编程式事务：通过编程代码在业务逻辑时需要时自行实现，粒度更小；
声明式事务：通过注解或XML配置实现；

编程式和声明式事务的区别

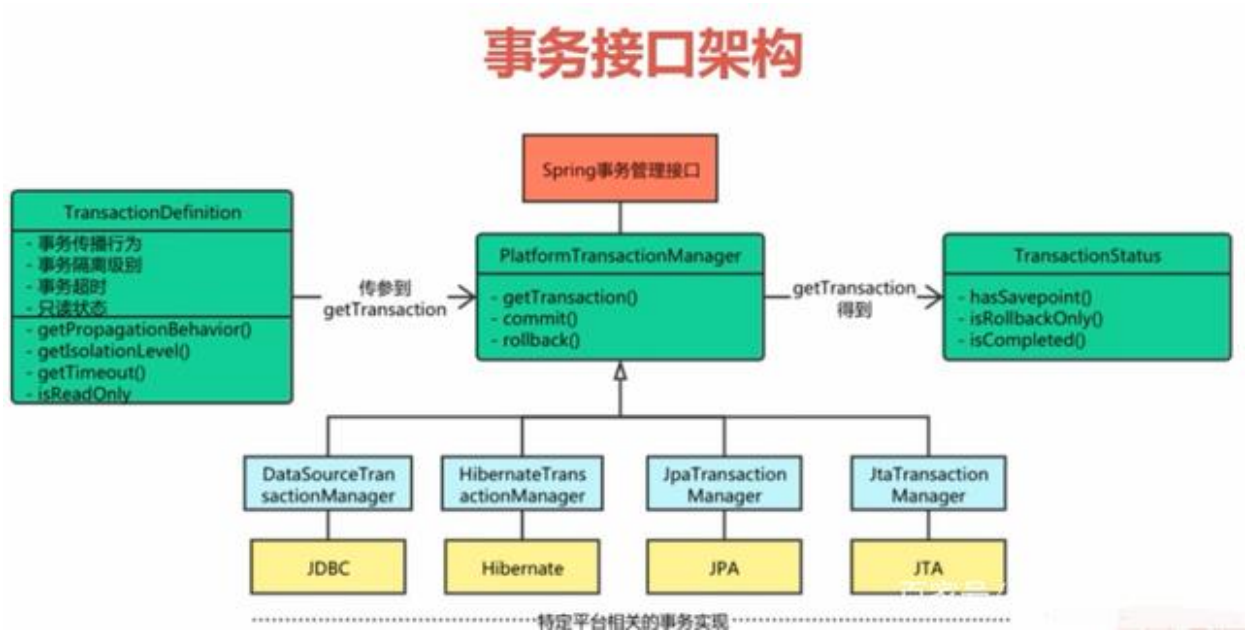
Spring提供了对编程式事务和声明式事务的支持，编程式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于AOP）有助于用户将操作与事务规则进行解耦。
简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

如何实现编程式事务？

Spring提供两种方式的编程式事务管理，分别是：使用TransactionTemplate和直接使用PlatformTransactionManager。
是侵入性事务管理，直接使用底层的PlatformTransactionManager、使用TransactionTemplate(Spring推荐使用)；
编程式事务管理对基于 POJO 的应用来说是唯一选择。我们需要在代码中调用beginTransaction()、commit()、rollback()等事务管理相关的方法；
编程式事务每次实现都要单独实现，但业务量大且功能复杂时，使用编程性事务无疑是痛苦的；而声明式事务不同，声明式事务属于非侵入性，不会影响业务逻辑的实现，只需在配置文件中做相关的事务规则声明（或通过基于@Transactional注解的方式），便可以将事务规则应用到业务逻辑中；

声明式事务：该事务是建立在AOP之上的，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加入一个事务，在执行完目标方法之后根据执行情况提交或回滚事务。
Spring配置文件中关于事务配置总是由三个组成部分，分别是DataSource、TransactionManager和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。DataSource、TransactionManager这两部分只是会根据数据访问方式有所变化，比如使用Hibernate进行数据访问时，DataSource实际为SessionFactory，TransactionManager的实现为HibernateTransactionManager。根据代理机制的不同，总结了四种Spring事务的配置方式，如下图：

1. 使用拦截器：基于TransactionInterceptor 类来实施声明式事务管理功能（Spring最初提供的实现方式）；
2. Bean和代理：基于 TransactionProxyFactoryBean的声明式事务管理
3. 使用tx标签配置的拦截器：基于tx和aop名字空间的xml配置文件（基于Aspectj AOP配置事务）；
4. 全注解：基于@Transactional注解；



编程式事务

通过TransactionTemplate或TransactionManager手动管理事务，因为是手动，所以实际开发中很少使用。但便于理解Spring事务管理原理，建议新手先学习编程式事务。

我们通过一个案例去理解Spring中的事务，在数据表account中有zhangsan和lisi两个人，现在用编程式事务的方式实现转账功能。

首先在pom.xml文件中添加依赖spring-context + spring-jdbc + druid + mysql-connector-java + junit（用于测试）

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>tx</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.2.9</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.27</version>
        </dependency>

