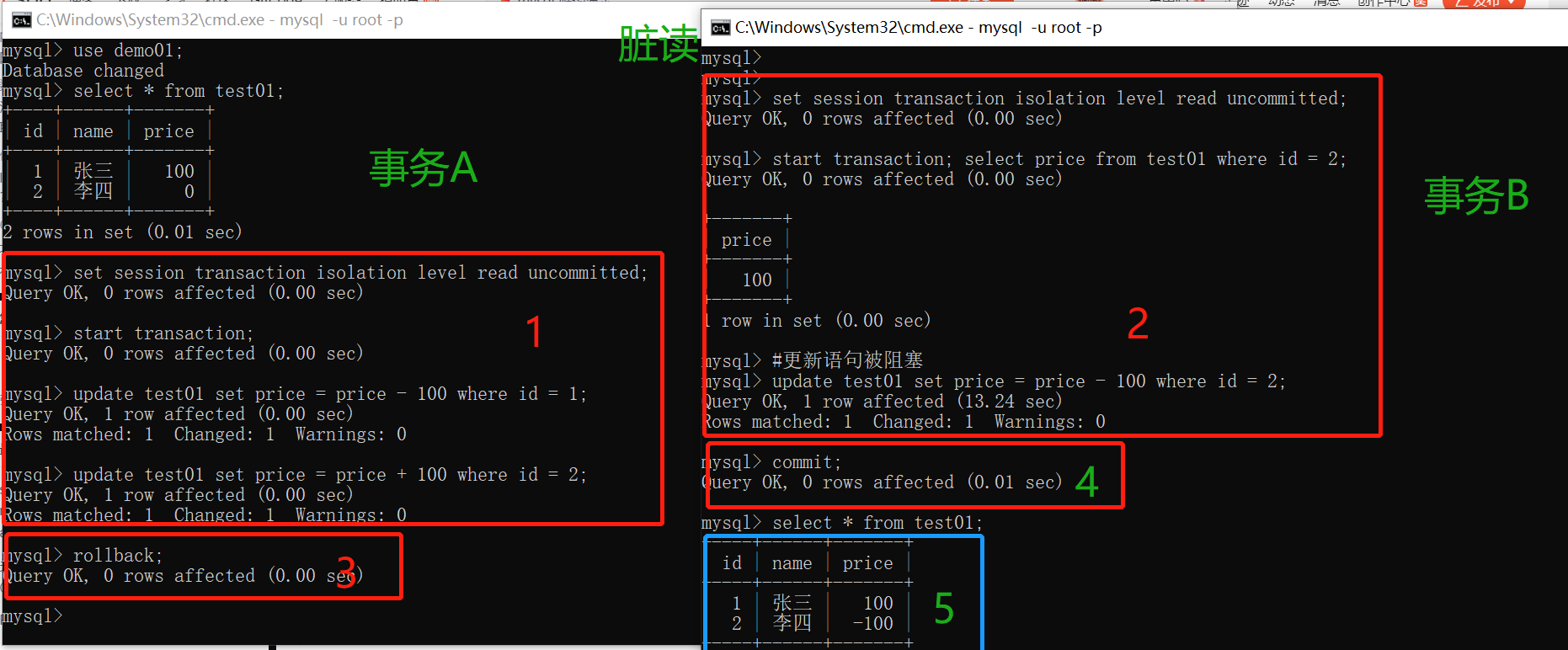
#### 4.1脏读

脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。





再举一个严重的例子，证明一下危害

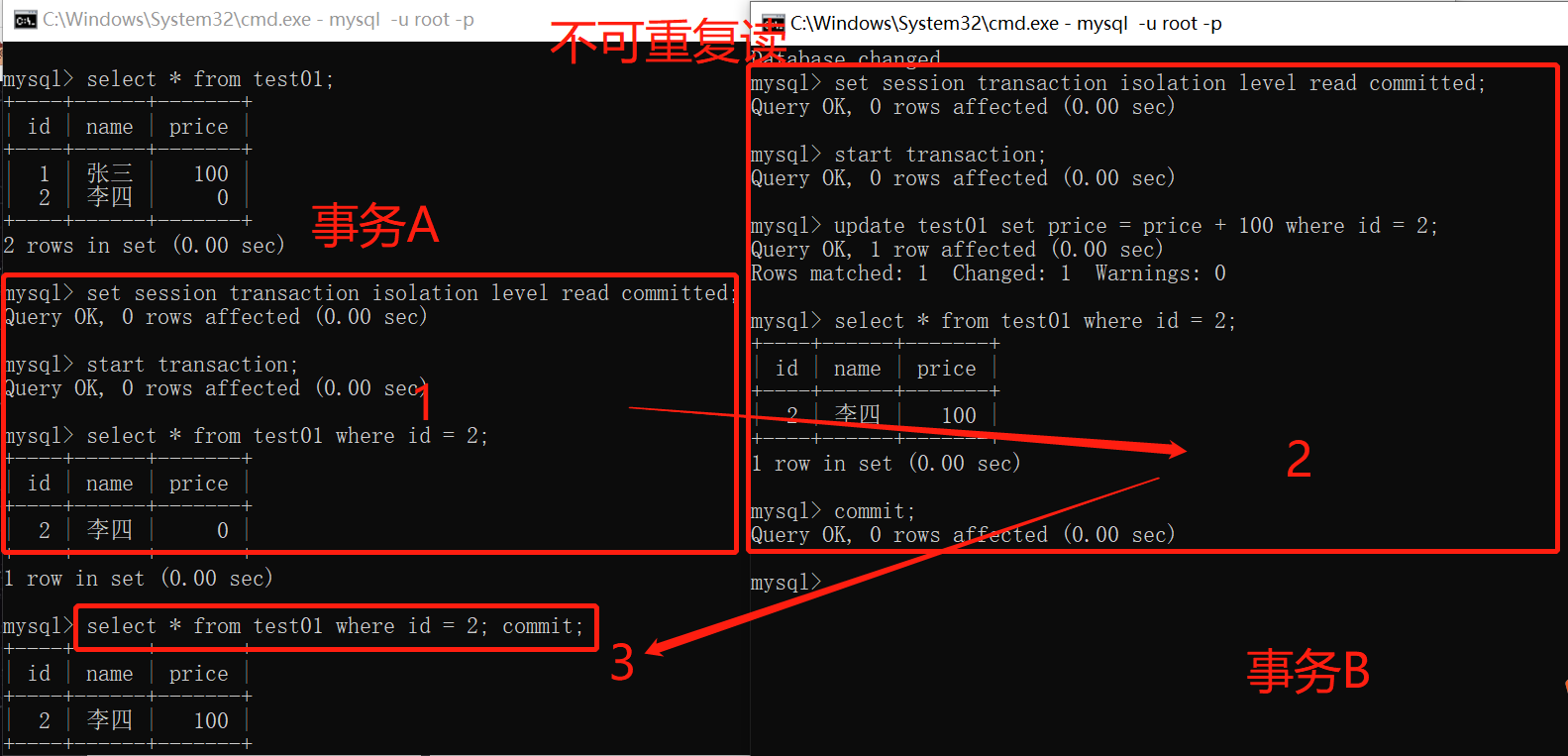


|  |  |
| --- | --- |
| **时间** | **解释** |
| T1 | 1给2转100 |
| T2 | 2的余额够100元，购买100元的东西，更新语句被阻塞 |
| T3 | 1回滚，1的余额变成100，2的余额变成0 |
| T4 | 2成功扣款，余额0-100=-100 |

这样无缘无故的银行少了100块钱

#### 4.2不可重复读

不可重复读是指在事务1内，读取了一个数据，事务1还没有结束时，事务2也访问了这个数据，修改了这个数据，并提交。紧接着，事务1又读这个数据。由于事务2的修改，那么事务1两次读到的的数据可能是不一样的，因此称为是不可重复读。



