Spring

概念

Spring是开源、轻量级、企业级的Java程序开发框架。

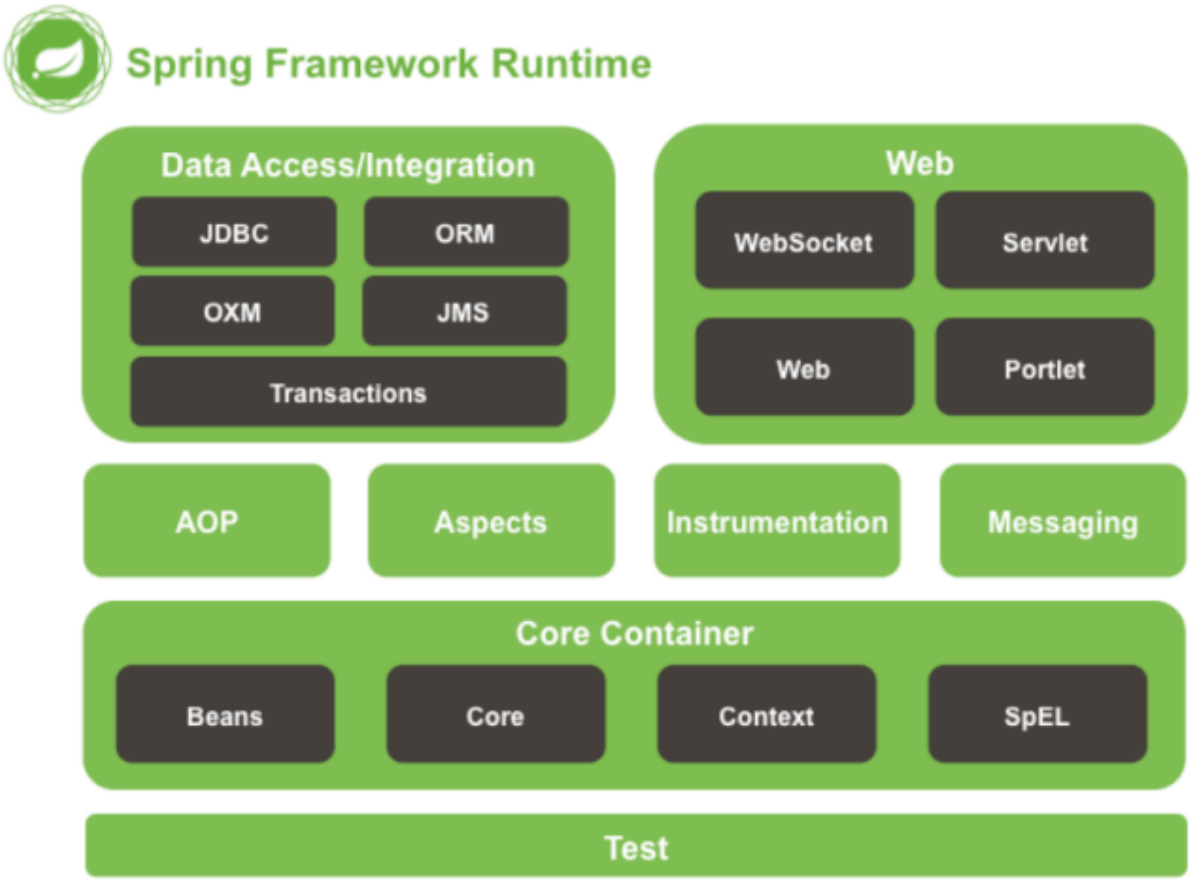
功能

1. **核心技术：提供IOC、AOP、事件、验证、国际化、表达式等**
2. 提供强大的测试：web应用测试
3. **数据访问：DAO、事务、JDBC、ORM等**
4. **Web应用支持：SpringMVC**
5. **集成其他技术：JMX、定时任务、缓存等**
6. 支持其他语言：Kotlin、Groovy等动态语言

优点

1. **低侵入式设计，代码污染极低**
2. **Spring的IOC（DI）机制降低了业务对象替换的复杂性，提高了组件之间的解耦**
3. **Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中式管理，从而提供了更好的复用**
4. 独立于各种应用服务器，基于Spring框架的应用，可以真正实现Write Once,Run Anywhere的承诺
5. Spring的ORM和DAO提供了与第三方持久层框架的良好整合，并简化了底层的数据库访问
6. Spring并不强制应用完全依赖于Spring，开发者可自由选用Spring框架的部分或全部

