12. Spring事务和事务传播机制

本节目标

1. 掌握Spring事务的实现方式
2. 掌握事务的传播机制

1. 事务回顾

在数据库阶段, 我们已经学习过事务了.

1.1 什么是事务?

事务是一组操作的集合, 是一个不可分割的操作.

事务会把所有的操作作为一个整体, 一起向数据库提交或者是撤销操作请求. 所以这组操作要么同时成功, 要么同时失败.

1.2 为什么需要事务?

我们在进行程序开发时, 也会有事务的需求.

比如转账操作:

第一步：A 账户 -100 元.
第二步：B 账户 +100 元.

如果没有事务，第一步执行成功了, 第二步执行失败了, 那么A 账户的100 元就平白无故消失了. 如果使用事务就可以解决这个问题, 让这一组操作要么一起成功, 要么一起失败.

比如秒杀系统,

第一步: 下单成功

第二步: 扣减库存

下单成功后, 库存也需要同步减少. 如果下单成功, 库存扣减失败, 那么就会造成下单超出的情况. 所以就需要把这两步操作放在同一个事务中. 要么一起成功, 要么一起失败.

> 理解事务概念为主, 实际企业开发时, 并不是简单的通过事务来处理.

1.3 事务的操作

事务的操作主要有三步:

1. 开启事start transaction/ begin (一组操作前开启事务)
2. 提交事务: commit (这组操作全部成功, 提交事务)
3. 回滚事务: rollback (这组操作中间任何一个操作出现异常, 回滚事务)

```
-- 开启事务
start transaction;

-- 提交事务
commit;

-- 回滚事务
rollback;
```

2. Spring 中事务的实现

前面课程我们讲了MySQL的事务操作, Spring对事务也进行了实现.

Spring 中的事务操作分为两类:

1. 编程式事务(手动写代码操作事务).
2. 声明式事务(利用注解自动开启和提交事务).

在学习事务之前, 我们先准备数据和数据的访问代码

需求: 用户注册, 注册时在日志表中插入一条操作记录.

数据准备:

```
-- 创建数据库
DROP DATABASE IF EXISTS trans_test;

CREATE DATABASE trans_test DEFAULT CHARACTER SET utf8mb4;

-- 用户表
DROP TABLE IF EXISTS user_info;
CREATE TABLE user_info (
                `id` INT NOT NULL AUTO_INCREMENT,
```

```
                `user_name` VARCHAR (128) NOT NULL,
                `password` VARCHAR (128) NOT NULL,
                `create_time` DATETIME DEFAULT now(),
                `update_time` DATETIME DEFAULT now() ON UPDATE now(),
                PRIMARY KEY (`id`)
) ENGINE = INNODB DEFAULT CHARACTER
SET = utf8mb4 COMMENT = '用户表';

-- 操作日志表
DROP TABLE IF EXISTS log_info;
CREATE TABLE log_info (
        `id` INT PRIMARY KEY auto_increment,
        `user_name` VARCHAR ( 128 ) NOT NULL,
        `op` VARCHAR ( 256 ) NOT NULL,
        `create_time` DATETIME DEFAULT now(),
        `update_time` DATETIME DEFAULT now() ON UPDATE now()
) DEFAULT charset 'utf8mb4';
```

代码准备:

1. 创建项目 spring-trans, 引入Spring Web, Mybatis, mysql等依赖
2. 配置文件

```
spring:
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/trans_test?
characterEncoding=utf8&useSSL=false
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  configuration: # 配置打印 MyBatis日志
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
    map-underscore-to-camel-case: true #配置驼峰自动转换
```

3. 实体类

```
import lombok.Data;
import java.util.Date;

@Data
public class UserInfo {
    private Integer id;
```

```
    private String userName;
    private String password;
    private Date createTime;
    private Date updateTime;
}
```

```
import lombok.Data;
import java.util.Date;

@Data
public class LogInfo {
    private Integer id;
    private String userName;
    private String op;
    private Date createTime;
    private Date updateTime;
}
```

4. Mapper

```
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface UserInfoMapper {
    @Insert("insert into user_info(`user_name`,`password`)values(#{name},#{password})")
    Integer insert(String name,String password);
}
```

```
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface LogInfoMapper {
    @Insert("insert into log_info(`user_name`,`op`)values(#{name},#{op})")
    Integer insertLog(String name,String op);
}
```

5. Service

```java
@Slf4j
@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;

    public void registryUser(String name,String password){
        //插入用户信息
        userInfoMapper.insert(name,password);
    }
}
```

```java
@Slf4j
@Service
public class LogService {
    @Autowired
    private LogInfoMapper logInfoMapper;

    public void insertLog(String name,String op){
        //记录用户操作
        logInfoMapper.insertLog(name,"用户注册");
    }
}
```

6. Contrller

```java
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/user")
@RestController
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/registry")
    public String registry(String name,String password){
       //用户注册
       userService.registryUser(name,password);
       return "注册成功";
```

```
    }
}
```

2.1 Spring 编程式事务(了解)

Spring 手动操作事务和上面 MySQL 操作事务类似, 有 3 个重要操作步骤:

- 开启事务(获取事务)
- 提交事务
- 回滚事务

SpringBoot 内置了两个对象:

1. `DataSourceTransactionManager` 事务管理器. 用来获取事务(开启事务), 提交或回滚事务的
2. `TransactionDefinition` 是事务的属性, 在获取事务的时候需要将 `TransactionDefinition` 传递进去从而获得一个事务 `TransactionStatus`

我们还是根据代码的实现来学习:

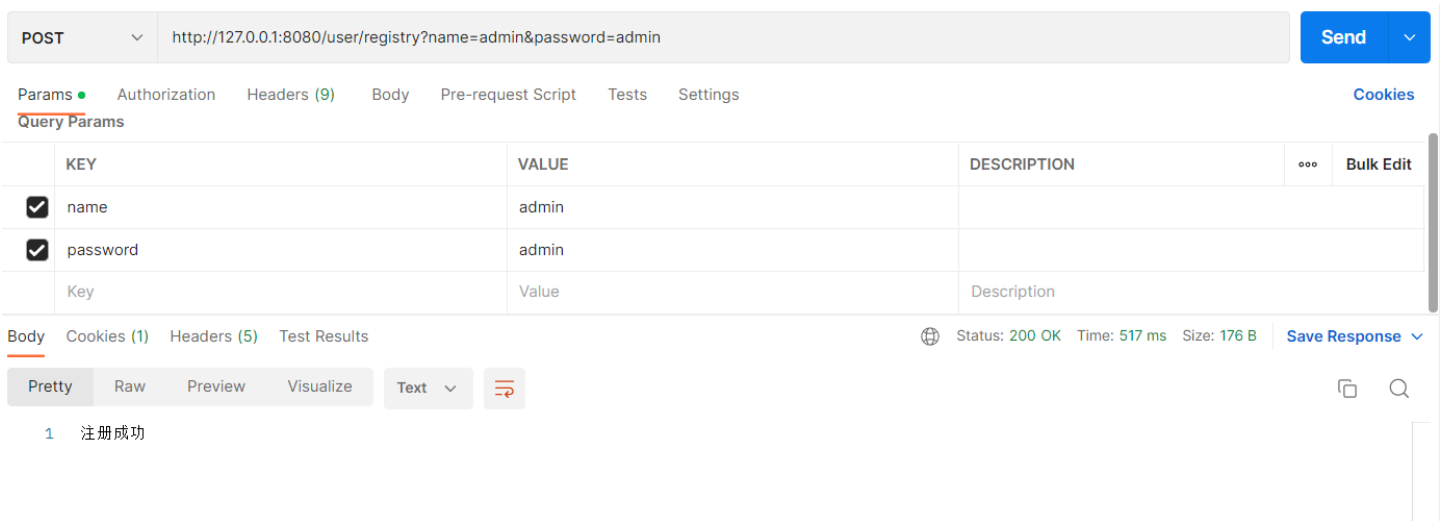
```java
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/user")
@RestController
public class UserController {
    // JDBC 事务管理器
    @Autowired
    private DataSourceTransactionManager dataSourceTransactionManager;
    // 定义事务属性
    @Autowired
    private TransactionDefinition transactionDefinition;

    @Autowired
    private UserService userService;

    @RequestMapping("/registry")
    public String registry(String name,String password){
        // 开启事务
        TransactionStatus transactionStatus = dataSourceTransactionManager
```

```
26                  .getTransaction(transactionDefinition);
27          //用户注册
28          userService.registryUser(name,password);
29          //提交事务
30          dataSourceTransactionManager.commit(transactionStatus);
31          //回滚事务
32          //dataSourceTransactionManager.rollback(transactionStatus);
33          return "注册成功";
34      }
35  }
```

观察事务提交

```
1  //提交事务
2  dataSourceTransactionManager.commit(transactionStatus);
```

运行程序: http://127.0.0.1:8080/user/registry?name=admin&password=admin

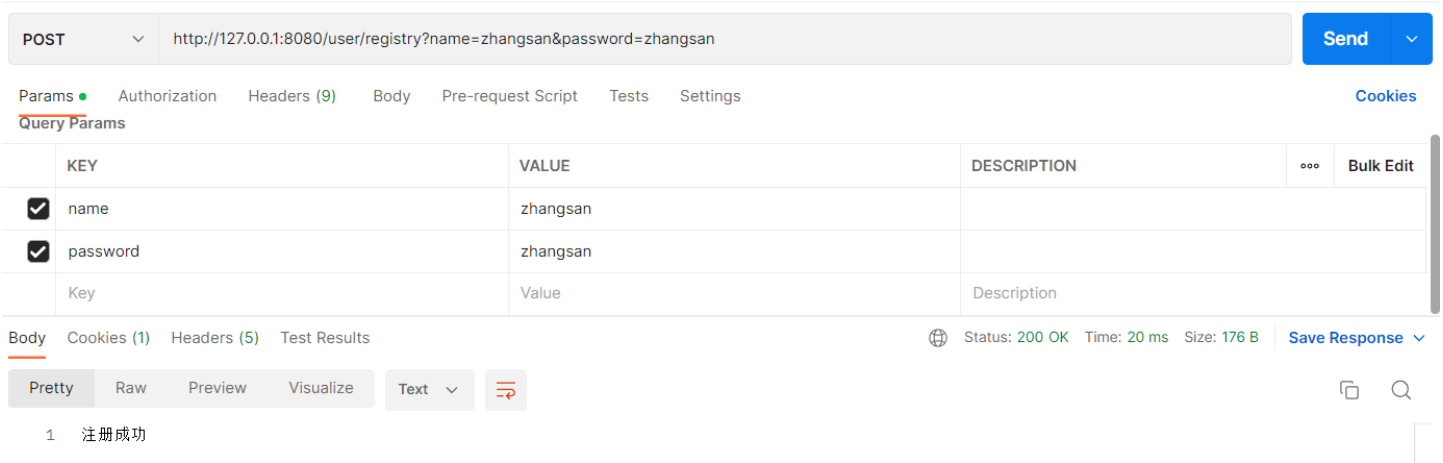


观察数据库的结果, 数据插入成功.

观察事务回滚

```
1  //回滚事务
2  dataSourceTransactionManager.rollback(transactionStatus);
```

运行程序:



观察数据库, 虽然程序返回"注册成功", 但数据库并没有新增数据.

以上代码虽然可以实现事务, 但操作也很繁琐, 有没有更简单的实现方法呢?

接下来我们学习声明式事务

2.2 Spring 声明式事务 @Transactional

声明式事务的实现很简单, 只需要在需要事务的方法上添加 `@Transactional` 注解就可以实现了. 无需手动开启事务和提交事务, 进入方法时自动开启事务, 方法执行完会自动提交事务, 如果中途发生了没有处理的异常会自动回滚事务.

我们来看代码实现:

```
@RequestMapping("/trans")
@RestController
public class TransactionalController {
    @Autowired
    private UserService userService;

    @Transactional
    @RequestMapping("/registry")
    public String registry(String name,String password){
        //用户注册
        userService.registryUser(name,password);
        return "注册成功";
    }
}
```

运行程序, 发现数据插入成功.

