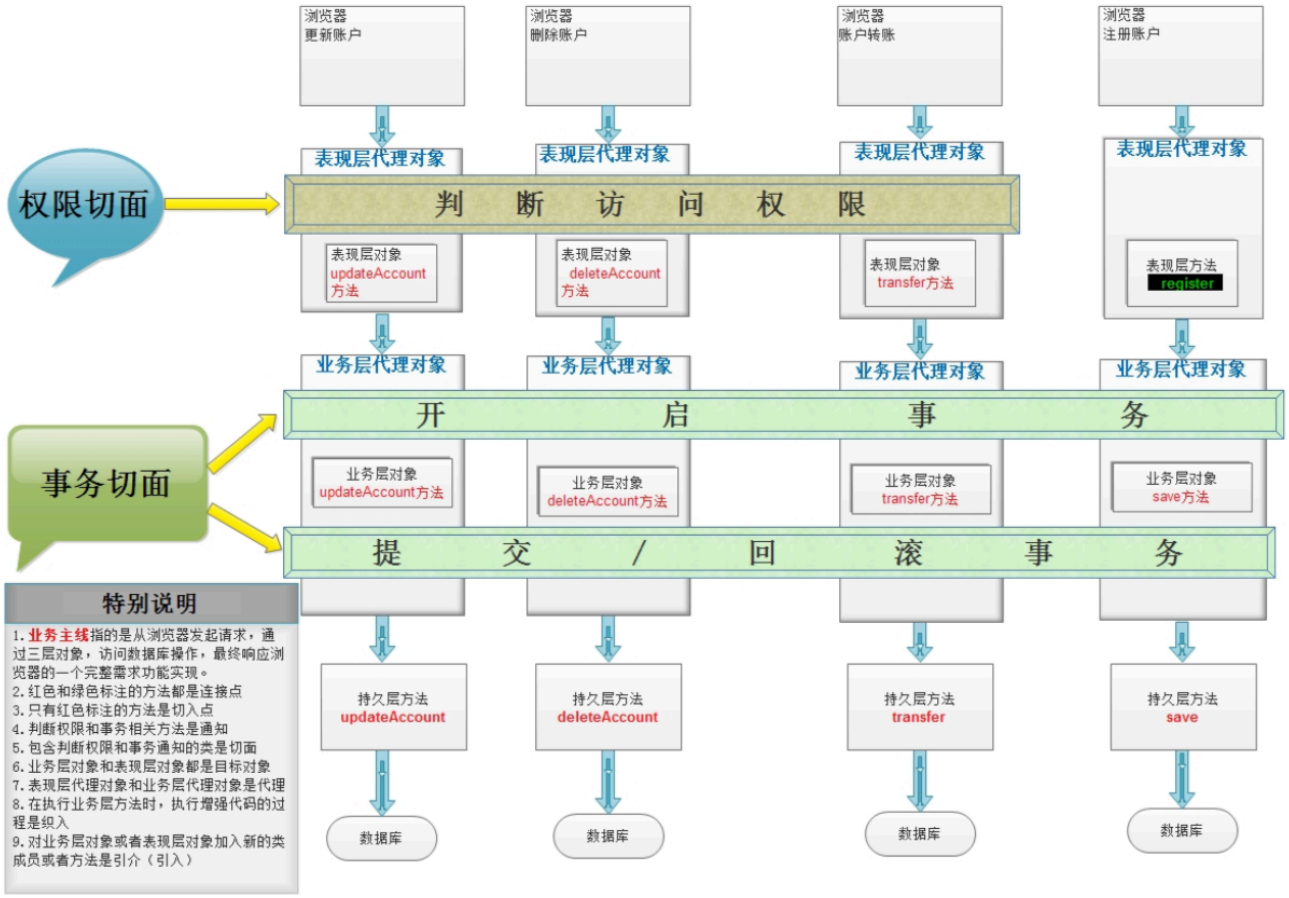
第六部分 Spring AOP应用

1. AOP 相关术语

AOP本质：在不改变原有业务逻辑的情况下，增强横切逻辑，横切逻辑往往是：权限校验代码、日志代码、事务控制代码、性能监控代码。

1.1 业务主线



1.2 AOP术语

- Joinpoint（连接点）：方法开始时、结束时、正常运行完毕时、方法异常时等这些特殊的时机点。项目中每个方法都有连接点，连接点是一种候选点（不一定每个方法都用AOP处理）。
- Pointcut（切入点）：指定需要被AOP思想影响的具体方法
- Advice（通知/增强）：
    - 指的是横切逻辑