模块



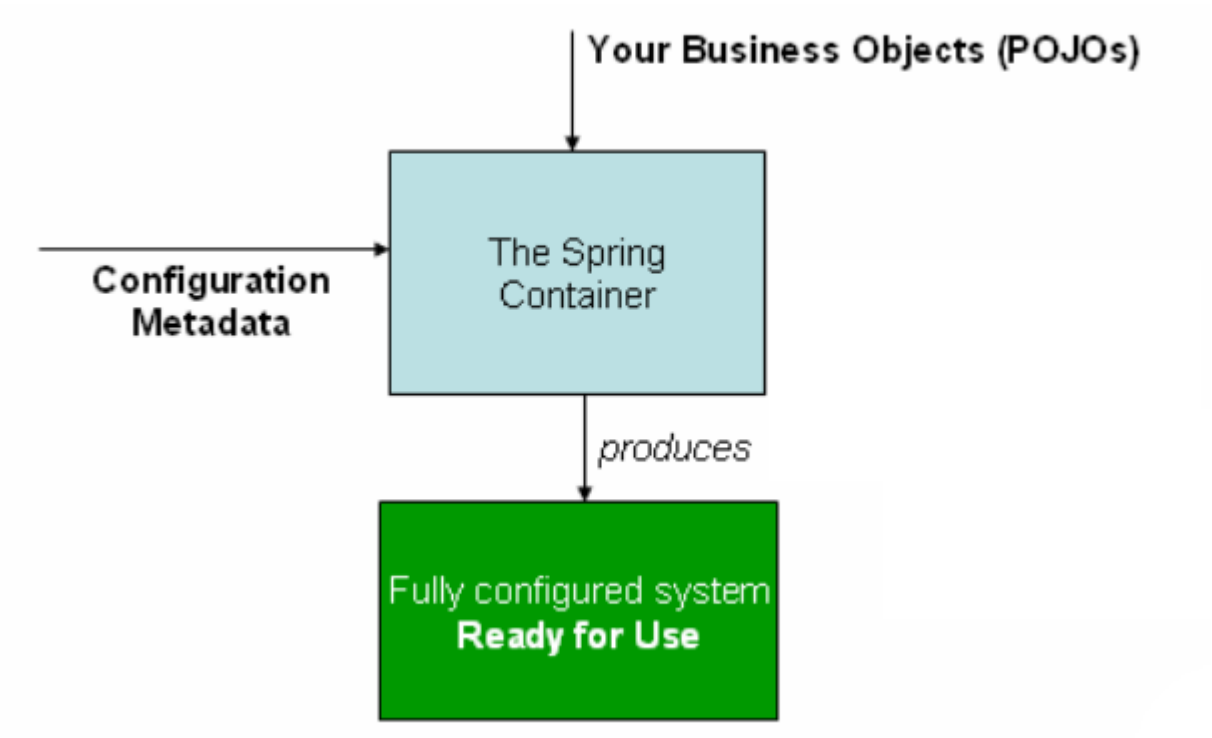
核心思想

1. IOC：控制反转，将对象创建、与其他对象之间的关系的控制权交给第三方（Spring容器）维护，主要是通过DI（依赖注入）实现。
2. AOP：面向切面编程，将一些通用任务如安全、事务、日志等进行集中式管理，从而提供了更好的复用。

IOC容器

Spring 容器是 Spring 框架的核心。容器将创建对象，把它们连接在一起，配置它们，并管理他们的整个生命周 期从创建到销毁。

Figure 7.1. The Spring IoC container



IOC容器种类

1. BeanFactory容器：它是最简单的容器，给 DI 提供了基本的支持。
    - XmlBeanFactory
2. ApplicationContext容器：该容器添加了更多的企业特定的功能。
    - FileSystemXmlAppliationContext
    - **ClassPathAppliationContext**
    - WebXmlAppliationContext
    - **AnnatationApplicationContext**

IOC容器执行过程

1. 加载元数据文件（xml、注解、java代码）
2. 初始化bean（通过反射创建bean对象）
3. 注入
    - **构造器注入**
    - **setter注入**
4. 执行程序

