一、事务简介

1.事务特性

事务在逻辑上是一组操作，要么执行，要不都不执行。主要是针对数据库而言的，比如说 MySQL。
为了保证事务是正确可靠的，在数据库进行写入或者更新操作时，就必须得表现出 ACID 的 4 个重要特性：

- 原子性（Atomicity）：一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- 一致性（Consistency）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。
- 事务隔离（Isolation）：数据库允许多个并发事务同时对其数据进行读写和修改，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
- 持久性（Durability）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

其中，事务隔离又分为 4 种不同的级别，包括：

- 未提交读（Read uncommitted），最低的隔离级别，允许“脏读”（dirty reads），事务可以看到其他事务“尚未提交”的修改。如果另一个事务回滚，那么当前事务读到的数据就是脏数据。
- 提交读（read committed），一个事务可能会遇到不可重复读（Non Repeatable Read）的问题。不可重复读是指，在一个事务内，多次读同一数据，在这个事务还没有结束时，如果另一个事务恰好修改了这个数据，那么，在第一个事务中，两次读取的数据就可能不一致。
- 可重复读（repeatable read），一个事务可能会遇到幻读（Phantom Read）的问题。幻读是指，在一个事务中，第一次查询某条记录，发现没有，但是，当试图更新这条不存在的记录时，竟然能成功，并且，再次读取同一条记录，它就神奇地出现了。
- 串行化（Serializable），最严格的隔离级别，所有事务按照次序依次执行，因此，脏读、不可重复读、幻读都不会出现。虽然 Serializable 隔离级别下的事务具有最高的安全性，但是，由于事务是串行执行，所以效率会大大下降，应用程序的性能会急剧降低。如果没有特别重要的情景，一般都不会使用 Serializable 隔离级别。

需要格外注意的是：事务能否生效，取决于数据库引擎是否支持事务，MySQL 的 InnoDB 引擎是支持事务的，但 MyISAM 就不支持。

二、spring的事务支持

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：

```
/**
* 模拟转账
*/
@Transactional
public void handle() {
// 转账
transfer(double money);
// 减自己的钱
Reduce(double money);
}
```

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：

```
@Autowired
private TransactionTemplate transactionTemplate;
public void testTransaction() {

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus
transactionStatus) {

                try {

                    // ....  业务代码
                } catch (Exception e){
                    //回滚
                    transactionStatus.setRollbackOnly();
                }

            }
        });

}
```

再比如说，使用 TransactionManager 来管理事务：

```
@Autowired
private PlatformTransactionManager transactionManager;

public void testTransaction() {

  TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
          try {
               // ....  业务代码
              transactionManager.commit(status);
          } catch (Exception e) {
              transactionManager.rollback(status);
          }
}
```

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。

3.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：

```
public interface TransactionManager {

}
```

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：

```
interface PlatformTransactionManager extends TransactionManager{
    // 根据事务定义获取事务状态
    TransactionStatus getTransaction(TransactionDefinition definition)
            throws TransactionException;

    // 提交事务
    void commit(TransactionStatus status) throws TransactionException;

    // 事务回滚
    void rollback(TransactionStatus status) throws TransactionException;

}
```

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：

```
public interface TransactionDefinition {

 // 事务的传播行为
 default int getPropagationBehavior() {
  return PROPAGATION_REQUIRED;
```

```
 }

 // 事务的隔离级别
 default int getIsolationLevel() {
  return ISOLATION_DEFAULT;
 }

  // 事务超时时间
  default int getTimeout() {
  return TIMEOUT_DEFAULT;
 }

  // 事务是否只读
  default boolean isReadOnly() {
  return false;
 }
}
```

Transactional注解如下：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {

 Propagation propagation() default Propagation.REQUIRED;
 Isolation isolation() default Isolation.DEFAULT;
  int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
  boolean readOnly() default false;

}
```

- @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。

4.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：

```
@Transactional(propagation = Propagation.REQUIRED)
public void savePosts(PostsParam postsParam) {
}
```

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方法是不会回滚 ，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。

5.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

```
@Transactional(readOnly=true)
```

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。