    - 方位点(在某一些连接点上加入横切逻辑，那么这些连接点就叫做方位点，描述的是具体的特殊时机)
- Target（目标对象）：被代理的对象，即原始对象
- Proxy（代理）：一个类被AOP织入增强后，产生的代理类，即代理对象
- Weaving（织入）：指 把 增强 应用到 目标对象 然后来创建 代理对象 的过程。
    - Spring采用JDK动态代理
    - AspectJ采用另外的织入方式
- Aspect（切面）： 切面class包含相关的增强代码

  Aspect = 切入点 + 增强 = 切入点（锁定方法） + 方位点（锁定方法中的特殊时机） + 横切逻辑

最终目的：要在 哪个地方 插入 什么横切逻辑代码

2. Spring中AOP的代理选择

Spring 实现AOP思想使用的是动态代理技术。

默认情况下，Spring会根据被代理对象是否实现接口来选择使用JDK还是CGLIB。当被代理对象没有实现 任何接口时，Spring会选择CGLIB。当被代理对象实现了接口，Spring会选择JDK官方的代理技术，不过 我们可以通过配置的方式，让Spring强制使用CGLIB。

3. Spring中AOP的3种实现方式

需求：希望在某些目标方法中的某些特定位置，能够打印日志。故横切逻辑代码是打印日志。

目标方法：Service层的transfer方法

code base：`lagou-transfer-ioc-xml-anno`

3.1 纯xml

3.1.1 代码

新Code：`lagou-transfer-aop-xml`

Step 1：引入AOP的dependency

```
    <!-- 引入Spring aop支持 -->
```

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>

    <!-- 引入AspectJ，这是个第三方的AOP框架，
      Spring AOP实现时有用到其中的一些功能 -->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.13</version>
    </dependency>
```

Step 2：创建横切逻辑类 + xml中配置AOP

AspectJ 表达式来表示切入点的expression

```
<!-- 配置AOP，即术语落地 -->

    <!-- 横切逻辑类 -->
    <bean id="logUtils" class="com.lagou.edu.utils.LogUtils"/>

    <aop:config>

        <!-- Aspect = 切入点（某方法）+ 方位点（方法中某时机）+ 横切逻辑 -->

        <aop:aspect id="logAspect" ref="logUtils">
            <!-- 切入点 expression参数使用AspectJ表达式来锁定某方法 -->
            <aop:pointcut id="pt1"
                          expression="execution(public void
com.lagou.edu.service.impl.TransferServiceImpl.transfer(java.lang.String,java.lan
g.String,int))">
            </aop:pointcut>
            <!--  方位信息  -->
            <aop:before method="beforeMethod" pointcut-ref="pt1"/>
        </aop:aspect>

    </aop:config>
```

Step 3: Test

```
 @Test
    public void testXMLAop() throws Exception {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

        TransferService transferService =
applicationContext.getBean(TransferService.class);
        transferService.transfer("6029621011000", "6029621011001", 100);
    }
```

Test结果：

```
业务逻辑开始之前 执行...
开始执行转账业务逻辑
```

3.1.2 AOP 5种通知类型

所有通知类型标签都只能出现在 aop:aspect 标签内部。

3.1.2.0 标签层级example

```
<aop:config>

    <aop:aspect id="logAspect" ref="logUtils">
        <!-- 切入点 锁定某方法，expression 使用AspectJ表达式  -->
        <aop:pointcut id="pt1"
                      expression="execution(public void
com.lagou.edu.service.impl.TransferServiceImpl.transfer(java.lang.String,java.lan
g.String,int))"/>
        <!--  方位信息  -->
        <aop:before method="beforeMethod" pointcut-ref="pt1"/>
    </aop:aspect>

