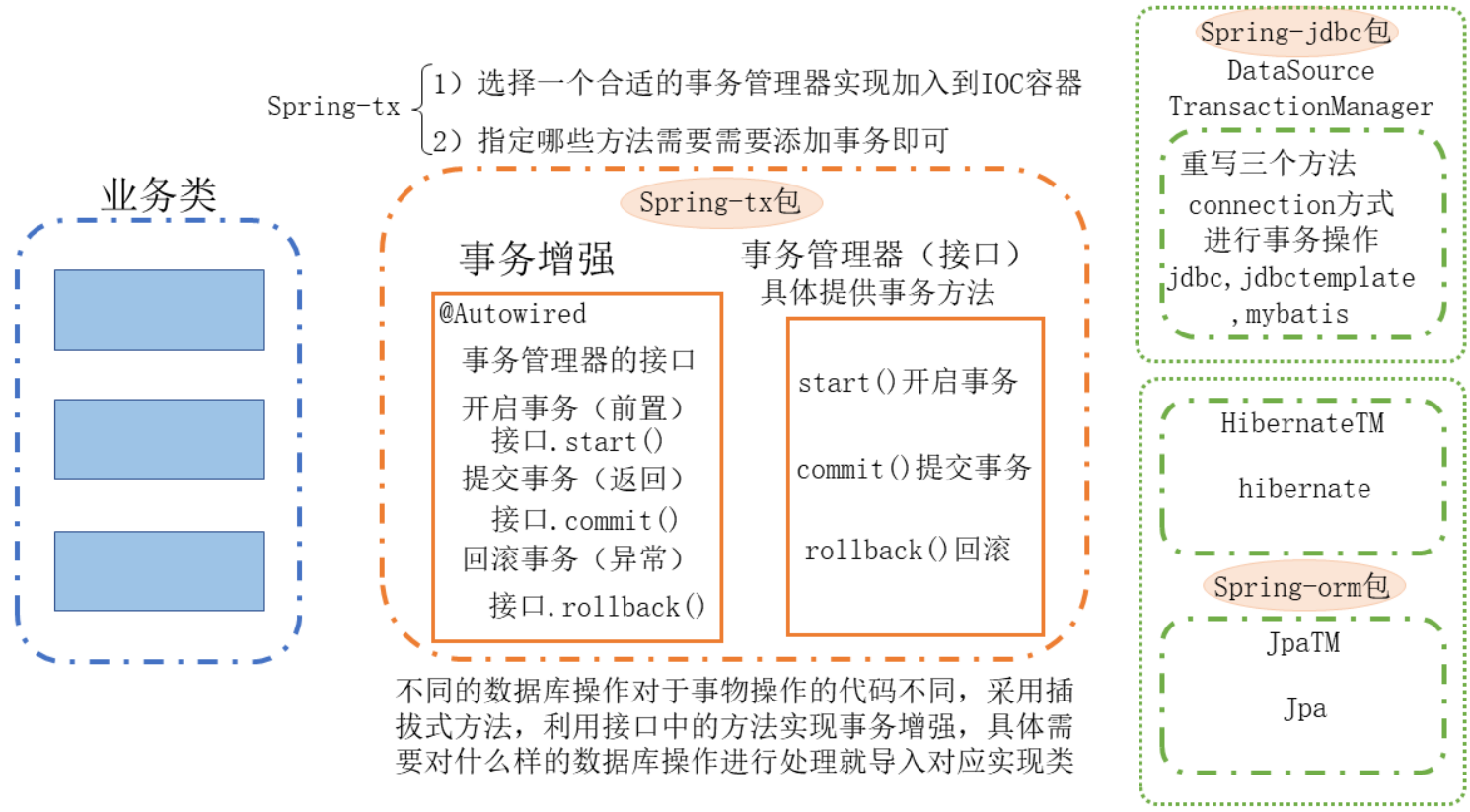


1、编程式事务

指手动编写程序来管理事务，即**通过编写代码**的方式**直接控制事务的提交和回滚**。在 Java 中,通常使用事务管理器(如 Spring 中的 PlatformTransactionManager)来实现编程式事务。