修改程序, 使之出现异常

```java
@Slf4j
@RequestMapping("/trans")
@RestController
public class TransactionalController {
    @Autowired
    private UserService userService;

    @Transactional
    @RequestMapping("/registry")
    public String registry(String name,String password){
        //用户注册
        userService.registryUser(name,password);
        log.info("用户数据插入成功");
        //强制程序抛出异常
        int a = 10/0;
        return "注册成功";
    }
}
```

运行程序:

发现虽然日志显示数据插入成功, 但数据库却没有新增数据, 事务进行了回滚.

```
Releasing transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1c7c0ee3]
16:47:09.938 com.example.demo.service.UserService registryUser [http-nio-8080-exec-1] 用户表数据插入成功
Transaction synchronization deregistering SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1c7c0ee3]
Transaction synchronization closing SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1c7c0ee3]
16:47:09.951 org.apache.catalina.core.ContainerBase.[Tomcat].[localhost].[/].[dispatcherServlet] log [http-nio-8080-exec-1] Servlet.service() for servlet
 [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is java.lang.ArithmeticException: / by zero] with root cause
java.lang.ArithmeticException Create breakpoint : / by zero
```

> 我们一般会在业务逻辑层当中来控制事务, 因为在业务逻辑层当中, 一个业务功能可能会包含多个数据访问的操作. 在业务逻辑层来控制事务, 我们就可以将多个数据访问操作控制在一个事务范围内.
>
> 上述代码在Controller中书写, 只是为了方便学习.

@Transactional 作用

`@Transactional` 可以用来修饰方法或类:

- 修饰方法时: 只有修饰**public** 方法时才生效(修饰其他方法时不会报错, 也不生效)**[推荐]**
- 修饰类时: 对 `@Transactional` 修饰的类中所有的 public 方法都生效.

方法/类被 `@Transactional` 注解修饰时, 在目标方法执行开始之前, 会自动开启事务, 方法执行结束之后, 自动提交事务.

如果在方法执行过程中, 出现异常, 且异常未被捕获, 就进行事务回滚操作.

如果异常被程序捕获, 方法就被认为是成功执行, 依然会提交事务.

修改上述代码, 对异常进行捕获

```
@Transactional
@RequestMapping("/registry")
public String registry(String name,String password){
    //用户注册
    userService.registryUser(name,password);
    log.info("用户数据插入成功");
    //对异常进行捕获
    try {
        //强制程序抛出异常
        int a = 10/0;
    }catch (Exception e){
        e.printStackTrace();
    }
    return "注册成功";
}
```

运行程序, 发现虽然程序出错了, 但是由于异常被捕获了, 所以事务依然得到了提交.

如果需要事务进行回滚, 有以下两种方式:

1. 重新抛出异常

```
@Transactional
@RequestMapping("/registry")
public String registry(String name,String password){
    //用户注册
    userService.registryUser(name,password);
    log.info("用户数据插入成功");
    //对异常进行捕获
    try {
        //强制程序抛出异常
        int a = 10/0;
    }catch (Exception e){
        //将异常重新抛出去
        throw e;
    }
    return "注册成功";
}
```

2. 手动回滚事务

使用 `TransactionAspectSupport.currentTransactionStatus()` 得到当前的事务, 并使用 `setRollbackOnly` 设置 `setRollbackOnly`

```
@Transactional
@RequestMapping("/registry")
public String registry(String name,String password){
    //用户注册
    userService.registryUser(name,password);
    log.info("用户数据插入成功");
    //对异常进行捕获
    try {
        //强制程序抛出异常
        int a = 10/0;
    }catch (Exception e){
        // 手动回滚事务
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
    return "注册成功";
}
```

3. @Transactional 详解

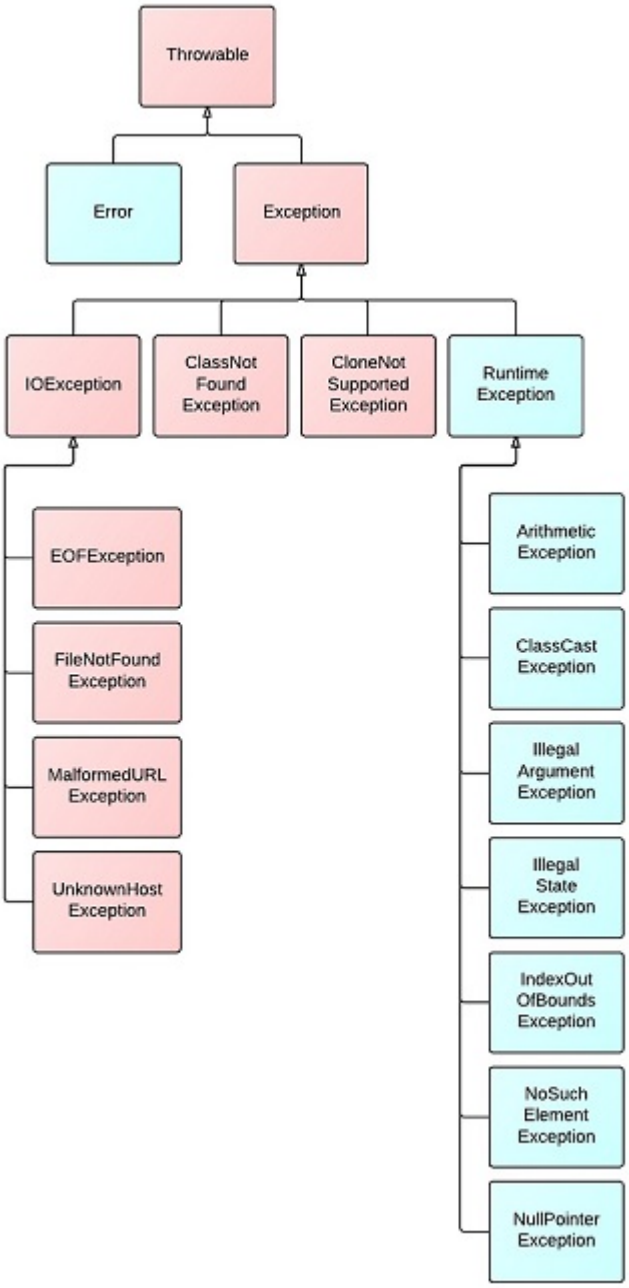
通过上面的代码, 我们学习了 `@Transactional` 的基本使用. 接下来我们学习 `@Transactional` 注解的使用细节.