</aop:config>
```

3.1.2.1 前置通知

- 标签：<aop:before ... />
    - 属性：
        - method：指定前置通知在class中的方法名
        - pointcut = pointcut的expression表达式，即"execution(public xxxx“

        - pointcut-ref：指向切入点的id
- 执行时机：在切入点方法（业务逻辑方法）执行之前 就会执行
- 其他用途：前置通知可以获取 切入点方法（业务逻辑方法）传入的参数，并对其进行增强。

3.1.2.2 正常执行时 通知

- 标签：<aop:after-returning ... />
    - 属性
        - method：指定 正常执行时通知 在class中的方法名
        - pointcut = pointcut的expression表达式，即"execution(public xxxx“
        - point cut-ref：指向切入点的id

3.1.2.3 异常通知

- 标签：<aop:after-throwing ... />
    - 属性
        - method：指定 正常执行时通知 在class中的方法名
        - pointcut = pointcut的expression表达式，即"execution(public xxxx“
        - point cut-ref：指向切入点的id
- 执行时机：在切入点方法（业务逻辑方法）执行产生异常之后 执行。如果没有异常则不会执行。
- 细节：异常通知不仅可以获取切入点方法执行的参数，也可以获取切入点方法执行产生的异常信息

3.1.2.4 最终通知

- 标签：<aop:after ... />
    - 属性：
        - method：指定 最终通知 在class中的方法名
        - pointcut = pointcut的expression表达式，即"execution(public xxxx“
        - point cut-ref：指向切入点的id
- 执行时机：在切入点方法（业务逻辑方法）执行完成之后，切入点方法返回之前执行，无论切入点方法发生异常与否，都会执行。
- 细节： 最终通知执行时，可以获取到通知方法的参数。同时它可以做一些清理操作。

3.1.2.5 环绕通知

- 标签
    - 属性

        - method：指定 环绕通知 在class中的方法名
        - pointcut = pointcut的expression表达式，即"execution(public xxxx“
        - point cut-ref：指向切入点的id
- 特别说明

3.1.3 环绕通知切勿与另外4种通知混用

```
/*
     * 环绕通知
     * 1. 可以控制原有业务逻辑是否执行
     * 2. 可以替代另外4种通知方式
     * */
    public Object aroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        System.out.println("环绕通知中的 beforemethod ...");

        Object result = null;
        try {
            // 类似于 method.invoke，如果没写这句话，那么原有业务逻辑就不会被执行
            result = proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());
        }catch (Exception e) {
            System.out.println("环绕通知中的 exceptionmethod ...");
        }finally {
            System.out.println("环绕通知中的 aftermethod ...");
        }

        return result;
    }
```

```
<!--  使用环绕通知，就不再使用另外四种通知  -->
<aop:around method="aroundMethod" pointcut-ref="pt1"/>
```

3.2 xml + annotation

3.2.0 AspectJ 语法

“*” 来代替pkg、方法名字或者参数的类型。

“..” 来表示参数可以是0个或者多个

3.2.1 用@Component，@Aspect来标识横切逻辑类

3.2.2 @Pointcut 创建 pointcut 方法，名字是“id”

```
@Component
@Aspect
public class LogUtils {

    @Pointcut("execution(* com.lagou.edu.service.impl.TransferServiceImpl.*
(..))")
    public void pt1() {