Bean

bean 是一个被实例化，组装，并通过 Spring IoC 容器所管理的对象。

Bean的定义

bean 定义包含称为配置元数据的信息，下述容器也需要知道配置元数据：

• 如何创建一个 bean

• bean 的生命周期的详细信息

• bean 的依赖关系

class	**这个属性是强制性的，并且指定用来创建 bean 的 bean 类。**
id(name)	这个属性指定唯一的 bean 标识符。在基于 XML 的配置元数据中，你可以使用ID 和/或 name 属性来指定 bean 标识符。
scope	这个属性指定由特定的 bean 定义创建的对象的作用域，它将会在 bean 作用域的章节中进行讨论。
constructor-arg	它是用来注入依赖关系的，并会在接下来的章节中进行讨论。
properties	它是用来注入依赖关系的，并会在接下来的章节中进行讨论。
autowiring mode	它是用来注入依赖关系的，并会在接下来的章节中进行讨论。
lazy-initialization mode	延迟初始化的 bean 告诉 IoC 容器在它第一次被请求时，而不是在启动时去创建一个 bean 实例。
initialization 方法	在 bean 的所有必需的属性被容器设置之后，调用回调方法。它将会在 bean 的生命周期章节中进行讨论。
destruction 方法	当包含该 bean 的容器被销毁时，使用回调方法。它将会在 bean 的生命周期章节中进行讨论

Bean作用域

Bean的作用域指Bean作用的时机，总共五个作用域，常用Singleton（单例，默认，每次调用都返回同一个实例）、prototype（原型，每次调用都会返回新的实例，由web容器创建，不受ioc容器管理）。

singleton	**该作用域将 bean 的定义的限制在每一个 Spring IoC 容器中的一个单一实例(默认)。**
prototype	**该作用域将单一 bean 的定义限制在任意数量的对象实例。**
request	该作用域将 bean 的定义限制为 HTTP 请求。只在 web-aware Spring ApplicationContext 的上下文中有效。
session	该作用域将 bean 的定义限制为 HTTP 会话。 只在web-aware Spring ApplicationContext的上下文中有效。
global-session	该作用域将 bean 的定义限制为全局 HTTP 会话。只在 web-aware Spring ApplicationContext 的上下文中有效。

Bean的生命周期

(1) 初始化

① 实现 InitializingBean接口

② 指定bean的init-method方法

③ 设置全局初始化方法

(2) 使用

(3) 销毁

① 实现 DisposableBean接口

② 指定bean的destroy-method方法

③ 设置全局销毁方法

DI

IOC(Inversion of Control)：其思想是反转资源获取的方向。传统的资源查找方式要求组件向容器发起请求查找资源，作为回应，容器适时的返回资源。 而应用了 IOC 之后，则是容器主动地将资源推送给它所管理的组件，组件所要做的仅是选择一种合适的方式来接受资源。 这种行为也被称为查找的被动形式。

DI(Dependency Injection) — IOC 的另一种表述方式：即组件以一些预先定义好的方式(例如: setter 方法)接受来自如容器的资源注入。相对于 IOC 而言，这种表述更直接。

DI注入方式

1. **setter注入（必须有setter方法）**

```
<!-- userService -->
<bean id="userService"
      class="com.qhcs.spring.ioc.bean.inject.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"></property>
</bean>
```

2. **constructor注入（有相应的构造方法）**

```
<bean id="userService"
      class="com.qhcs.spring.ioc.bean.inject.service.impl.UserServiceImpl">
    <!-- 构造器注入，name指构造器参数名，ref指引用的bean的id -->
    <!-- <constructor-arg name="userDao" ref="userDao"></constructor-arg> -->
    <!-- 构造器注入，type指构造器参数类型，ref指引用的bean的id -->
    <constructor-arg type="com.qhcs.spring.ioc.bean.inject.dao.UserDao"
                     ref="userDao"></constructor-arg>
    <!-- 构造器注入，index指构造器参数序号，ref指引用的bean的id -->
    <!-- <constructor-arg index="0" ref="userDao"></constructor-arg> -->
</bean>
```