我们主要学习 `@Transactional` 注解当中的三个常见属性:

1. rollbackFor: 异常回滚属性. 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
2. Isolation: 事务的隔离级别. 默认值为 `Isolation.DEFAULT`
3. propagation: 事务的传播机制. 默认值为 `Propagation.REQUIRED`

3.1 rollbackFor

`@Transactional` 默认只在遇到运行时异常和Error时才会回滚, 非运行时异常不回滚. 即Exception的子类中, 除了RuntimeException及其子类.



我们上面为了演示事务回滚, 手动设置了程序异常

```
int a = 10/0;
```

接下来我们把异常改为如下代码

```
@Transactional
@RequestMapping("/r2")
public String r2(String name,String password) throws IOException {
    //用户注册
    userService.registryUser(name,password);
    log.info("用户数据插入成功");
    if (true){
```

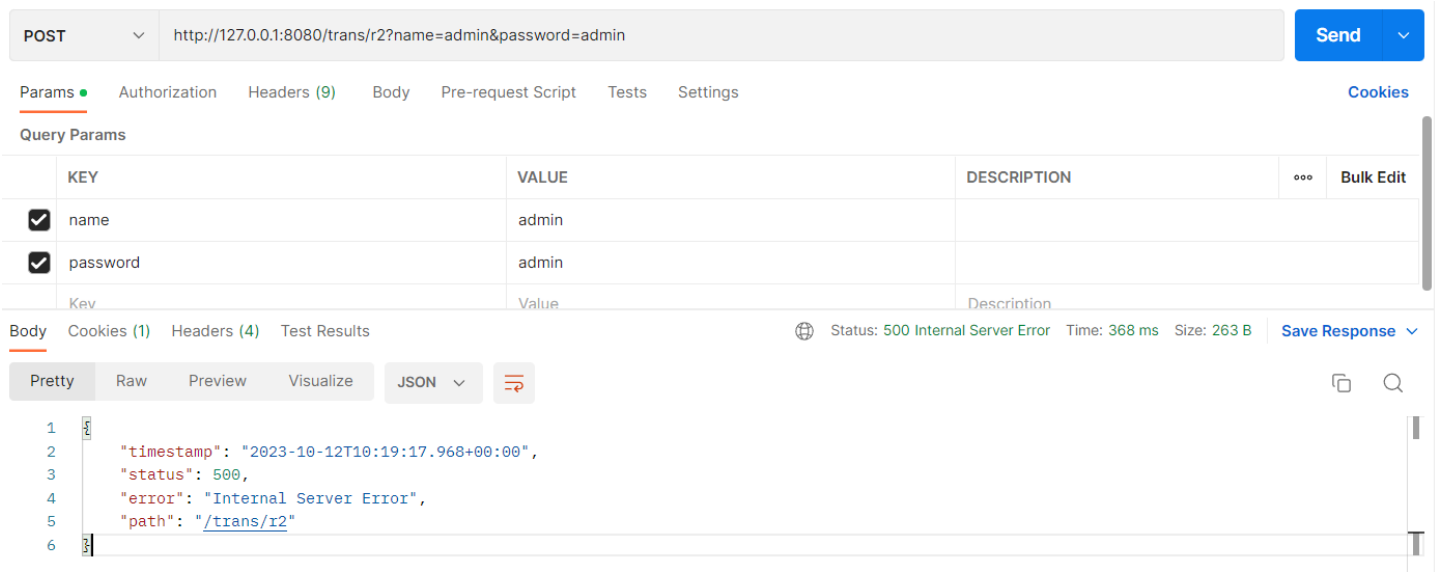
```
        throw new IOException();
    }
    return "r2";
}
```

运行程序

> 当前表中数据:

信息 | 摘要 | 结果 1 | 剖析 | 状态

id	user_name	password	create_time	update_time
(N/A)	(N/A)	(N/A)	(N/A)	(N/A)



发现虽然程序抛出了异常, 但是事务依然进行了提交.

> 运行后表中数据:

信息 | 摘要 | 结果 1 | 剖析 | 状态

id	user_name	password	create_time	update_time
13	admin	admin	2023-10-12 18:23:53	2023-10-12 18:23:53

如果我们需要所有异常都回滚, 需要来配置 `@Transactional` 注解当中的 `rollbackFor` 属性, 通过 `rollbackFor` 这个属性指定出现何种异常类型时事务进行回滚.

```
@Transactional(rollbackFor = Exception.class)
@RequestMapping("/r2")
public String r2(String name,String password) throws IOException {
    //用户注册
    userService.registryUser(name,password);
    log.info("用户数据插入成功");
```

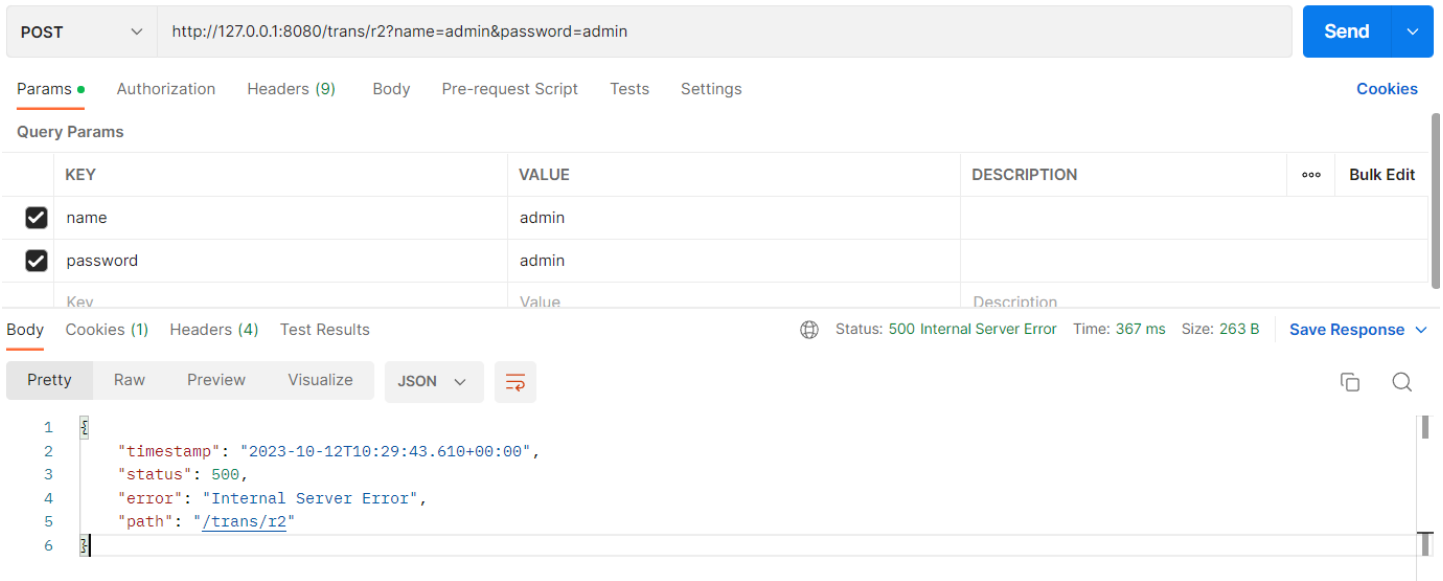
```
7       if (true){
8           throw new IOException();
9       }
10      return "r2";
11  }
```