    }
```

3.3 纯annotation

用注解替换掉xml中的：

```
<aop:aspectj-autoproxy/>
```

在横切逻辑类上加上@EnableAspectJAutoProxy 即可。

```
@EnableAspectJAutoProxy     // 开启Spring对注解AOP的支持
@Component
@Aspect
public class LogUtils {
```

4. Spring 声明式事务的支持

编程式事务：在业务代码中添加事务控制代码

声明式事务：通过xml或者注解配置的方式达到事务控制的目的

4.1 事务回顾

4.1.1 概念

事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不成功。从而确保了数据的准确与安全。

4.1.2 四大特性

- 原子性（Atomicity）：从操作的角度来描述，事务中的各个操作要么全部都成功，要么全部都失败。
- 一致性（Consistency）：从数据的角度，收发双方都始终保持一致（此消彼必长）。
- 隔离性（Isolation）：针对并发情况，每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。
- 持久性（Durability）：指一个事务一旦被提交，它对数据库中数据的改变就是永久性的。接下来即使数据库发生故障也不应该对其有任何影响。

4.1.3 事务的隔离级别

4.1.3.1 三大并发问题

如果不考虑事务的隔离级别，可能会出现一些**并发问题**，例如：

- 脏读：一个线程中的事务读到了另外一个线程中**未提交**的数据
  - 例子：事务1给员工涨工资2000，但是事务1尚未被提交，员工发起事务2查询工资，发现工资涨了2000 块钱，读到了事务1尚未提交的数据
- 不可重复读：针对Update操作。一个线程中的事务读到了另外一个线程中已经提交 **Update** 之后的数据，导致前后内容不一样。
  - 例子：员工A发起事务1，查询工资，工资为1w，此时事务1尚未关闭。财务人员发起了事务2，给员工A张了2000块钱，**并且提交了事务2**。员工A通过事务1再次发起查询请求，发现工资为1.2w，原来读出来1w读不到了。
- 虚读（幻读）：针对Insert或者Delete操作。一个线程中的事务读到了另外一个线程中已经提交 **Insert** 或者 **Delete** 之后的数据，导致前后条数不一样。
  - 例子：事务1查询所有工资为1w的员工的总数，查询出来了10个人，此时事务尚未关闭。事务2财务人员发起，新来员工，工资1w，向表中插入了2条数据，并且提交了事务。事务1再次查询工资为1w的员工个数，发现有12个人，见了鬼了。

4.1.3.2 处理措施：事务的四种隔离级别（安全级别最高到最低）

- Serializable（串行化）：可避免脏读、不可重复读、虚读的发生。事务一个一个处理，无并行而言。【最高】
- Repeatable read（可重复读）：可避免脏读、不可重复读情况的发生，**但 幻读 可能发生**，**因为其针对的是Update操作**，会对要Update的record进行加锁。
- Read committed（读已提交）：可避免脏读。但 不可重复读 和 幻读 一定会发生，因为只要有提交就能被读。
- Read uncommitted（读未提交）：以上问题均无法解决。

实际工程中其实不会在代码层面做太多干预，因为很多数据库自身就有隔离级别的设置。例如 MySQL的默认隔离级别是:REPEATABLE READ。

查询当前使用的隔离级别：select @@tx_isolation;

设置MySQL事务的隔离级别：set session transaction isolation level xxx;

4.1.4 事务的传播行为

事务往往在service层进行控制，如果出现service层方法A调用了另外一个service层方法B，A和B方法本身都已经被添加了事务控制，那么A调用B的时候，就需要进行事务的一些协商，这就叫做事务的传播行为。

A调用B，我们站在**B的角度**来观察来定义事务的传播行为（只需了解前两个常用行为即可）：

PROPAGATION_REQUIRED	**如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。**
PROPAGATION_SUPPORTS	**支持当前事务，如果当前没有事务，就以非事务方式执行。**
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

PROPAGATION_SUPPORTS 常用于查询的场景，如果上下文中有事务，就把查询加进去，若没有，不加也行，因为查询和其他事务没什么太大关系。

4.2 Spring中事务的API

PlatformTransactionManager Interface 定义了Spring事务的一些规范：

```
public interface PlatformTransactionManager {

  // 获取事务状态信息
  TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
throws TransactionException;

  // 提交事务
  void commit(TransactionStatus status) throws TransactionException;

  // 回滚事务
  void rollback(TransactionStatus status) throws TransactionException;