#### 4.3幻读

##### 4.3.1可重复读

当我们将当前会话的隔离级别设置为可重复读的时候，当前会话可以重复读，就是每次读取的结果集都相同，而不管其他事务有没有提交。

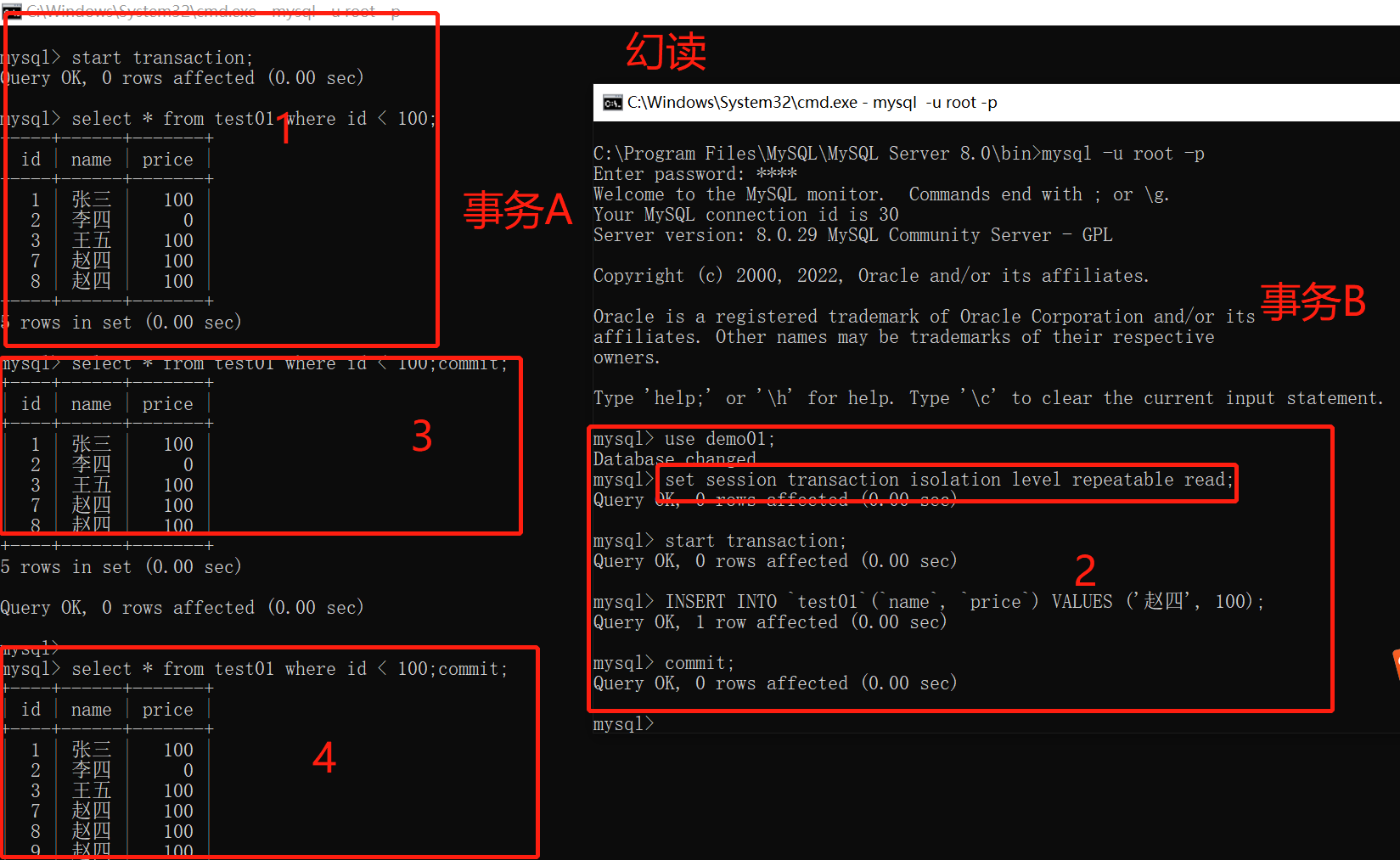


但是在可重复读的隔离级别上，会产生幻读的问题。

#### 4.3.2幻读

所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行。InnoDB存储引擎通过多版本并发控制（MVCC）解决了幻读的问题。