3. 工厂注入

DI注入集合

(1) 注入list集合，使用<list>标签

(2) 注入set集合，使用<set>标签

(3) 注入map集合，使用<map>标签

(4) 注入properties集合，使用<props>标签

```
    <!-- 注入集合 -->
    <bean id="collectionBean" class="com.qhcs.spring.ioc.inject.collection.CollectionBean">
        <!-- 注入list集合 -->
        <property name="addressList">
            <list>
                <value>北京</value>
                <value>上海</value>
                <value>广州</value>
                <value>深圳</value>
                <value>北京</value>
                <!-- 空字符串 -->
                <value></value>
                <!-- null -->
                <null></null>
                <!-- 注入bean -->
                <ref bean="user"/>
            </list>
        </property>

        <!-- 注入set集合 -->
        <property name="addressSet">
            <set>
                <value>北京</value>
                <value>上海</value>
                <value>广州</value>
```

```
                <value>深圳</value>
                <value>广州</value>
                <!-- 注入bean -->
                <ref bean="user"/>
            </set>
        </property>

        <!-- 注入map集合 -->
        <property name="addressMap">
            <map>
                <entry key="bj" value="北京"></entry>
                <entry key="sh" value="上海"></entry>
                <entry key="gz" value="广州"></entry>
                <entry key="sz" value="深圳"></entry>
                <entry key="sh" value="上海-虹口区"></entry>
                <entry key="guangzhou" value="广州"></entry>
                <!-- 注入bean -->
                <entry key="Kobe" value-ref="user"></entry>
            </map>
        </property>

        <!-- 注入properties集合 -->
        <property name="addressProp">
            <props>
                <prop key="bj">北京</prop>
                <prop key="sh">上海</prop>
                <prop key="gz">广州</prop>
                <prop key="sz">深圳</prop>
                <prop key="sh">上海-浦东区</prop>
                <prop key="guangzhou">广州</prop>
            </props>
        </property>
    </bean>
```

自动装配

自动装配是指不需要指定bean之间的关系，让容器自动配置，减少编写一个大的基于 Spring 的应用程序的 XML 配置的数量。

自动装配方式

no	**这是默认的设置，它意味着没有自动装配，你应该使用显式的bean引用来连线。你不用为了连线做特殊的事。在依赖注入章节你已经看到这个了。**
byName (beans-autowiring/spring-autowiring-byname.md)	由属性名自动装配。Spring 容器看到在 XML 配置文件中 bean 的自动装配的属性设置为 byName。然后尝试匹配，并且将它的属性与在配置文件中被定义为相同名称的 beans 的属性进行连接。
byType (beans-auto-wiring/spring-autowiring-byType.md)	由属性数据类型自动装配。Spring 容器看到在 XML 配置文件中 bean 的自动装配的属性设置为 byType。然后如果它的匹配配置文件中的一个确切的 bean 名称，它将尝试匹配和连接属性的类型。如果存在不止一个这样的 bean，则一个致命的异常将会被抛出。
constructor (beans-auto-wiring/spring-autowiring-by-Constructor.md)	类似于 byType，但该类型适用于构造函数参数类型。如果在容器中没有一个构造函数参数类型的 bean，则一个致命错误将会发生。
autodetect	Spring首先尝试通过 constructor 使用自动装配来连接，如果它不执行，Spring尝试通过 byType 来自动装配。

基于注解形式的自动装配

(1) @Component：将类注解成由spring容器管理的组件

① @Controller：该注解作用跟@Component一样，一般注解在控制类上，比如UserController.java

② @Service：该注解作用跟@Component一样，一般注解在服务类上，比如UserServiceImpl.java

③ @Repository：该注解作用跟@Component一样，一般注解在数据访问类上，比如UserDaoImpl.java

(2) @Autowired：相当于byType的自动装配，可用于注解在属性、构造方法、setter方法

(3) @Qualifier：与@Autowired结合使用，指定多个相同类型的bean中的一个

(4) @Resource：相当于byName的方式，可用于注解属性

AOP

概念

Aspect	**一个模块具有一组提供横切需求的 APIs。例如，一个日志模块为了记录日志将被 AOP 方面调用。应用程序可以拥有任意数量的方面，这取决于需求。**
Join point	在你的应用程序中它代表一个点，你可以在插件 AOP 方面。你也能说，它是在实际的应用程序中，其中一个操作将使用 Spring AOP 框架。
Advice	这是实际行动之前或之后执行的方法。这是在程序执行期间通过 Spring AOP 框架实际被调用的代码。
Pointcut	这是一组一个或多个连接点，通知应该被执行。你可以使用表达式或模式指定切入点正如我们将在 AOP 的例子中看到的。
Introduction	引用允许你添加新方法或属性到现有的类中。
Target object	被一个或者多个方面所通知的对象，这个对象永远是一个被代理对象。也称为被通知对象。
Weaving	Weaving 把方面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象。这些可以在编译时，类加载时和运行时完成。