}
```

【注意】Spring本身并不支持事务实现，只是负责提供标准，应用层支持什么样的事务，需要提供具体实现。、Spring框架中，也内置了一些具体的策略 （DataSourceTransactionManager，HibernateTransactionManager 等）

Spring JDBCTemplate（数据库操作工具）、Mybatis（mybatis-spring.jar）用的是DataSourceTransactionManager。而Hibernate框架用的是HibernateTransactionManager。

DataSourceTransactionManager归根结底是横切逻辑代码，声明式事务要做的就是使用AOP思想（这里用动态代理来实现）来将事务控制逻辑织入到业务逻辑代码中。

4.3 Spring 声明式事务的配置

Base code：`lagou-transfer-transaction-task1`

新code：`lagou-transfer-transaction-task3`

4.3.1 纯xml

Step 1: 改造Dao层

1. Rename JDBCAccountDaoImpl to JdbcTemplateDaoImpl 并且 改写query和update方法

Step 2：删除手写的事务处理代码

在Servlet中，绕过proxy直接去拿transferService，方便删掉proxy class、TransactionManager class、ConnectionUtils class这条workflow。

Step 3：配置Spring声明式事务

1. pom中引入4个dependency，这个代码里在Step 1 就做了

```
    <!--spring aop的jar包支持-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>

    <!--第三方的aop框架aspectj的jar-->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.13</version>
    </dependency>

    <!--引入spring声明式事务相关-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>
```

2. xml中配置：
    1. 添加aop约束 和 tx约束（tx有一些针对事务的标签）
    2. 添加bean, tx:advice 和 aop:advisor

```
    <!-- Spring声明式事务。声明式事务就是一个aop，只不过有些标签不一样而已 -->
    <!-- 横切逻辑 -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <constructor-arg name="dataSource" ref="dataSource"/>
    </bean>

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!-- 定制事务细节、传播行为、隔离级别等 -->
        <tx:attributes>
```

```
        <!-- 拦截到某个方法后（*表示 一般性配置，即针对增删改查所有方法），定制他的隔离属性，isolation为Default即默认用数据库自己那套
            rollback-for 即为了谁而回滚，no-rollback-for 即不为谁回滚
            timeout 是秒为单位，= -1 即不设定timeout -->
             <tx:method name="*" read-only="false" propagation="REQUIRED" isolation="DEFAULT" timeout="-1"/>
             <!-- 针对查询类方法，添加覆盖性配置     -->
             <tx:method name="query*" read-only="true" propagation="SUPPORTS"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <!--  advice-ref 指向增强(Aspect) = 横切逻辑+方位    -->
        <aop:advisor advice-ref="txAdvice" pointcut="execution(* com.lagou.edu.service.impl.TransferServiceImpl.*(..))"/>
    </aop:config>
```

4.3.2 xml+annotation

原则：第三方引入 留在xml，自己写的转为annotation。所以这里只改造tx:advice和aop:config。

Step 1:

```
<!-- 开启声明式事务的注解驱动 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>
```

Step 2: 在指定实现类上加上注解@Transactional

注解@Transactional可以加在Interface上（会影响所有实现类），或者加在特定实现类上，或者加在某个特定方法上 【优先级最高】。一般加在实现类上即可。

4.3.3 纯annotation

参考code `lagou-transfer-ioc-anno`(github link)，在 com.lagou.edu folder中添加 SpirngConfiguration class，然后只要添加上注解 `@EnableTransactionManagement` 即可让Spring能够支持事务注解。

第七部分 Spring AOP 源码深度剖析

1. 代理对象创建

1.1 AOP基础用例准备

Step 1: 在spring-lagou-yu 的 build.gradle 文件中，引入第三方dependency：

```
dependencies {
    compile(project(":spring-context"))
    compile group: 'org.aspectj', name: 'aspectjweaver', version: '1.8.6'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

Step 2: 新建横切逻辑类 LogUtils，添加横切逻辑方法 beforeMethod。

```
public class LogUtils {

  public void beforeMethod() {
    System.out.println("前置通知（横切逻辑）");
  }
}
```