编程式的实现方式**存在缺陷**：

1）**细节没有被屏蔽**：具体操作过程中，所有细节都需要程序员自己来完成，**比较繁琐**；

2）**代码复用性不高**：如果没有有效抽取出来，每次实现功能都需要自己编写代码，代码就没有得到复用。

2、声明式事务

指使用**注解**或 **XML 配置**的方式来**控制事务的提交和回滚**。

开发者**只需要添加配置**即可，具体事务的实现**由第三方框架实现**，避免我们直接进行事务操作；

使用声明式事务可以**将事务的控制**和**业务逻辑分离**开来，提高代码的可读性和可维护性。

3、Spring 事务管理器

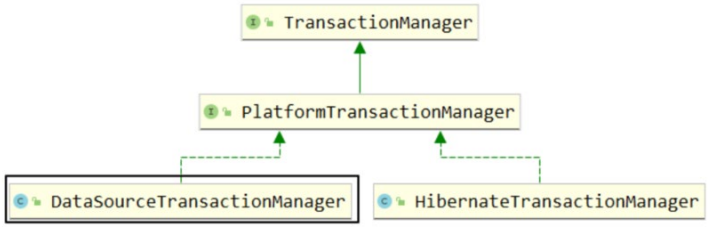
1）Spring 声明式事务对应依赖

spring-tx：包含声明式事务实现的基本规范（事务管理器规范接口和事务增强等等）；

spring-jdbc：包含 DataSource 方式事务管理器实现类 DataSourceTransactionManager；

spring-orm：包含其他持久层框架的事务管理器实现类例如：Hibernate/Jpa 等；

2）Spring 声明式事务对应事务管理器接口



我们现在要使用的事务管理器是 org.springframework.jdbc.datasource.**DataSourceTransactionManager**，将来整合 **JDBC 方式**、**JdbcTemplate 方式**、**Mybatis 方式**的事务实现。DataSourceTransactionManager 类中的主要方法：

doBegin()：开启事务

doSuspend()：挂起事务

doResume()：恢复挂起的事务

doCommit()：提交事务

doRollback()：回滚事务

4、事务属性：只读（readOnly=true）

对一个查询操作来说，如果我们把它设置成只读，就能够明确告诉数据库，这个操作不涉及写操作。数据库就能针对查询操作来进行优化。

```
// readOnly = true把当前事务设置为只读 默认是false!
@Transactional(readOnly = true)
```

【注意】1>）针对 DML 动作设置只读模式会抛出下面异常：

Caused by: java.sql.SQLException: Connection is read-only. Queries leading to data modification are not allowed

2> @Transactional 注解放类上（再对 DML 动作设置 readOnly = false）

- 生效原则

如果一个类中**每一个方法上都使用了@Transactional 注解**，那么就可以将@Transactional 注解提取到类上。反过来说：@Transactional 注解在类级别标记，会影响到类中的每一个方法。同时，类级别标记的@Transactional 注解中设置的事务属性也会延续影响到方法执行时的事务属性。除非在方法上又设置了@Transactional 注解。

对一个方法来说，**离它最近的@Transactional 注解中的事务属性生效**。

- 用法举例

在**类级别**@Transactional 注解中**设置只读**，这样类中**所有的查询方法**都**不需要设置@Transactional 注解**了。因为对查询操作来说，其他属性通常不需要设置，所以使用公共设置即可。然后在这个基础上，对**增删改方法设置@Transactional 注解 readOnly 属性为 false**。

```
@Service
@Transactional(readOnly = true)
public class EmpService {

    // 为了便于核对数据库操作结果，不要修改同一条记录
    @Transactional(readOnly = false)
    public void updateTwice(……) {
    ……
    }

    // readOnly = true把当前事务设置为只读
    // @Transactional(readOnly = true)
    public String getEmpName(Integer empId) {
    ……
    }
```

5、事务属性：超时时间（timeout=3，单位为秒）
事务在执行过程中，有可能因为遇到某些问题（可能是 Java 程序或MySQL 数据库或网络连接等），导致程序卡住，从而长时间占用数据库资源。此时这个很可能出问题的程序应该被回滚，撤销它已做的操作，事务结束，把资源让出来，让其他正常程序可以执行。
概括来说就是一句话：**超时回滚，释放资源**。

```
@Service
public class StudentService {

    @Autowired
    private StudentDao studentDao;

    /**
     * timeout设置事务超时时间,单位秒! 默认: -1 永不超时,不限制事务时间!
     */
    @Transactional(readOnly = false,timeout = 3)
    public void changeInfo(){
        studentDao.updateAgeById(100,1);
        //休眠4秒,等待方法超时!
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        studentDao.updateNameById("test1",1);
    }
}
```