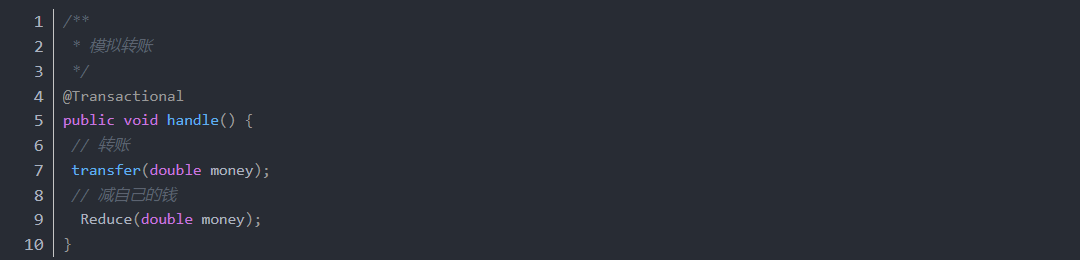
用大白话解释一下，就是事务1查询id<10的记录时，返回了2条记录，接着事务2插入了一条id为3的记录，并提交。接着事务1查询id<10的记录时，返回了3条记录，说好的可重复读呢？结果却多了一条数据。



##### 4.3.2.1如何解决幻读

# Spring事务

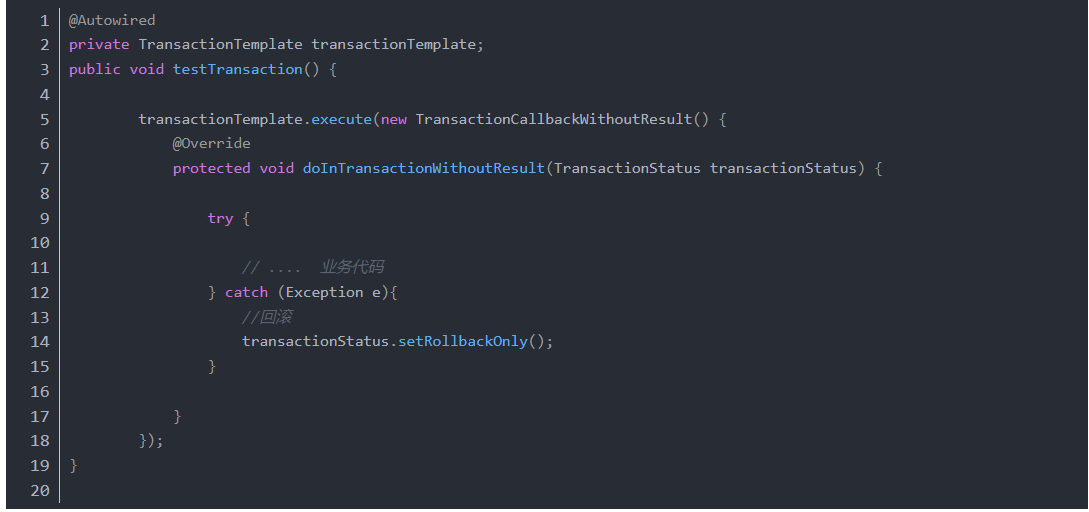
Spring 支持两种事务方式，分别是编程式事务和声明式事务，后者最常见，通常情况下只需要一个 \*\*@Transactional \*\*就搞定了（代码侵入性降到了最低），就像这样：