运行程序

> 当前表中数据:

信息 | 摘要 | 结果 1 | 剖析 | 状态

id	user_name	password	create_time	update_time
13	admin	admin	2023-10-12 18:23:53	2023-10-12 18:23:53



发现虽然程序抛出了异常, 但是事务依然进行了提交.

> 运行后表中数据:

信息 | 摘要 | 结果 1 | 剖析 | 状态

id	user_name	password	create_time	update_time
13	admin	admin	2023-10-12 18:23:53	2023-10-12 18:23:53

结论:

- 在Spring的事务管理中，默认只在遇到运行时异常RuntimeException和Error时才会回滚.
- 如果需要回滚指定类型的异常, 可以通过rollbackFor属性来指定.

3.2 事务隔离级别

3.2.1 MySQL 事务隔离级别(回顾)

SQL 标准定义了四种隔离级别, MySQL 全都支持. 这四种隔离级别分别是:

1. 读未提交(READ UNCOMMITTED): 读未提交, 也叫未提交读. 该隔离级别的事务可以看到其他事务中未提交的数据.

   > 因为其他事务未提交的数据可能会发生回滚, 但是该隔离级别却可以读到, 我们把该级别读到的数据称之为脏数据, 这个问题称之为**脏读.**

2. 读提交(READ COMMITTED): 读已提交, 也叫提交读. 该隔离级别的事务能读取到已经提交事务的数据,

   > 该隔离级别不会有脏读的问题.但由于在事务的执行中可以读取到其他事务提交的结果, 所以在不同时间的相同 SQL 查询可能会得到不同的结果, 这种现象叫做**不可重复读**

3. 可重复读(REPEATABLE READ): 事务不会读到其他事务对已有数据的修改, 即使其他事务已提交. 也就可以确保同一事务多次查询的结果一致, 但是其他事务新插入的数据, 是可以感知到的. 这也就引发了幻读问题. **可重复读, 是 MySQL 的默认事务隔离级别.**

   > 比如此级别的事务正在执行时, 另一个事务成功的插入了某条数据, 但因为它每次查询的结果都是一样的, 所以会导致查询不到这条数据, 自己重复插入时又失败(因为唯一约束的原因). 明明在事务中查询不到这条信息，但自己就是插入不进去, 这个现象叫**幻读**.

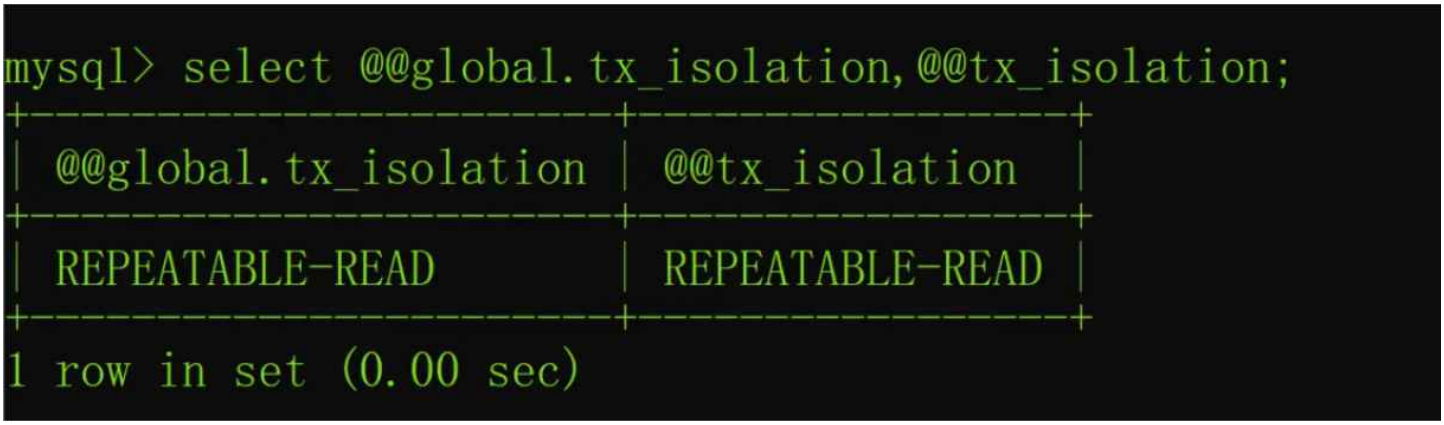
4. 串行化(SERIALIZABLE): 序列化, 事务最高隔离级别. 它会强制事务排序, 使之不会发生冲突, 从而解决了脏读, 不可重复读和幻读问题, 但因为执行效率低, 所以真正使用的场景并不多.