【注意】如果在**类上加了 timeout=3，方法上不加 timeout=3** 则不会生效，因为方法上的注解会覆盖类上的注解。**（这里和只读不一样）**
6、事务属性：事务异常
默认**只针对运行时异常回滚**，编译时异常不回滚。
rollbackFor=异常类.class：指定哪些异常才会回滚，默认是 RuntimeException 和 Error 异常方可回滚；
noRollbackFor=异常类.class：指定哪些异常不会回滚， 默认没有指定，如果指定，应该在 rollbackFor 的范围内。

```
@Service
public class StudentService {

    @Autowired
    private StudentDao studentDao;

 @Transactional(rollbackFor=Exception.class,noRollbackFor=FileNotFoundException.class)
    public void changeInfo() throws FileNotFoundException {
        studentDao.updateAgeById(100,1);
        //主动抛出一个检查异常,测试! 发现不会回滚,因为不在rollbackFor的默认范围内!
        new FileInputStream("xxxx");
        studentDao.updateNameById("test1",1);
    }
}
```

7、事务属性：事务隔离级别（isolation = Isolation.某个隔离级别）
数据库事务的隔离级别是指**在多个事务并发执行时**，数据库系统为了**保证数据一致性所遵循的规定**。
常见的隔离级别包括：
1）读未提交（Read Uncommitted）：事务**可以读取未被提交的**数据，**易产生**脏读/不可重复读/幻读等问题。实现简单但不太安全，一般不用。
2）读已提交（Read Committed）：事务**只能读取已经提交的**数据，可以**避免脏读**问题，但**可能引发**不可重复读和幻读。
3）可重复读（Repeatable Read）：在一个事务中，相同的查询将返回相同的结果集，不管其他事务对数据做了什么修改。可以**避免**脏读和不可重复读，但**仍有幻读的问题**。
4）串行化（Serializable）：**最高的隔离级别，完全禁止了并发**，只允许一个事务执行完毕之后才能执行另一个事务。可以避免以上所有问题，但**效率较低，不适用于高并发**场景。 不同的隔离级别适用于不同的场景，需要根据实际业务需求进行选择和调整。

```
/**
 * timeout设置事务超时时间,单位秒! 默认: -1 永不超时,不限制事务时间!
 * rollbackFor = 指定哪些异常才会回滚,默认是 RuntimeException and Error 异常方可回滚!
 * noRollbackFor = 指定哪些异常不会回滚, 默认没有指定,如果指定,应该在rollbackFor的范围内!
 * isolation = 设置事务的隔离级别,mysql默认是repeatable read!
 */
@Transactional(readOnly = false,
               timeout = 3,
               rollbackFor = Exception.class,
               noRollbackFor = FileNotFoundException.class,
               isolation = Isolation.REPEATABLE_READ)
public void changeInfo() throws FileNotFoundException {
    studentDao.updateAgeById(100,1);
    //主动抛出一个检查异常,测试! 发现不会回滚,因为不在rollbackFor的默认范围内!
    new FileInputStream("xxxx");
    studentDao.updateNameById("test1",1);
}
```

8、事务属性：事务传播行为（在**子方法上**设置传播行为）
（**propagation = Propagation.**REQUIRES(默认值)/REQUIRES_NEW）
事务传播行为要研究的问题：较早执行的方法开启事务后，如何传播给后面需要使用的方法（较晚执行的方法）

```
@Transactional
public void MethodA(){
    // ...
    MethodB();
    // ...
}

//在被调用的子方法中设置传播行为，代表如何处理调用的事务！ 是加入，还是新事务等！
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void MethodB(){
    // ...
}
```

@Transactional 注解通过 propagation 属性设置事务传播行为。属性 propagation 可选值**由 transaction.annotation.Propagation 枚举类**提供：

名称	含义
REQUIRED 默认值	如果父方法有事务，就加入，如果没有就新建自己独立！
REQUIRES_NEW	不管父方法是否有事务，我都新建事务，都是独立的！