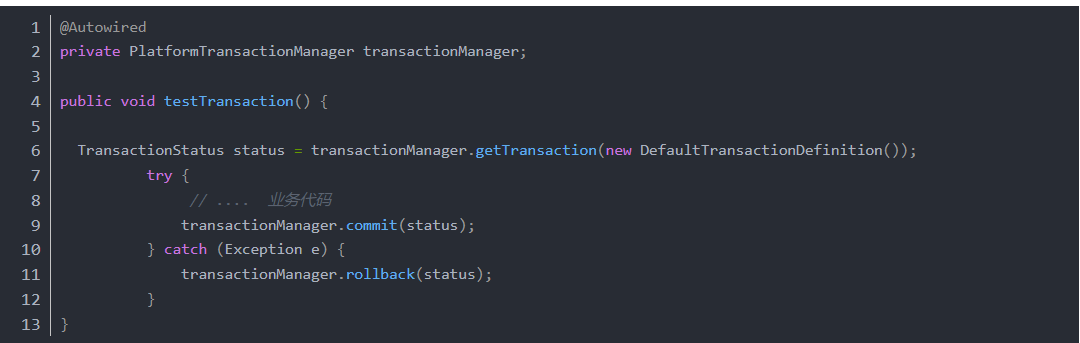
### 1.编程式事务

编程式事务是指将事务管理代码嵌入嵌入到业务代码中，来控制事务的提交和回滚。

你比如说，使用 TransactionTemplate 来管理事务：



再比如说，使用 TransactionManager 来管理事务：



就编程式事务管理而言，Spring 更推荐使用 TransactionTemplate。

在编程式事务中，必须在每个业务操作中包含额外的事务管理代码，就导致代码看起来非常的臃肿，但对理解 Spring 的事务管理模型非常有帮助。

### 2.声明式事务

声明式事务将事务管理代码从业务方法中抽离了出来，以声明式的方式来实现事务管理，对于开发者来说，声明式事务显然比编程式事务更易用、更好用。

当然了，要想实现事务管理和业务代码的抽离，就必须得用到 Spring 当中的AOP，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，执行完目标方法之后根据执行的情况提交或者回滚。

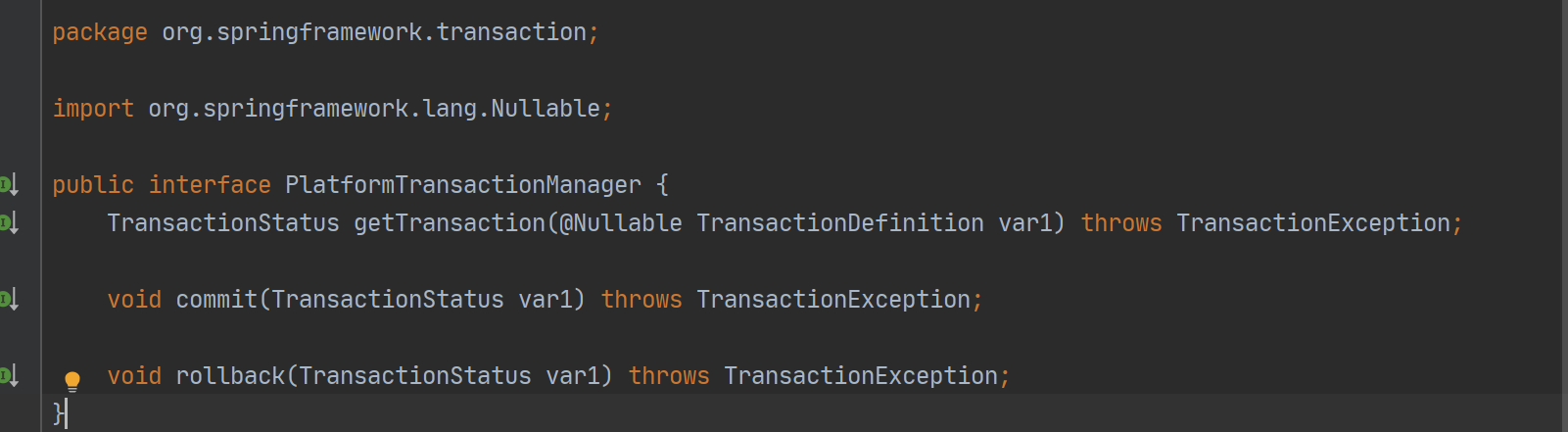
**声明式事务虽然优于编程式事务，但也有不足，声明式事务管理的粒度是方法级别，而编程式事务是可以精确到代码块级别的.**