    </dependencies>
</project>
```

```
package com.qfedu.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

/**
 * 操作类
 */
@Repository
public class AccountDao {
    @Autowired
    JdbcTemplate jdbcTemplate;

    /**
     * 向指定用户的余额中添加指定额度
     * @param username 指定用户
     * @param money 添加多少金额
     */
    public void addMoney(String username, Double money) {
        jdbcTemplate.update("update account set money=money+? where
username=?;", money, username);
    }

    /**
     * 减少指定用户的余额
     * @param username 指定用户
     * @param money 较少多少金额
     */
    public void minusMoney(String username, Double money) {
        jdbcTemplate.update("update account set money=money-? where
username=?;", money, username);
    }
}
```

使用TransactionTamplate

使用TransactionTemplate，该类继承了接口DefaultTransactionDefinition，用于简化事务管理，事务管理由模板类定义，主要是通过TransactionCallback回调接口或TransactionCallbackWithoutResult回调接口指定，通过调用模板类的参数类型为TransactionCallback或TransactionCallbackWithoutResult的execute方法来自动享受事务管理。

TransactionTemplate模板类使用的回调接口：

- TransactionCallback：通过实现该接口的“T doInTransaction(TransactionStatus status)”方法来定义需要事务管理的操作代码；
- TransactionCallbackWithoutResult：继承TransactionCallback接口，提供“void doInTransactionWithoutResult(TransactionStatus status)”便利接口用于方便那些不需要返回值的事务操作代码。

```
package com.qfedu.demo.service;

import com.qfedu.demo.dao.AccountDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

/**
 * 服务类
 */
@Service
public class AccountService {
    @Autowired
    protected AccountDao accountDao;
    @Autowired
    private TransactionTemplate transactionTemplate;

    /**
     * 转账方法
     * @param from 金额减少的用户
     * @param to 金额增加的用户
     * @param money 转账的金额
     */
    public void transferMoney(String from, String to, Double money) {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                try {
                    accountDao.minusMoney(from, money);
                    accountDao.addMoney(to, money);
                } catch (Exception e) {
                    e.printStackTrace();
                    //回滚
                    status.setRollbackOnly();
                }
            }
        });
```

```
    }
}
```

PlatformTransactionManager接口定义如下

```
public interface PlatformTransactionManager {
        TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;//返回一个已经激活的事务或创建一个新的事务（具体由TransactionDefinition参数定义的事务属性决定），返回的TransactionStatus对象代表了当前事务的状态，其中该方法抛出TransactionException（未检查异常）表示事务由于某种原因失败。
        void commit(TransactionStatus status) throws TransactionException;//用于提交TransactionStatus参数代表的事务。
        void rollback(TransactionStatus status) throws TransactionException;//用于回滚TransactionStatus参数代表的事务。
}
```

TransactionDefinition接口定义如下：

```
public interface TransactionDefinition {
    int getPropagationBehavior(); //返回定义的事务传播行为
    int getIsolationLevel(); //返回事务隔离级别
    int getTimeout(); //返回定义的事务超时时间
    boolean isReadOnly(); //返回定义的事务是否是只读的
    String getName(); //返回事务名称
}
```

TransactionStatus接口定义如下：

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction(); //返回当前事务是否是新的事务
    boolean hasSavepoint(); //返回当前事务是否有保存点
    void setRollbackOnly(); //设置事务回滚
    boolean isRollbackOnly(); //设置当前事务是否应该回滚
    void flush(); //用于刷新底层会话中的修改到数据库，一般用于刷新如Hibernate/JPA的会话，可能对如JDBC类型的事务无任何影响；
    boolean isCompleted(); //返回事务是否完成
}
```

使用TransactionManager

TransactionManager是一个接口，并且是空接口。所以在具体的代码中，要用其的子类PlatformTransactionManager，在这个子类有且仅有三个方法，对应的功能分别是开启事务、提交事务和回滚事务。虽然在AccountService中使用的是PlatformTransactionManager，但是在applicationContext.xml配置文件中使用的是DataSourceTransactionManager。这利用的是泛型的知识。

```
package com.qfedu.demo.service;

import com.qfedu.demo.dao.AccountDao;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

@Service
public class AccountService01 {
    @Autowired
    private AccountDao accountDao;
    @Autowired
    private PlatformTransactionManager platformTransactionManager;

    public void TransferMoney(String from, String to, Double money) {
        DefaultTransactionDefinition definition = new DefaultTransactionDefinition();
        //开启事务
        TransactionStatus status =
platformTransactionManager.getTransaction(definition);
        try {
            accountDao.minusMoney(from, money);
            accountDao.addMoney(to, money);
            platformTransactionManager.commit(status);
        } catch (Exception e) {
            e.printStackTrace();
            platformTransactionManager.rollback(status);
        }
    }
}
```

applicationContext.xml配置文件，如果是使用TransactionManager，可以不用配置TransactionTemplate。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.qfedu.demo"/>

    <context:property-placeholder location="classpath:db.properties"/>
    <bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource">
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
        <property name="url" value="jdbc:mysql:///jdbc01"/>
```