Step 3: 在LagouBean中，添加切入点方法 print()

```
public class LagouBean implements InitializingBean {

  ...

  public void print() {
    System.out.println("LagouBean的业务逻辑 print方法 执行");
  }
}
```

Step 4: 为了将横切逻辑方法 beforeMethod 插入到 LagouBean中的切入点方法print() 中，需要在xml中配置LogUtils bean 和aop:aspect

```
  <!-- 配置横切逻辑类 -->
  <bean id="logUtils" class="com.lagou.edu.LogUtils"/>

  <aop:config>
    <aop:aspect ref="logUtils">
      <aop:before method="beforeMethod" pointcut="execution(*
com.lagou.edu.LagouBean.print())"/>
    </aop:aspect>
  </aop:config>
```

Step 5: Test

```
  @Test
  public void testMyAOP() {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

    LagouBean result = applicationContext.getBean(LagouBean.class);
    result.print();
  }
```

Test result:

```
前置通知（横切逻辑）
LagouBean的业务逻辑 print方法 执行
```

1.2 时机点分析

在getBean() 之前，可以看到 LagouBean已经是一个 由Spring CGLIB 创建的代理对象了。所以可以断定，IoC容器在初始化原始对象的时候，也创建了其代理对象。

代理对象的创建肯定在原始对象创建之后。所以，先找到原始对象创建的入口： refresh方法中的finishBeanFactoryInitialization() -- 它实例化所有非懒加载的单例bean，同时，它还创建了原始对象的代理对象。

1.3 代理对象创建流程

调用栈：

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean(java.lang.String, java.lang.Object, org.springframework.beans.factory.support.RootBeanDefinition)

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#applyBeanPostProcessorsAfterInitialization

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#**postProcessAfterInitialization** （AbstractAutoProxyCreator是一个后置处理器，它的After方法 创建出了代理对象）

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#wrapIfNecessary

org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator#createProxy （这里把原始对象自定义的aop增强和Spring的通用拦截进行合并成 Advisor array，然后返回给代理对象让它承担一切。）

org.springframework.aop.framework.DefaultAopProxyFactory#createAopProxy （返回cglib动态代理）

org.springframework.aop.framework.CglibAopProxy#getProxy(java.lang.ClassLoader)

2. Spring声明式事务控制

2.1. @EnableTransactionManagement 引入了 TransactionManagementConfigurationSelector 类

通过import 引入了TransactionManagementConfigurationSelector：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {
```

TransactionManagementConfigurationSelector 的selectImport方法 向容器中导入/注册了2个重要组件：AutoProxyRegistrar 和 ProxyTransactionManagementConfiguration：

```
@Override
protected String[] selectImports(AdviceMode adviceMode) {
   /*
   * 该方法会返回一个String array给容器并完成注册，String array包含一批全类名
   * */
   switch (adviceMode) {
      case PROXY:
         /*
         * 会走这里。向容器中导入/注册了2个重要组件：AutoProxyRegistrar 和
ProxyTransactionManagementConfiguration
         * */
         return new String[] {AutoProxyRegistrar.class.getName(),
               ProxyTransactionManagementConfiguration.class.getName()};
      case ASPECTJ:
         return new String[] {determineTransactionAspectClass()};
      default:
         return null;
   }
}
```

2.1.1 分析 AutoProxyRegistrar 类

org.springframework.context.annotation.AutoProxyRegistrar#registerBeanDefinitions 方法中，通过以下方法引入了其他类：

```
AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
```

这个其他类是 org.springframework.aop.framework.autoproxy.InfrastructureAdvisorAutoProxyCreator， 它extends了 AbstractAdvisorAutoProxyCreator，所以是个PostProcessor class。

2.1.2 分析 ProxyTransactionManagementConfiguration 类

由于有`@Configuration`注解，所以是个配置类，用于注册各种bean：

```
@Configuration(proxyBeanMethods = false)
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class ProxyTransactionManagementConfiguration extends
AbstractTransactionManagementConfiguration {