实现方式

1. jdk代理(Proxy)：常用的，代理接口形式
2. Cglib：第三方实现方式，代理非接口形式

通知

前置通知	**在一个方法执行之前，执行通知。**
后置通知	在一个方法执行之后，不考虑其结果，执行通知。
返回后通知	在一个方法执行之后，只有在方法成功完成时，才能执行通知。
抛出异常后通知	在一个方法执行之后，只有在方法退出抛出异常时，才能执行通知。
环绕通知	在建议方法调用之前和之后，执行通知。

基于xml方式实现AOP

```
<!-- 被代理的目标 -->
<bean id="target" class="com.qhcs.spring.aop.xml.Target"></bean>

<!-- 通知 -->
<bean id="advice" class="com.qhcs.spring.aop.xml.Advice"></bean>

<!-- aop配置 -->
<aop:config>
    <!-- 切面，ref指定执行通知的bean的id -->
```

```
    <aop:aspect ref="advice">
        <!-- 切点，expression指定切点规则 -->
        <aop:pointcut expression="execution(* com.qhcs.spring.aop.xml.Target.*(..))"
id="pointcut"/>
        <!-- 前置通知，method指切点匹配是执行通知类的方法，pointcut-ref指定切点，一旦目标符合切点规则
执行method指定的通知类的方法 -->
        <aop:before method="before" pointcut-ref="pointcut"/>
    </aop:aspect>
</aop:config>
```

基于注解方式实现AOP

1. 编写Aop配置类，启用注解

```
// 声明该类为配置类
@Configuration
// 配置组件扫描路径
@ComponentScan("com.qhcs.spring.aop.annotation")
// 启用aop注解
@EnableAspectJAutoProxy
public class AOPConfig {

}
```

2. 编写被代理类

```
@Service
public class UserServiceImpl implements UserService {

    @Override
    public boolean addUser(User user) {
        if (user == null) {
            return false;
        }
        return true;
    }

}
```

3. 编写通知类

```
@Component
// 声明该类为切面
@Aspect
public class LogAdvice {

    @Pointcut(value = "execution(* com.qhcs.spring.aop.annotation.service.UserService.*
(..))")
```

```
    public void pointcut() {

    }

    @Before(value = "pointcut()")
    public void before(JoinPoint joinPoint) {
        // 获取被代理的对象
        Object target = joinPoint.getTarget();
        System.out.println("被代理目标对象: " + target);
        // 获取被切的方法信息
        Signature signature = joinPoint.getSignature();
        String name = signature.getName();
        System.out.println("被代理目标的方法: " + name);
        // 代理目标信息
        String typeName = signature.getDeclaringTypeName();
        System.out.println("被代理目标: " + typeName);
        Class clazz = signature.getDeclaringType();
        System.out.println("被代理的目标Class对象: " + clazz);
        // 获取参数值
        Object[] args = joinPoint.getArgs();
        for (Object object : args) {
            System.out.println("被代理目标方法的参数值: " + object);
        }
        System.out.println(">>>>>正在执行LogAdvice的before方法");
    }
}
```

切点表达式

(1) 最典型的切入点表达式时根据方法的签名来匹配各种方法:

execution(* com.atguigu.spring.ArithmeticCalculator.*(..)): 匹配 ArithmeticCalculator 中声明的所有方法,第一个 * 代表任意修饰符及任意返回值. 第二个 * 代表任意方法。.. 匹配任意数量的参数. 若目标类与接口与该切面在同一个包中, 可以省略包名。

execution(public * ArithmeticCalculator.*(..))): 匹配 ArithmeticCalculator 接口的所有公有方法.

execution(public double ArithmeticCalculator.*(..)): 匹配 ArithmeticCalculator 中返回 double 类型数值的公有方法

execution(public double ArithmeticCalculator.*(double, ..)): 匹配第一个参数为 double 类型并且返回值类型为 double的公开的方法, .. 匹配任意数量任意类型的参数

execution(public double ArithmeticCalculator.*(double, double)): 匹配参数类型为 double, double 类型并且返回值类型为double的公开的方法