### 事务管理模型

Spring 将事务管理的核心抽象为一个事务管理器（TransactionManager），它的源码只有一个简单的接口定义，属于一个标记接口：



该接口有两个子接口，分别是编程式事务接口 ReactiveTransactionManager 和声明式事务接口 PlatformTransactionManager。我们来重点说说 PlatformTransactionManager，该接口定义了 3 个接口方法：

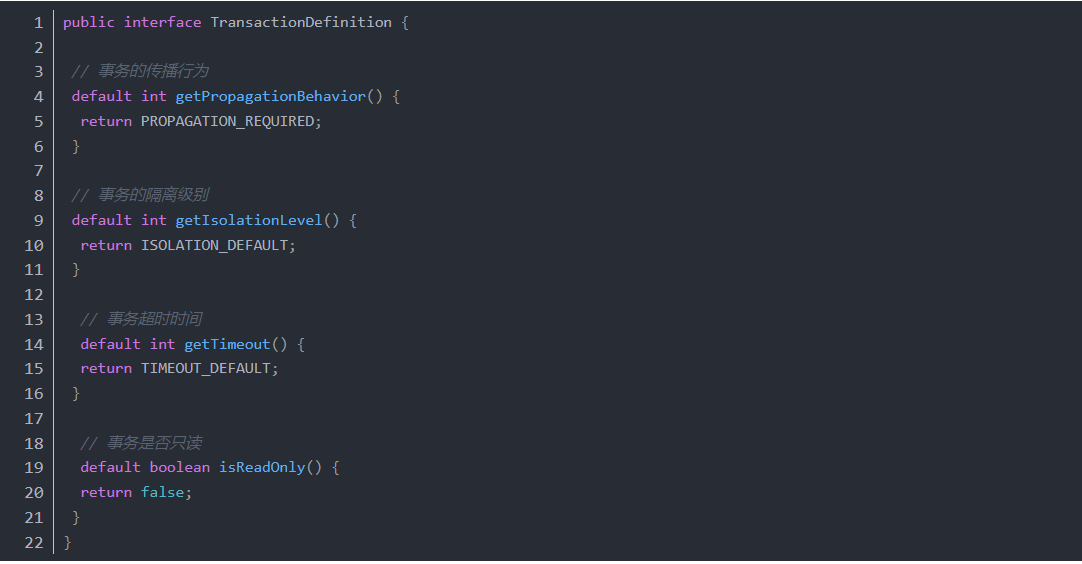


通过PlatformTransactionManager 这个接口，Spring 为各个平台如 JDBC(DataSourceTransactionManager)、Hibernate(HibernateTransactionManager)、JPA(JpaTransactionManager)等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

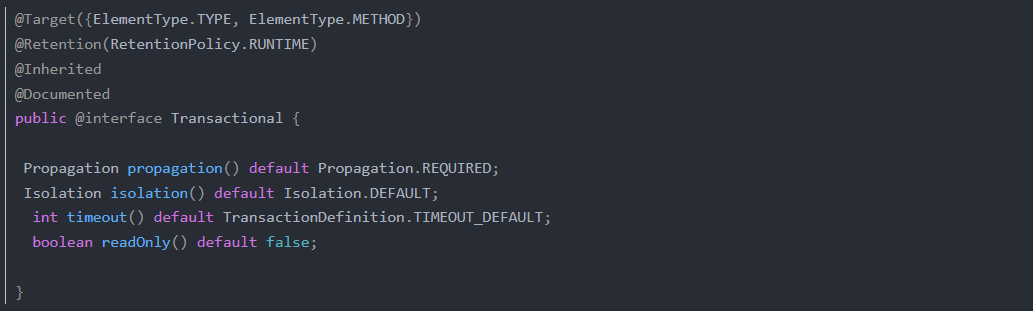
参数 TransactionDefinition 和 @Transactional 注解是对应的，比如说 @Transactional 注解中定义的事务传播行为、隔离级别、事务超时时间、事务是否只读等属性，在 TransactionDefinition 都可以找得到。