  @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
```

```
  public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor(
      TransactionAttributeSource transactionAttributeSource,
TransactionInterceptor transactionInterceptor) {

    // 事务增强Advisor
    BeanFactoryTransactionAttributeSourceAdvisor advisor = new
BeanFactoryTransactionAttributeSourceAdvisor();
    // 注入 属性解析器
    advisor.setTransactionAttributeSource(transactionAttributeSource);
    // 注入 事务拦截器
    advisor.setAdvice(transactionInterceptor);
    if (this.enableTx != null) {
      advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
    }
    return advisor;
  }

  @Bean
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
  // 属性解析器 是个 AnnotationTransactionAttributeSource
  public TransactionAttributeSource transactionAttributeSource() {
    return new AnnotationTransactionAttributeSource();
  }

  @Bean
  @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
  // 事务拦截器
  public TransactionInterceptor transactionInterceptor(TransactionAttributeSource
transactionAttributeSource) {
    TransactionInterceptor interceptor = new TransactionInterceptor();
    interceptor.setTransactionAttributeSource(transactionAttributeSource);
    if (this.txManager != null) {
      interceptor.setTransactionManager(this.txManager);
    }
    return interceptor;
  }

}
```

- TransactionAttributeSource：是一个AnnotationTransactionAttributeSource。AnnotationTransactionAttributeSource内部拥有一个 事务注解解析器 的集合 `Set<TransactionAnnotationParser> annotationParsers`。其中，TransactionAnnotationParser是个Interface，这里用到的实现类是SpringTransactionAnnotationParser，主要作用是解析各种 @Transactional注解的属性：

```
public class SpringTransactionAnnotationParser implements
TransactionAnnotationParser, Serializable {

...

  protected TransactionAttribute
parseTransactionAnnotation(AnnotationAttributes attributes) {
    RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();

    Propagation propagation = attributes.getEnum("propagation");
    rbta.setPropagationBehavior(propagation.value());
    Isolation isolation = attributes.getEnum("isolation");
    rbta.setIsolationLevel(isolation.value());
    rbta.setTimeout(attributes.getNumber("timeout").intValue());
    rbta.setReadOnly(attributes.getBoolean("readOnly"));
    rbta.setQualifier(attributes.getString("value"));

    List<RollbackRuleAttribute> rollbackRules = new ArrayList<>();
    for (Class<?> rbRule : attributes.getClassArray("rollbackFor")) {
      rollbackRules.add(new RollbackRuleAttribute(rbRule));
    }
    for (String rbRule : attributes.getStringArray("rollbackForClassName")) {
      rollbackRules.add(new RollbackRuleAttribute(rbRule));
    }
    for (Class<?> rbRule : attributes.getClassArray("noRollbackFor")) {
      rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
    }
    for (String rbRule : attributes.getStringArray("noRollbackForClassName"))
{
      rollbackRules.add(new NoRollbackRuleAttribute(rbRule));
    }
    rbta.setRollbackRules(rollbackRules);

    return rbta;
  }
  ...
}
```

- TransactionInterceptor：该类实现了MethodInterceptor Interface，这是个通用拦截。在真正创建代理对象之前，该通用拦截会与原始对象自定义aop增强合并成一个Advisor类array，最终返回给代理对象让它承担一起。对于原始对象的业务逻辑，TransactionInterceptor相当于他的一个横切逻辑，其中的invoke方法中的invokeWithinTransaction 方法会触发原有业务逻辑的调用（即增强了事务）：

```
public class TransactionInterceptor extends TransactionAspectSupport
implements MethodInterceptor, Serializable {
...

@Override
@Nullable
public Object invoke(MethodInvocation invocation) throws Throwable {
   // Work out the target class: may be {@code null}.
   // The TransactionAttributeSource should be passed the target class
   // as well as the method, which may be from an interface.
   Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass(invocation.getThis()) : null);

   // Adapt to TransactionAspectSupport's invokeWithinTransaction...
   return invokeWithinTransaction(invocation.getMethod(), targetClass,
invocation::proceed);
}
  ...
```