(2) 在 AspectJ 中, 切入点表达式可以通过操作符 &&, ||, ! 结合起来.

```
@Pointcut("execution(* *.add(int, ..)) || execution(* *.sub(int, ..))")
private void loggingOperation(){}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint){
    log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}
```

Spring JDBC

Spring jdbc封装了jdbc，提供更高级更抽象的api，简化jdbc操作。

JdbcTemplate：Spring JDBC 框架的核心, JDBC 模板的设计目的是为不同类型的 JDBC 操作提供模板方法. 每个模板方法都能控制整个过程, 并允许覆盖过程中的特定任务. 通过这种方式, 可以在尽可能保留灵活性的情况下, 将数据库存取的工作量降到最低。

事务

事务就是一系列的动作, 它们被当做一个单独的工作单元. 这些动作要么全部完成, 要么全部不起作用，确保数据的完整性和一致性。

四大特性

1. 原子性(atomicity): 事务是一个原子操作, 由一系列动作组成。 事务的原子性确保动作要么全部完成要么完全不起作用。
2. 一致性(consistency): 一旦所有事务动作完成，事务就被提交。数据和资源就处于一种满足业务规则的一致性状态中。
3. 隔离性(isolation): 可能有许多事务会同时处理相同的数据， 因此每个事物都应该与其他事务隔离开来，防止数据损坏。
4. 持久性(durability): 一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果被写到持久化存储器中。

Spring管理事务两种方式

(1) 编程式事务：在业务的逻辑内编写管理事务的代码，入侵性高，事务管理代码和业务的代码混杂一起，不方便后期维护。步骤：

① 通过事务配置类DefaultTransactionDefinition（设置传播行为、隔离级别、超时时间等）创建DataSourceTansactionManager对象

② 通过DataSourceTansactionManager获取事务

③ 事务完成执行commit操作

④ 事务发生异常执行rollback操作

```
/**
* 编程式事务
*/
```

```java
@Override
public boolean transfer(String rollOutName, String rollInName, double amount) throws
RuntimeException {
    // 事务配置
    DefaultTransactionDefinition transactionDefinition = new DefaultTransactionDefinition();
    // 设置事务传播行为
    transactionDefinition.setPropagationBehaviorName("PROPAGATION_REQUIRED");
    // 设置事务隔离级别
    transactionDefinition.setIsolationLevelName("ISOLATION_DEFAULT");
    // 设置超时时间
    transactionDefinition.setTimeout(3);

    // 通过事务管理器获取事务，返回事务状态，后面提交/回滚事务需要根据事务状态提交/回滚
    TransactionStatus transactionStatus =
transactionManager.getTransaction(transactionDefinition);

    try {
        // 先查询用户id
        User rollOutUser = userDao.queryByName(rollOutName);
        User rollInUser = userDao.queryByName(rollInName);
        // 验证用户是否存在
        if (Objects.isNull(rollOutUser) || Objects.isNull(rollInUser)) {
            throw new RuntimeException("用户不存在! ");
        }

        // 查询账户
        Account rollOutAccount = accountDao.queryById(rollOutUser.getId());
        Account rollInAccount = accountDao.queryById(rollInUser.getId());
        // 验证账户是否存在
        if (Objects.isNull(rollOutAccount) || Objects.isNull(rollInAccount)) {
            throw new RuntimeException("账户不存在! ");
        }

        // 转出
        // 组装账户实体
        // 用户id
        rollOutAccount.setUserId(rollOutUser.getId());
        // 实际转账金额=转账金额+手续费
        rollOutAccount.setBalance(-(amount + amount * 0.07));
        // 更新余额
        accountDao.update(rollOutAccount);

        // 记录转账记录
        // 组装转账记录实体
        TransferLog rollOutTransferLog = new TransferLog();
        rollOutTransferLog.setAccountNo(rollOutAccount.getAccountNo());
        rollOutTransferLog.setAmount(amount);
        rollOutTransferLog.setFee(amount * 0.07);
        rollOutTransferLog.setType(1);
        // 插入转账记录
        transferLogDao.add(rollOutTransferLog);

        // 模拟转账过程中发生异常
```

```
        int i = 1 / 0;

        // 转入
        // 组装账户实体
        // 用户id
        rollInAccount.setUserId(rollInUser.getId());
        // 转账金额
        rollInAccount.setBalance(amount);
        // 更新余额
        accountDao.update(rollInAccount);

        // 记录转账记录
        // 组装转账记录实体
        TransferLog rollInTransferLog = new TransferLog();
        rollInTransferLog.setAccountNo(rollInAccount.getAccountNo());
        rollInTransferLog.setAmount(amount);
        rollInTransferLog.setFee(0);
        rollInTransferLog.setType(0);
        // 插入转账记录
        transferLogDao.add(rollInTransferLog);

        // 提交事务
        transactionManager.commit(transactionStatus);
    } catch (Exception e) {
        System.out.println("转账过程发生异常，回滚事务！");
        // 回滚事务
        transactionManager.rollback(transactionStatus);
        // 返回结果
        return false;
    }

    return true;
}
```