返回类型 TransactionStatus 主要用来存储当前事务的一些状态和数据，比如说事务资源（connection）、回滚状态等。

TransactionDefinition如下：



Transactional注解如下：



* @Transactional 注解中的 propagation 对应 TransactionDefinition 中的 getPropagationBehavior，默认值为 Propagation.REQUIRED(TransactionDefinition.PROPAGATION\_REQUIRED)
* @Transactional 注解中的 isolation 对应 TransactionDefinition 中的 getIsolationLevel，默认值为 DEFAULT(TransactionDefinition.ISOLATION\_DEFAULT).
* @Transactional 注解中的 timeout 对应 TransactionDefinition 中的 getTimeout，默认值为TransactionDefinition.TIMEOUT\_DEFAULT。
* @Transactional 注解中的 readOnly 对应 TransactionDefinition 中的 isReadOnly，默认值为 false。

说到这，我们来详细地说明一下 Spring 事务的传播行为、事务的隔离级别、事务的超时时间、事务的只读属性，以及事务的回滚规则。

### 事务传播行为

当事务方法被另外一个事务方法调用时，必须指定事务应该如何传播，例如，方法可能继续在当前事务中执行，也可以开启一个新的事务，在自己的事务中执行。

声明式事务的传播行为可以通过 @Transactional 注解中的 propagation 属性来定义，比如说：



TransactionDefinition 一共定义了 7 种事务传播行为，其中PROPAGATION\_REQUIRED、PROPAGATION\_REQUIRES\_NEW 两种传播行为是比较常用的。

* PROPAGATION\_REQUIRED

这也是 @Transactional 默认的事务传播行为，指的是如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。更确切地意思是：

如果外部方法没有开启事务的话，Propagation.REQUIRED 修饰的内部方法会开启自己的事务，且开启的事务相互独立，互不干扰。

如果外部方法开启事务并且是 Propagation.REQUIRED 的话，所有 Propagation.REQUIRED 修饰的内部方法和外部方法均属于同一事务 ，只要一个方法回滚，整个事务都需要回滚。

也就是说如果a方法和b方法都添加了注解，在默认传播模式下，a方法内部调用b方法，会把两个方法的事务合并为一个事务。

* PROPAGATION\_REQUIRES\_NEW

创建一个新的事务，如果当前存在事务，则把当前事务挂起。也就是说不管外部方法是否开启事务，Propagation.REQUIRES\_NEW 修饰的内部方法都会开启自己的事务，且开启的事务与外部的事务相互独立，互不干扰。

当类A中的 a 方法用默认 Propagation.REQUIRED模式，类B中的 b方法加上采用 Propagation.REQUIRES\_NEW模式，然后在 a 方法中调用 b方法操作数据库，然而 a方法抛出异常后，b方法并没有进行回滚，因为Propagation.REQUIRES\_NEW会暂停 a方法的事务 ，总结就是a不影响b，b影响a

* PROPAGATION\_NESTED

如果当前存在事务，就在当前事务内执行；否则，就执行与 PROPAGATION\_REQUIRED 类似的操作。

当类A中的 a 方法用默认 Propagation.REQUIRED模式，类B中的 b方法加上采用 Propagation.NESTED模式，然后在 在a 方法里调用 b方法操作数据库，然而 b方法抛出异常后，a方法是不的回滚 ，总结就是b不影响a，a影响b。

