有道云链接：http://note.youdao.com/noteshare?id=73adfd2c0d72e9be2f8f613a75008f71&sub=75CD877539EA4470960CCD82C2077264

@EnableTransactionManagement工作原理

开启Spring事务本质上就是增加了一个Advisor，但我们使用@EnableTransactionManagement注解来开启Spring事务是，该注解代理的功能就是向Spring容器中添加了两个Bean：

1. AutoProxyRegistrar
2. ProxyTransactionManagementConfiguration

AutoProxyRegistrar主要的作用是向Spring容器中注册了一个**InfrastructureAdvisorAutoProxyCreator**的Bean。 而InfrastructureAdvisorAutoProxyCreator继承了**AbstractAdvisorAutoProxyCreator**，所以这个类的主要作用就是**开启自动代理**的作用，也就是一个BeanPostProcessor，会在初始化后步骤中去寻找Advisor类型的Bean，并判断当前某个Bean是否有匹配的Advisor，是否需要利用动态代理产生一个代理对象。

ProxyTransactionManagementConfiguration是一个配置类，它又定义了另外三个bean：

1. BeanFactoryTransactionAttributeSourceAdvisor：一个Advisor
2. AnnotationTransactionAttributeSource：相当于BeanFactoryTransactionAttributeSourceAdvisor中的Pointcut
3. TransactionInterceptor：相当于BeanFactoryTransactionAttributeSourceAdvisor中的Advice

AnnotationTransactionAttributeSource就是用来判断某个类上是否存在@Transactional注解，或者判断某个方法上是否存在@Transactional注解的。

TransactionInterceptor就是代理逻辑，当某个类中存在@Transactional注解时，到时就产生一个代理对象作为Bean，代理对象在执行某个方法时，最终就会进入到TransactionInterceptor的invoke()方法。

Spring事务基本执行原理

一个Bean在执行Bean的创建生命周期时，会经过InfrastructureAdvisorAutoProxyCreator的初始化后的方法，会判断当前当前Bean对象是否和BeanFactoryTransactionAttributeSourceAdvisor匹配，匹配逻辑为判断该Bean的类上是否存在@Transactional注解，或者类中的某个方法上是否存在@Transactional注解，如果存在则表示该Bean需要进行动态代理产生一个代理对象作为Bean对象。

该代理对象在执行某个方法时，会再次判断当前执行的方法是否和BeanFactoryTransactionAttributeSourceAdvisor匹配，如果匹配则执行该Advisor中的TransactionInterceptor的invoke()方法，执行基本流程为：

1. 利用所配置的PlatformTransactionManager事务管理器新建一个数据库连接
2. 修改数据库连接的autocommit为false
3. 执行MethodInvocation.proceed()方法，简单理解就是执行业务方法，其中就会执行sql
4. 如果没有抛异常，则提交
5. 如果抛了异常，则回滚

Spring事务详细执行流程

Spring事务执行流程图：https://www.processon.com/view/link/5fab6edf1e0853569633cc06

Spring事务传播机制

在开发过程中，经常会出现一个方法调用另外一个方法，那么这里就涉及到了多种场景，比如a()调用b()：

1. a()和b()方法中的所有sql需要在同一个事务中吗?
2. a()和b()方法需要单独的事务吗?
3. a()需要在事务中执行，b()还需要在事务中执行吗?
4. 等等情况...

所以，这就要求Spring事务能支持上面各种场景，这就是Spring事务传播机制的由来。那Spring事务传播机制是如何实现的呢?

先来看上述几种场景中的一种情况，a()在一个事务中执行，调用b()方法时需要新开一个事务执行：

1. 首先，代理对象执行a()方法前，先利用事务管理器新建一个数据库连接a
2. 将数据库连接a的autocommit改为false
3. 把数据库连接a设置到ThreadLocal中
4. 执行a()方法中的sql
5. 执行a()方法过程中，调用了b()方法（注意用代理对象调用b()方法）
    i. 代理对象执行b()方法前，判断出来了当前线程中已经存在一个数据库连接a了，表示当前线程其实已经拥有一个Spring事务了，则进行**挂起**
    ii. 挂起就是把ThreadLocal中的数据库连接a从ThreadLocal中移除，并放入一个**挂起资源对象**中
    iii. 挂起完成后，再次利用事务管理器新建一个数据库连接b
    iv. 将数据库连接b的autocommit改为false
    v. 把数据库连接b设置到ThreadLocal中
    vi. 执行b()方法中的sql
    vii. b()方法正常执行完，则从ThreadLocal中拿到数据库连接b进行提交
    viii. 提交之后会恢复所挂起的数据库连接a，这里的恢复，其实只是把在**挂起资源对象**中所保存的数据库连接a再次设置到ThreadLocal中

6. a()方法正常执行完，则从ThreadLocal中拿到数据库连接a进行提交

这个过程中最为核心的是：**在执行某个方法时，判断当前是否已经存在一个事务，就是判断当前线程的ThreadLocal中是否存在一个数据库连接对象，如果存在则表示已经存在一个事务了。**

Spring事务传播机制分类

其中，以非事务方式运行，表示以非Spring事务运行，表示在执行这个方法时，Spring事务管理器不会去建立数据库连接，执行sql时，由Mybatis或JdbcTemplate自己来建立数据库连接来执行sql。