(2) 声明式事务：通过xml或者注解配置方式声明事务，底层通过AOP实现事务管理，将需要事务处理的业务集中管理，复用性强，让开发者更专注于业务逻辑处理，方便维护。

```
<!-- 配置TransactionManager -->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- 配置事务通知 -->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- 事务属性，哪些方法应用事务 -->
    <tx:attributes>
        <!-- get、find、query开头的方法是查询操作，不需要应用事务， read-only表示只读不应用事务-->
        <tx:method name="get*" read-only="true"/>
        <tx:method name="query*" read-only="true"/>
        <tx:method name="find*" read-only="true"/>
        <!-- 其余方法，比如add、insert、update、modify、delete、remove等开头的方法是增删改操作，需要
应用事务 -->
```

```
        <!-- 当应用事务的方法发生RuntimeException才会回滚，否则通过rollback-for来制定异常回滚 -->
        <!--             <tx:method name="*" rollback-for="FileNotFoundException"/> -->
        <!-- isolation设置隔离级别，timeout设置超时时间表示当事务超过规定时间还没完成时退出事务，
propagation设置传播行为-->
        <tx:method name="*" isolation="READ_COMMITTED" timeout="60" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>

<!-- 配置aop，主要配置切点表达式 -->
<aop:config>
    <aop:pointcut id="transferService" expression="execution(*
com.qhcs.spring.transaction.service..*Service.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="transferService"/>
</aop:config>
```

注意：声明式事务必须在service方法发生RuntimeExection(运行期异常)才会回滚，除非指定异常

Spring事务隔离级别

(1) TransactionDefinition.ISOLATION_DEFAULT：这是默认的隔离级别

(2) TransactionDefinition.ISOLATION_READ_COMMITTED（读已提交）：表明能够阻止误读；可以发生不可重复读和虚读，Oracle默认的隔离级别

(3) **TransactionDefinition.ISOLATION_READ_UNCOMMITTED（读未提交）**：表明可以发生误读、不可重复读和虚读

(4) **TransactionDefinition.ISOLATION_REPEATABLE_READ（可重复读）**：表明能够阻止误读和不可重复读；可以发生虚读，Mysql默认的隔离级别

(5) **TransactionDefinition.ISOLATION_SERIALIZABLE（序列化）**：表明能够阻止误读、不可重复读和虚读

Spring事务传播行为类型

(1) TransactionDefinition.PROPAGATION_MANDATORY：支持当前事务；如果不存在当前事务，则抛出一个异常

(2) TransactionDefinition.PROPAGATION_NESTED：如果存在当前事务，则在一个嵌套的事务中执行

(3) TransactionDefinition.PROPAGATION_NEVER：不支持当前事务；如果存在当前事务，则抛出一个异常

(4) TransactionDefinition.PROPAGATION_NOT_SUPPORTED：不支持当前事务；而总是执行非事务性

(5) **TransactionDefinition.PROPAGATION_REQUIRED**：支持当前事务；如果不存在事务，则创建一个新的事务，默认

(6) **TransactionDefinition.PROPAGATION_REQUIRES_NEW**：创建一个新事务，如果存在一个事务，则把当前事务挂起

(7) **TransactionDefinition.PROPAGATION_SUPPORTS**：支持当前事务；如果不存在，则执行非事务性，比较常用

(8) TransactionDefinition.TIMEOUT_DEFAULT：使用默认超时的底层事务系统，或者如果不支持超时则没有