事务隔离级别	脏读	不可重复读	幻读
读未提交（READ UNCOMMITTED）	√	√	√
读已提交（READ COMMITTED）	×	√	√
可重复读（REPEATABLE READ）	×	×	√
串行化（SERIALIZABLE）	×	×	×

在数据库中通过以下 SQL 查询全局事务隔离级别和当前连接的事务隔离级别:

```
1 select @@global.tx_isolation,@@tx_isolation;
```

以上 SQL 的执行结果如下:



3.2.2 Spring 事务隔离级别

Spring 中事务隔离级别有5 种:

1. **`Isolation.DEFAULT`** : 以连接的数据库的事务隔离级别为主.
2. **`Isolation.READ_UNCOMMITTED`** : 读未提交, 对应SQL标准中 `READ UNCOMMITTED`
3. **`Isolation.READ_COMMITTED`** : 读已提交,对应SQL标准中 `READ COMMITTED`
4. **`Isolation.REPEATABLE_READ`** : 可重复读, 对应SQL标准中 `REPEATABLE READ`
5. **`Isolation.SERIALIZABLE`** : 串行化, 对应SQL标准中 `SERIALIZABLE`

```
public enum Isolation {
    DEFAULT(-1),
    READ_UNCOMMITTED(1),
    READ_COMMITTED(2),
    REPEATABLE_READ(4),
    SERIALIZABLE(8);

    private final int value;

    private Isolation(int value) {
        this.value = value;
    }

    public int value() {
        return this.value;
    }
}
```

Spring 中事务隔离级别可以通过 `@Transactional` 中的 `isolation` 属性进行设置

```
@Transactional(isolation = Isolation.READ_COMMITTED)
@RequestMapping("/r3")
```

```
3 public String r3(String name,String password) throws IOException {
4     //... 代码省略
5     return "r3";
6 }
```

3.3 Spring 事务传播机制

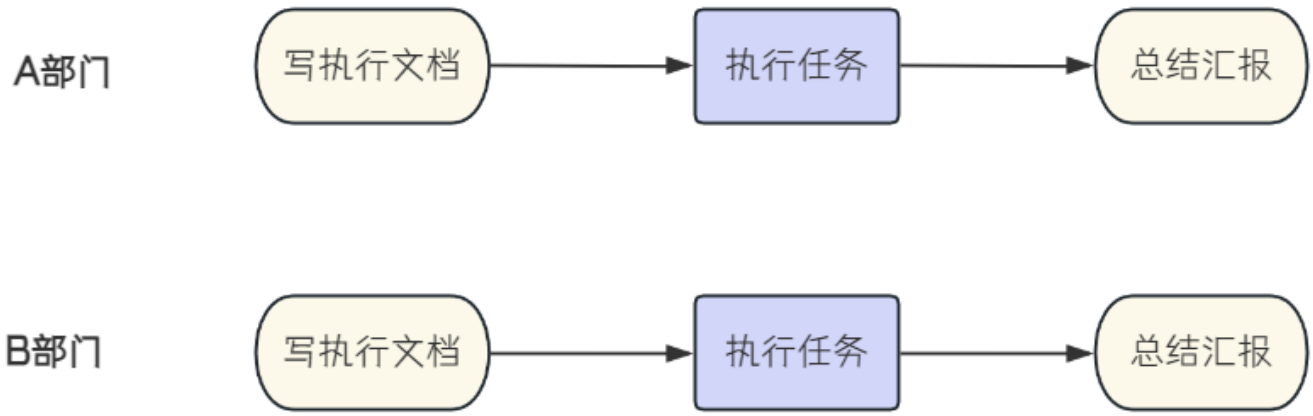
3.3.1 什么是事务传播机制

事务传播机制就是: 多个事务方法存在调用关系时, 事务是如何在这些方法间进行传播的.

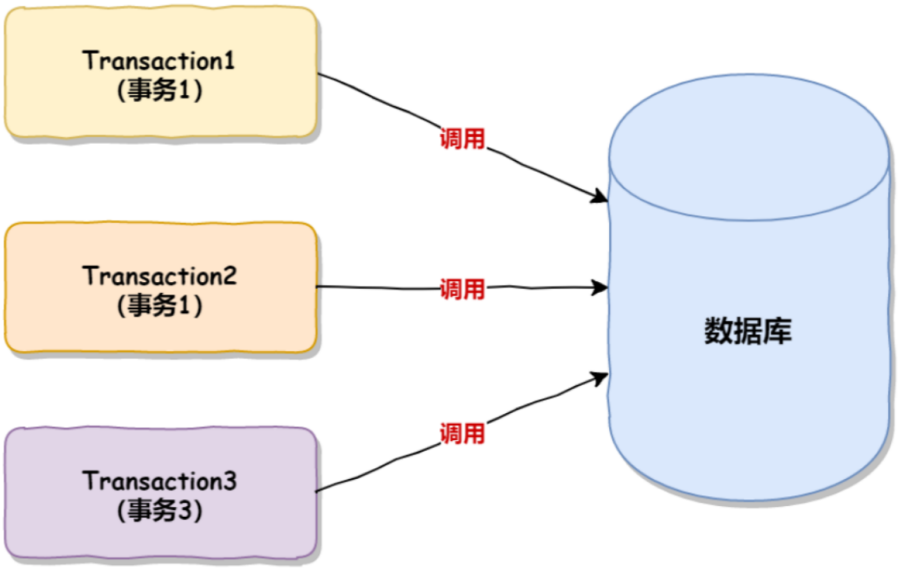
比如有两个方法A, B都被 `@Transactional` 修饰, A方法调用B方法

A方法运行时, 会开启一个事务. 当A调用B时, B方法本身也有事务, 此时B方法运行时, 是加入A的事务, 还是创建一个新的事务呢?

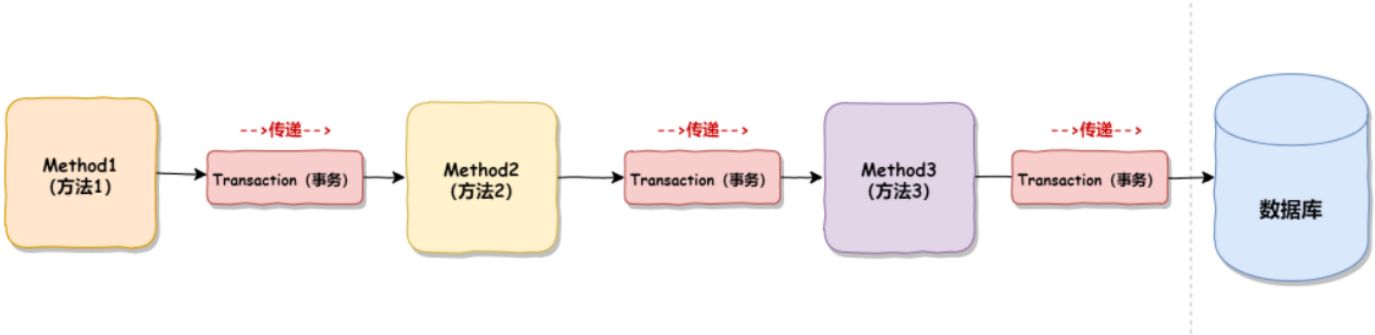
这个就涉及到了事务的传播机制.