【注意】1）当两个事务整合到一个父事务中时，**一个事务出错，另一个事务也不会执行成功**（默认值 required）；**另一个事务作为独立事务依旧执行成功**（required_new）；

```
@Service
public class StudentService {

    @Autowired
    private StudentDao studentDao;
    /**
     * 声明两个独立修改数据库的事务业务方法
     * propagation = Propagation.REQUIRED: 父方法有事务，子方法加入父方法的事务
     */
    @Transactional(propagation = Propagation.REQUIRED)
    public void changeAge(){
        studentDao.updateAgeById(99,1);
    }
    @Transactional(propagation = Propagation.REQUIRED)
    public void changeName(){
        studentDao.updateNameById("test2",1);
        int i = 1/0;
    }
}
```

```
@Service
public class TopService {

    @Autowired
    private StudentService studentService;

    @Transactional
    public void  topService(){
        studentService.changeAge();
        studentService.changeName();
    }
}
```

2）**在同一个类中**，对于@Transactional 注解的方法调用，**事务传播行为不会生效**。这是因为 Spring 框架中**使用代理模式实现了事务机制，在同一个类中的方法调用并不经过代理**，而是**通过对象的方法调用**，因此@Transactional 注解的设置不会被代理捕获，也就不会产生任何事务传播行为的效果。
【了解】其他传播行为值：
1）Propagation.REQUIRED：如果当前存在事务，则加入当前事务，否则创建一个新事务；
2）Propagation.REQUIRES_NEW：创建一个新事务，并在新事务中执行。如果当前存在事务，则挂起当前事务，即使新事务抛出异常，也不会影响当前事务；
3）Propagation.NESTED：如果当前存在事务，则在该事务中嵌套一个新事务，如果没有事务，则与 Propagation.REQUIRED 一样；
4）Propagation.SUPPORTS：如果当前存在事务，则加入该事务，否则以非事务方式执行；
5）Propagation.NOT_SUPPORTED：以非事务方式执行，如果当前存在事务，挂起该事务；
6）Propagation.MANDATORY：必须在一个已有的事务中执行，否则抛出异常；
7）Propagation.NEVER：必须在没有事务的情况下执行，否则抛出异常。

基于注解的声明式事务（了解）

1、添加依赖

```
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>2.1.1</version>
</dependency>

<!-- 数据库驱动 和 连接池-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>

<!-- spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>6.0.6</version>
</dependency>

<!-- 声明式事务依赖-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>6.0.6</version>
</dependency>
```

2、外部配置文件 jdbc.properties

```
atguigu.url=jdbc:mysql://localhost:3306/studb
atguigu.driver=com.mysql.cj.jdbc.Driver
atguigu.username=root
atguigu.password=root
```

3、spring 配置文件

```
/**
 * projectName: com.atguigu.config
 *
 * description: 数据库和连接池配置类
 */

@Configuration
@ComponenScan("com.atguigu")
@PropertySource(value = "classpath:jdbc.properties")
@EnableTransactionManagement
public class DataSourceConfig {

    /**
     * 实例化dataSource加入到ioc容器
     * @param url
     * @param driver
     * @param username
     * @param password
     * @return
     */
    @Bean
    public DataSource dataSource(@Value("${atguigu.url}")String url,
                                 @Value("${atguigu.driver}")String driver,
                                 @Value("${atguigu.username}")String username,
                                 @Value("${atguigu.password}")String password){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);

        return dataSource;
    }

    /**
     * 实例化JdbcTemplate对象,需要使用ioc中的DataSource
     * @param dataSource
     * @return
     */
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource){
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    /**
     * 装配事务管理实现对象
     * @param dataSource
     * @return
     */
    @Bean
    public TransactionManager transactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}
```

4、dao 层和 service 层

```
@Repository
public class StudentDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void updateNameById(String name,Integer id){
        String sql = "update students set name = ? where id = ? ;";
        int rows = jdbcTemplate.update(sql, name, id);
    }

    public void updateAgeById(Integer age,Integer id){
        String sql = "update students set age = ? where id = ? ;";
        jdbcTemplate.update(sql,age,id);
    }
}
```

```
/**
 * projectName: com.atguigu.service
 *
 */
@Service
public class StudentService {

    @Autowired
    private StudentDao studentDao;

    @Transactional
    public void changeInfo(){
        studentDao.updateAgeById(100,1);
        System.out.println("-----------");
        int i = 1/0;
        studentDao.updateNameById("test1",1);
    }
}
```

【**重点**】在 springboot 中只需要引入 **spring-boot-starter-jdbc** 依赖，SpringBoot 项目会自动配置一个 DataSourceTransactionManager，所以我们只需在方法（或者类）加上@Transactional 注解，就自动纳入 Spring 的事务管理。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```