案例分析

情况1

```
@Component
public class UserService {
    @Autowired
    private UserService userService;

    @Transactional
    public void test() {
        // test方法中的sql
        userService.a();
    }

    @Transactional
    public void a() {
        // a方法中的sql
    }
}
```

默认情况下传播机制为**REQUIRED，表示当前如果没有事务则新建一个事务，如果有事务则在当前事务中执行。**

所以上面这种情况的执行流程如下：

1. 新建一个数据库连接conn
2. 设置conn的autocommit为false
3. 执行test方法中的sql
4. 执行a方法中的sql
5. 执行conn的commit()方法进行提交

情况2

假如是这种情况

```
@Component
public class UserService {
    @Autowired
    private UserService userService;

    @Transactional
    public void test() {
        // test方法中的sql
        userService.a();
        int result = 100/0;
    }

    @Transactional
    public void a() {
        // a方法中的sql
    }
}
```

所以上面这种情况的执行流程如下：

1. 新建一个数据库连接conn
2. 设置conn的autocommit为false
3. 执行test方法中的sql
4. 执行a方法中的sql
5. 抛出异常
6. 执行conn的rollback()方法进行回滚，所以两个方法中的sql都会回滚掉

情况3

假如是这种情况：

```
@Component
public class UserService {
    @Autowired
    private UserService userService;

    @Transactional
    public void test() {
        // test方法中的sql
        userService.a();
    }

    @Transactional
    public void a() {
        // a方法中的sql
        int result = 100/0;
    }
}
```

所以上面这种情况的执行流程如下：

1. 新建一个数据库连接conn
2. 设置conn的autocommit为false
3. 执行test方法中的sql
4. 执行a方法中的sql
5. 抛出异常
6. 执行conn的rollback()方法进行回滚，所以两个方法中的sql都会回滚掉

情况4

如果是这种情况：

```
@Component
public class UserService {
    @Autowired
    private UserService userService;

    @Transactional
    public void test() {
        // test方法中的sql
        userService.a();
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void a() {
        // a方法中的sql
        int result = 100/0;
    }
}
```

所以上面这种情况的执行流程如下：

1. 新建一个数据库连接conn
2. 设置conn的autocommit为false
3. 执行test方法中的sql
4. 又新建一个数据库连接conn2
5. 执行a方法中的sql
6. 抛出异常
7. 执行conn2的rollback()方法进行回滚
8. **继续抛异常，对于test()方法而言，它会接收到一个异常，然后抛出**
9. 执行conn的rollback()方法进行回滚，最终还是两个方法中的sql都回滚了

Spring事务强制回滚

正常情况下，a()调用b()方法时，如果b()方法抛了异常，但是在a()方法捕获了，那么a()的事务还是会正常提交的，但是有的时候，我们捕获异常可能仅仅只是不把异常信息返回给客户端，而是为了返回一些更友好的错误信息，而这个时候，我们还是希望事务能回滚的，那这个时候就得告诉Spring把当前事务回滚掉，做法就是：

```
@Transactional
public void test(){

    // 执行sql
    try {
        b();
    } catch (Exception e) {
        // 构造友好的错误信息返回
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }

}

public void b() throws Exception {
    throw new Exception();
}
```

TransactionSynchronization

Spring事务有可能会提交，回滚、挂起、恢复，所以Spring事务提供了一种机制，可以让程序员来监听当前Spring事务所处于的状态。

```
@Component
public class UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Autowired
    private UserService userService;

    @Transactional
    public void test(){
        TransactionSynchronizationManager.registerSynchronization(new
TransactionSynchronization() {

            @Override
            public void suspend() {
                System.out.println("test被挂起了");
            }

            @Override
            public void resume() {
                System.out.println("test被恢复了");
            }

            @Override
```

```
            public void beforeCommit(boolean readOnly) {
                System.out.println("test准备要提交了");
            }

            @Override
            public void beforeCompletion() {
                System.out.println("test准备要提交或回滚了");
            }

            @Override
            public void afterCommit() {
                System.out.println("test提交成功了");
            }

            @Override
            public void afterCompletion(int status) {
                System.out.println("test提交或回滚成功了");
            }
        });

        jdbcTemplate.execute("insert into t1 values(1,1,1,1,'1')");
        System.out.println("test");
        userService.a();
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void a(){
        TransactionSynchronizationManager.registerSynchronization(new
TransactionSynchronization() {

            @Override
            public void suspend() {
                System.out.println("a被挂起了");
            }

            @Override
            public void resume() {
                System.out.println("a被恢复了");
            }

            @Override
            public void beforeCommit(boolean readOnly) {
                System.out.println("a准备要提交了");
            }

            @Override
            public void beforeCompletion() {
                System.out.println("a准备要提交或回滚了");
            }

            @Override
            public void afterCommit() {
                System.out.println("a提交成功了");
            }

            @Override
            public void afterCompletion(int status) {
```

```
                System.out.println("a提交或回滚成功了");
            }
        });

        jdbcTemplate.execute("insert into t1 values(2,2,2,2,'2')");
        System.out.println("a");
    }


}
```