> 比如公司流程管理
>
> 执行任务之前, 需要先写执行文档, 任务执行结束, 再写总结汇报
>
> 
>
>
> 此时A部门有一项工作, 需要B部门的支援, 此时B部门是直接使用A部门的文档, 还是新建一个文档呢?

事务隔离级别解决的是多个事务同时调用一个数据库的问题



而事务传播机制解决的是一个事务在多个节点(方法)中传递的问题



3.3.2 事务的传播机制有哪些

`@Transactional` 注解支持事务传播机制的设置, 通过 `propagation` 属性来指定传播行为.

Spring 事务传播机制有以下 7 种:

1. **`Propagation.REQUIRED`** : 默认的事务传播级别. 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.
2. **`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.
3. **`Propagation.MANDATORY`** :强制性. 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.
4. **`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新开启自己的事务, 且开启的事务相互独立, 互不干扰.
5. **`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(不用).
6. **`Propagation.NEVER`** : 以非事务方式运行, 如果当前存在事务, 则抛出异常.

7. `Propagation.NESTED`: 如果当前存在事务, 则创建一个事务作为当前事务的嵌套事务来运行. 如果当前没有事务, 则该取值等价于 `PROPAGATION_REQUIRED`.

```
1  public enum Propagation {
2      REQUIRED(0),
3      SUPPORTS(1),
4      MANDATORY(2),
5      REQUIRES_NEW(3),
6      NOT_SUPPORTED(4),
7      NEVER(5),
8      NESTED(6);
9
10     private final int value;
11
12     private Propagation(int value) {
13         this.value = value;
14     }
15
16     public int value() {
17         return this.value;
18     }
19 }
```

> 比如一对新人要结婚了, 关于是否需要房子
>
> 1. `Propagation.REQUIRED`: 需要有房子. 如果你有房, 我们就一起住, 如果你没房, 我们就一起买房. (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)
> 2. `Propagation.SUPPORTS`: 可以有房子. 如果你有房, 那就一起住. 如果没房, 那就租房. (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)
> 3. `Propagation.MANDATORY`: 必须有房子. 要求必须有房, 如果没房就不结婚. (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)
> 4. `Propagation.REQUIRES_NEW`: 必须买新房. 不管你有没有房, 必须要两个人一起买房. 即使有房也不住. (创建一个新的事务. 如果当前存在事务, 则把当前事务挂起)
> 5. `Propagation.NOT_SUPPORTED`: 不需要房. 不管你有没有房, 我都不住, 必须租房.(以非事务方式运行, 如果当前存在事务, 则把当前事务挂起)
> 6. `Propagation.NEVER`: 不能有房子. (以非事务方式运行, 如果当前存在事务, 则抛出异常)
> 7. `Propagation.NESTED`: 如果你没房, 就一起买房. 如果你有房, 我们就以房子为根据地,做点下生意. (如果如果当前存在事务, 则创建一个事务作为当前事务的嵌套事务来运行. 如果当前没有事务, 则该取值等价于 `PROPAGATION_REQUIRED`)

3.3.3 Spring 事务传播机制使用和各种场景演示

对于以上事务传播机制，我们重点关注以下两个就可以了：

1. REQUIRED（默认值）
2. REQUIRES_NEW

3.3.3.1 REQUIRED(加入事务)

看下面代码实现:

1. 用户注册, 插入一条数据
2. 记录操作日志, 插入一条数据(出现异常)

观察 `propagation = Propagation.REQUIRED` 的执行结果

```
@RequestMapping("/propaga")
@RestController
public class PropagationController {
    @Autowired
    private UserService userService;

    @Autowired
    private LogService logService;

    @Transactional(propagation = Propagation.REQUIRED)
    @RequestMapping("/p1")
    public String r3(String name,String password){
        //用户注册
        userService.registryUser(name,password);
        //记录操作日志
        logService.insertLog(name,"用户注册");
        return "r3";
    }
}
```

对应的UserService和LogService都添加上 `@Transactional(propagation = Propagation.REQUIRED)`

```
@Slf4j
@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;
```

```
6
7       @Transactional(propagation = Propagation.REQUIRED)
8       public void registryUser(String name,String password){
9           //插入用户信息
10          userInfoMapper.insert(name,password);
11      }
12  }
```

```
1  @Slf4j
2  @Service
3  public class LogService {
4      @Autowired
5      private LogInfoMapper logInfoMapper;
6
7      @Transactional(propagation = Propagation.REQUIRED)
8      public void insertLog(String name,String op){
9          int a=10/0;
10         //记录用户操作
11         logInfoMapper.insertLog(name,"用户注册");
12     }
13 }
```

运行程序, 发现数据库没有插入任何数据.

流程描述:

1. p1 方法开始事务
2. 用户注册, 插入一条数据 (执行成功) (和p1 使用同一个事务)
3. 记录操作日志, 插入一条数据(出现异常, 执行失败) (和p1 使用同一个事务)
4. 因为步骤3出现异常, **事务回滚**. 步骤2和3使用同一个事务, 所以**步骤2的数据也回滚**了.

3.3.3.2 REQUIRES_NEW(新建事务)

将上述UserService 和LogService 中相关方法事务传播机制改为 `Propagation.REQUIRES_NEW`

```
1  @Service
2  public class UserService {
3      @Autowired
4      private UserInfoMapper userInfoMapper;
5
6      @Transactional(propagation = Propagation.REQUIRES_NEW)
7      public void registryUser(String name,String password){
```

```
        //插入用户信息
        userInfoMapper.insert(name,password);
    }
}
```

```
@Service
public class LogService {
    @Autowired
    private LogInfoMapper logInfoMapper;

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void insertLog(String name,String op){
        int a=10/0;
        //记录用户操作
        logInfoMapper.insertLog(name,"用户注册");
    }
}
```

运行程序, 发现用户数据插入成功了, 日志表数据插入失败.

LogService 方法中的事务不影响 UserService 中的事务.