* PROPAGATION\_SUPPORTS

如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

* PROPAGATION\_NOT\_SUPPORTED

以非事务方式运行，如果当前存在事务，则把当前事务挂起。

* PROPAGATION\_MANDATORY

如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

* PROPAGATION\_NEVER

以非事务方式运行，如果当前存在事务，则抛出异常。

### 事务隔离级别

前面我们已经了解了数据库的事务隔离级别，再来理解 Spring 的事务隔离级别就容易多了。

TransactionDefinition 中一共定义了 5 种事务隔离级别：

* ISOLATION\_DEFAULT，使用数据库默认的隔离级别，MySql 默认采用的是 REPEATABLE\_READ，也就是可重复读。
* ISOLATION\_READ\_UNCOMMITTED，最低的隔离级别，可能会出现脏读、幻读或者不可重复读。
* ISOLATION\_READ\_COMMITTED，允许读取并发事务提交的数据，可以防止脏读，但幻读和不可重复读仍然有可能发生。
* ISOLATION\_REPEATABLE\_READ，对同一字段的多次读取结果都是一致的，除非数据是被自身事务所修改的，可以阻止脏读和不可重复读，但幻读仍有可能发生。
* ISOLATION\_SERIALIZABLE，最高的隔离级别，虽然可以阻止脏读、幻读和不可重复读，但会严重影响程序性能。

通常情况下，我们采用默认的隔离级别 ISOLATION\_DEFAULT 就可以了，也就是交给数据库来决定.

### 6.事务的超时时间

事务超时\*\*timeout \*\*，也就是指一个事务所允许执行的最长时间，如果在超时时间内还没有完成的话，就自动回滚。

假如事务的执行时间格外的长，由于事务涉及到对数据库的锁定，就会导致长时间运行的事务占用数据库资源。

### 7.事务的只读属性

事务的只读属性readOnly， 如果一个事务只是对数据库执行读操作，那么该数据库就可以利用事务的只读属性，采取优化措施，适用于多条数据库查询操作中。

为什么一个查询操作还要启用事务支持呢？

这是因为 MySql（innodb）默认对每一个连接都启用了 autocommit 模式，在该模式下，每一个发送到 MySql 服务器的 SQL 语句都会在一个单独的事务中进行处理，执行结束后会自动提交事务。

那如果我们给方法加上了 @Transactional 注解，那这个方法中所有的 SQL 都会放在一个事务里。否则，每条 SQL 都会单独开启一个事务，中间被其他事务修改了数据，都会实时读取到。

有些情况下，当一次执行多条查询语句时，需要保证数据一致性时，就需要启用事务支持。否则上一条 SQL 查询后，被其他用户改变了数据，那么下一个 SQL 查询可能就会出现不一致的状态。

### 8.事务的回滚策略

\*\*回滚策略rollbackFor \*\*，用于指定能够触发事务回滚的异常类型，可以指定多个异常类型。默认情况下，事务只在出现运行时异常（Runtime Exception）时回滚，以及 Error，出现检查异常（checked exception，需要主动捕获处理或者向上抛出）时不回滚。

如果你想要回滚特定的异常类型的话，可以这样设置：



### 9.事务的不回滚策略

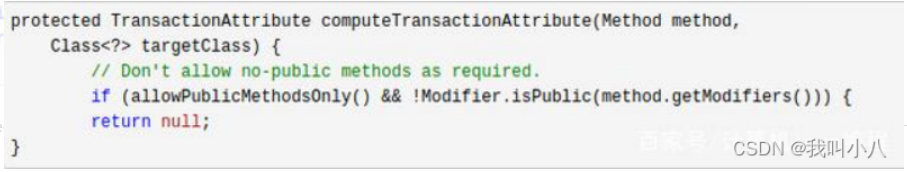
\*\*不回滚策略noRollbackFor \*\*，用于指定不触发事务回滚的异常类型，可以指定多个异常类型。

@Transaction失效场景

* 作用于非public方法上，之所以会失效是因为在Spring AOP 代理时，如下图所示 TransactionInterceptor （事务拦截器）在目标方法执行前后进行拦截，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的 intercept 方法或 JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource的 computeTransactionAttribute

方法，获取Transactional 注解的事务配置信息。

此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息。



注意：protected、private修饰的方法上使用 @Transactional 注解，虽然事务无效，但不会有任何报错，这是我们很容犯错的一点。

* propagation设置问题，会导致事务不生效，也就事务不会回滚
* rollbackFor指定事务回滚的异常类型
* 同个类中的调用被@transaction修饰的方法，会失效，因为只有当事务方法被当前类以外的代码调用，才会由spring生成的代理对象来管理。
* try catch导致失效
* 数据库不支持事务