```
    </bean>
    <bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--  创建一个，MySQL数据库的事务管理器-->
    <bean
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
id="transactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <bean class="org.springframework.transaction.support.TransactionTemplate"
id="transactionTemplate">
        <property name="transactionManager" ref="transactionManager"/>
    </bean>

</beans>


package com.qfedu.demo.service;

import org.junit.Before;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 测试类
 */
public class AccountTest01 {
    private ClassPathXmlApplicationContext ctx;

    @Before
    public void before() {
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
    }

    /**
     * 使用TransactionTemplate
     */
    @Test
    public void testTransactionManager() {
        AccountService accountService = ctx.getBean(AccountService.class);
        accountService.transferMoney("zhangsan", "lisi", 50.0);
    }

    /**
     * 使用TransactionManager
     */
```

```
    @Test
    public void testTransactionTemplate() {
        AccountService01 accountService01 = ctx.getBean(AccountService01.class);
        accountService01.TransferMoney("zhangsan","lisi",50.0);
    }
}
```

其实不管是使用TransactionManager，还是使用TransactionTemplate，都差不多。在实际开发中，主要是使用声明式事务， 因为声明式事务代码入侵少，其本质式通过AOP实现。

声明式事务

XML文件配置

首先在pom.xml配置文件中添加依赖，除了编程式事务的依赖外，因为使用了AOP，所以还需要aspectjrt 和 aspectjweaver

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>tx02</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.2.9</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.27</version>
```

```
        </dependency>
    </dependencies>
</project>
```

操作类AccountDao与编程式事务的AccoutDao一样，这里不再赘叙。

相比于编程式事务，声明式事务的AccountService显得极为简介，写上自己的业务就可以了，这是由于使用了AOP，那些相同的代码提前做好了。

```
package com.qfedu.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AccountService {
    @Autowired
    private AccountDao accountDao;

    public void transferMoney(String from, String to, Double money) {
        accountDao.minusMoney(from, money);
        accountDao.addMoney(to, money);
    }
}
```

声明式的 applicationContext.xml 配置文件主要分为五个步骤：配置数据源、配置JdbcTemplate、配置事务管理器、配置事务属性和配置AOP。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="com.qfedu.demo"/>
    <!--
    配置数据源
```

```
    -->
    <bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource">
        <property name="url" value="jdbc:mysql:///jdbc01"/>
        <property name="username" value="root"/>
        <property name="password" value="123456"/>
    </bean>
    <!--
    配置jdbcTemplate
    -->
    <bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!--
    配置事务管理器
    -->
    <bean
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
id="transactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

```
package com.qfedu.demo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 测试类
 */
class AccountServiceTest01 {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        AccountService accountService = ctx.getBean(AccountService.class);
        accountService.transferMoney("zhangsan", "lisi", 50.0);
    }
}
```

因为声明式事务的简洁，所以在实际开发中，一般都是使用声明式事务。

java代码配置---@transactional

配置类

```
package com.qfedu.demo01;

import com.alibaba.druid.pool.DruidDataSource;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

/**
 * 配置类
 */
@Configuration
@ComponentScan
//开启事务注解，通过注解标记事务
@EnableTransactionManagement
public class JavaConfig {

    @Bean
    DruidDataSource dataSource() {
        DruidDataSource ds = new DruidDataSource();
        ds.setUrl("jdbc:mysql:///jdbc01");
        ds.setUsername("root");
        ds.setPassword("123456");
        return ds;
    }

    @Bean
    JdbcTemplate jdbcTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource());
        return jdbcTemplate;
    }

    @Bean
    DataSourceTransactionManager transactionManager() {
        DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource());
        return transactionManager;
    }

}
```

其实就是把applicationContext.xml配置文件翻译程java代码，但是配置事务属性 和 配置AOP需要注意，applicationContext.xml中的配置事务属性对应Java代码配置

@EnableTransactionManager，表示开始事务注解，之后再AccountService#transferMoney添加注解@transactioal，表示为方法添加事务。

```
package com.qfedu.demo01;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * 服务类
 */
@Service
public class AccountService {
    @Autowired
    private AccountDao accountDao;

    //表示给方法添加事务，该注解如果在类上，则表示为该类的所有方法添加事务
    @Transactional
    public void transferMoney(String from, String to, Double money) {
        accountDao.minusMoney(from, money);
        accountDao.addMoney(to, money);
    }
}
```

```
package com.qfedu.demo01;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 * 测试类
 */
class JavaConfigTest01 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(JavaConfig.class);
        AccountService accountService = ctx.getBean(AccountService.class);
        accountService.transferMoney("zhangsan", "lisi", 50.0);
    }
}
```

使用@Transactional注意点：

1. 如果在接口、实现类或方法上都指定了@Transactional 注解，则优先级顺序为方法>实现类>接口；

2. 建议只在实现类或实现类的方法上使用@Transactional，而不要在接口上使用，这是因为如果使用JDK代理机制（基于接口的代理）是没问题；而使用使用CGLIB代理（继承）机制时就会遇到问题，因为其使用基于类的代理而不是接口，这是因为接口上的@Transactional注解是“不能继承的”；