> 当我们不希望事务之间相互影响时, 可以使用该传播行为.

3.3.3.3 NEVER (不支持当前事务, 抛异常)

修改UserService 中对应方法的事务传播机制为 `Propagation.NEVER`

```
@Slf4j
@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;

    @Transactional(propagation = Propagation.NEVER)
    public void registryUser(String name,String password){
        //插入用户信息
        userInfoMapper.insert(name,password);
    }
}
```

程序执行报错, 没有数据插入.

```
15:29:19.389 org.apache.catalina.core.ContainerBase.[Tomcat].[localhost].[/].[dispatcherServlet] log [http-nio-8080-exec-1] Servlet.service() for servlet
 [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is org.springframework.transaction
 .IllegalTransactionStateException: Existing transaction found for transaction marked with propagation 'never'] with root cause
org.springframework.transaction.IllegalTransactionStateException Create breakpoint : Existing transaction found for transaction marked with propagation 'never'
    at org.springframework.transaction.support.AbstractPlatformTransactionManager.handleExistingTransaction(AbstractPlatformTransactionManager.java:413)
    at org.springframework.transaction.support.AbstractPlatformTransactionManager.getTransaction(AbstractPlatformTransactionManager.java:352)
    at org.springframework.transaction.interceptor.TransactionAspectSupport.createTransactionIfNecessary(TransactionAspectSupport.java:595)
    at org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:382)
    at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:119)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
```

3.3.3.4 NESTED(嵌套事务)

将上述UserService 和LogService 中相关方法事务传播机制改为 `Propagation.NESTED`

```java
@Slf4j
@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;

    @Transactional(propagation = Propagation.NESTED)
    public void registryUser(String name,String password){
        //插入用户信息
        userInfoMapper.insert(name,password);
    }
}
```

```java
@Slf4j
@Service
public class LogService {
    @Autowired
    private LogInfoMapper logInfoMapper;

    @Transactional(propagation = Propagation.NESTED)
    public void insertLog(String name,String op){
        int a=10/0;
        //记录用户操作
        logInfoMapper.insertLog(name,"用户注册");
    }
}
```

运行程序, 发现没有任何数据插入.

流程描述:

1. Controller 中p1 方法开始事务
2. `UserService` 用户注册, 插入一条数据 (嵌套p1事务)

3. `LogService` 记录操作日志, 插入一条数据(出现异常, 执行失败) (嵌套p1事务, 回滚当前事务, 数据添加失败)
4. 由于是嵌套事务, `LogService` 出现异常之后, 往上找调用它的方法和事务, 所以用户注册也失败了.
5. 最终结果是两个数据都没有添加

> p1事务可以认为是父事务, 嵌套事务是子事务. 父事务出现异常, 子事务也会回滚, 子事务出现异常, 如果不进行处理, 也会导致父事务回滚.

3.3.3.5 NESTED和REQUIRED 有什么区别?

我们在 `LogService` 进行当前事务回滚, 修改 `LogService` 代码如下:

```
@Service
public class LogService {
    @Autowired
    private LogInfoMapper logInfoMapper;

    @Transactional(propagation = Propagation.NESTED)
    public void insertLog(String name,String op){
        try {
            int a=10/0;
        } catch (Exception e){
            //回滚当前事务

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
        }
        //记录用户操作
        logInfoMapper.insertLog(name,"用户注册");
    }
}
```

重新运行程序, 发现用户表数据添加成功, 日志表添加失败.

`LogService` 中的事务已经回滚, 但是嵌套事务不会回滚嵌套之前的事务, 也就是说嵌套事务可以实现部分事务回滚.

对比REQUIRED

把 NESTED 传播机制改为 REQUIRED, 修改代码如下:

```
@Service
public class UserService {
```

```
    @Autowired
    private UserInfoMapper userInfoMapper;

    @Transactional(propagation = Propagation.REQUIRED)
    public void registryUser(String name,String password){
        //插入用户信息
        userInfoMapper.insert(name,password);
    }
}
```

```
@Service
public class LogService {
    @Autowired
    private LogInfoMapper logInfoMapper;

    @Transactional(propagation = Propagation.REQUIRED)
    public void insertLog(String name,String op){
        try {
            int a=10/0;
        } catch (Exception e){
            //回滚当前事务

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
        }
        //记录用户操作
        logInfoMapper.insertLog(name,"用户注册");
    }
}
```

重新运行程序, 发现用户表和日志表的数据添加都失败了.

REQUIRED 如果回滚就是回滚所有事务, 不能实现部分事务的回滚. (因为属于同一个事务)

NESTED和REQUIRED区别

- 整个事务如果全部执行成功, 二者的结果是一样的.
- 如果事务一部分执行成功, **REQUIRED加入事务**会导致整个事务全部回滚. **NESTED**嵌套事务可以实现局部回滚, 不会影响上一个方法中执行的结果.

> 嵌套事务之所以能够实现部分事务的回滚, 是因为事务中有一个保存点(savepoint)的概念, 嵌套事务进入之后相当于新建了一个保存点, 而滚回时只回滚到当前保存点.
>
> 资料参考: https://dev.mysql.com/doc/refman/5.7/en/savepoint.html

13.3.4 SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

InnoDB supports the SQL statements SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT and the optional WORK keyword for ROLLBACK.

The SAVEPOINT statement sets a named transaction savepoint with a name of ***identifier***. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does *not* release the row locks that were stored in memory after the savepoint. (For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoints that were set at a later time than the named savepoint are deleted.

If the ROLLBACK TO SAVEPOINT statement returns the following error, it means that no savepoint with the specified name

---

REQUIRED 是加入到当前事务中, 并没有创建事务的保存点, 因此出现了回滚就是整个事务回滚, 这就是嵌套事务和加入事务的区别.

总结

1. Spring中使用事务, 有两种方式: 编程式事务(手动操作)和声明式事务. 其中声明式事务使用较多,在方法上添加 `@Transactional` 就可以实现了
2. 通过 `@Transactional(isolation = Isolation.SERIALIZABLE)` 设置事务的隔离级别. Spring 中的事务隔离级别有 5 种
3. 通过 @Transactional(propagation = Propagation.REQUIRED) 设置事务的传播机制, Spring 中的事务传播级别有 7 种, 重点关注 `REQUIRED` (默认值) 和 `REQUIRES_NEW`