第一部分 Spring概述

1. Spring简介

1. 分层（Service层，Dao层等）的full-stack轻量级（有jar包有JVM容器环境就能跑，不依赖第三方软件）开源框架
2. 内核为IoC和AOP
3. 提供了展现层Spring MVC和业务层事务管理等应用技术
4. 能整合第三方框架和类库

2 Spring发展历程

- 1997年IBM提出EJB思想
- Rod Johnson（Spring之父）提出Spring雏形框架

3 优势

1. 方便解耦，简化开发

   通过IoC容器，将对象

2. AOP编程支持

   Aspect Oriented Programming，面向切面编程，可应付许多不容易用传统OOP实现的功能

3. Transaction的支持

   @Transactional 的底层是用AOP实现的

4. 方便测试

   有专门的测试模块，可以模块化地测试别的模块

5. 方便集成各种优秀框架（如Struts、Hibernate、Hessian、Quartz等）

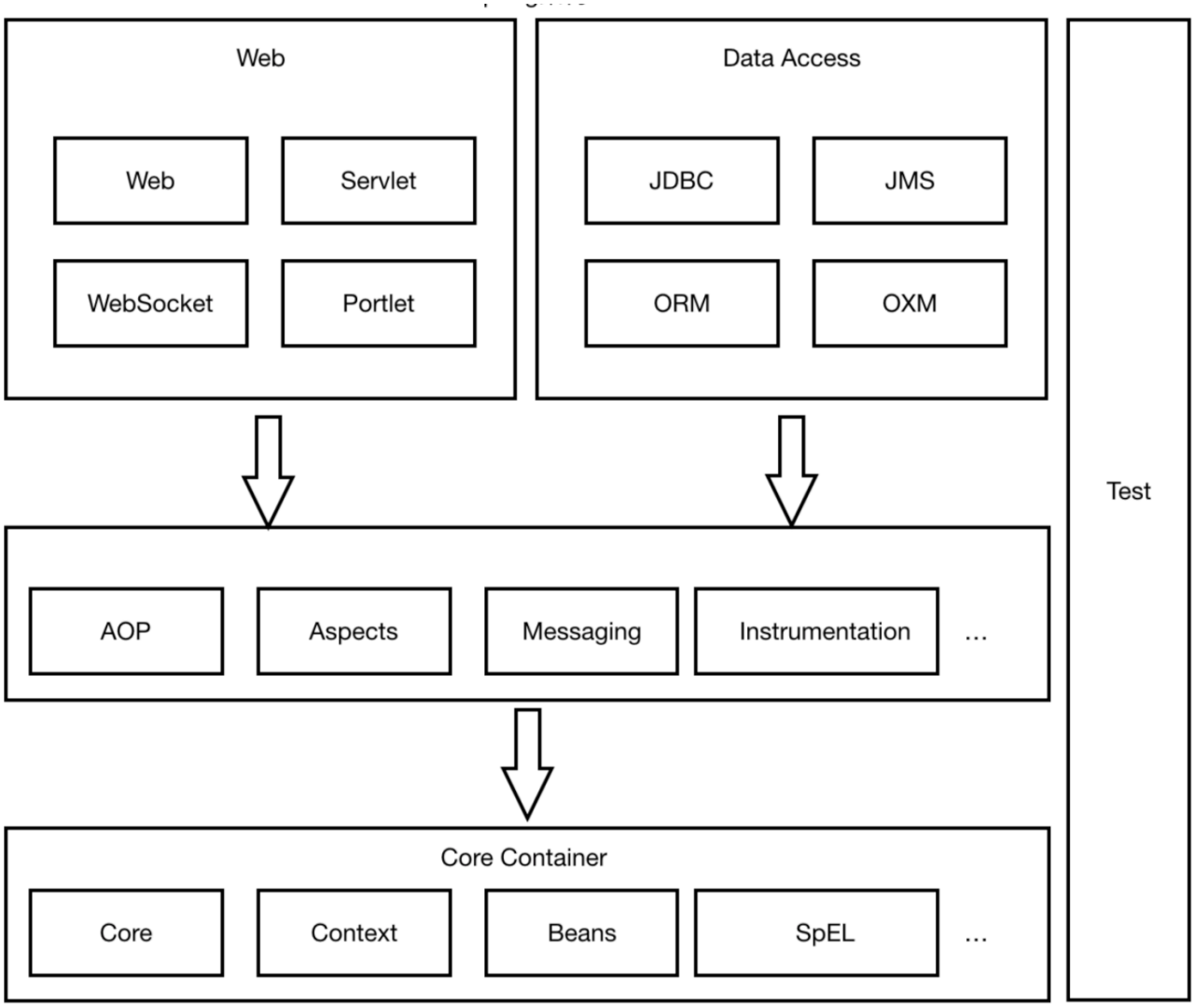
6. 降低JavaEE API的使用难度

   Spring对JavaEE API（如JDBC、JavaMail、远程调用等）进行了薄薄的封装层，使用

7. 源码是经典的Java学习范例

   优雅设计，结构清晰，Masterpiece。

4 核心结构



5 框架版本

JDK 8+ for Spring 5.x

我们使用：

- JDK 11.0.5
- IntelliJ IDEA 2019
- Maven 3.6.x

第二部分 核心思想

注意：IoC和AOP不是Spring提出的，在Spring之前就已经提出，不过更偏向理论化，Spring在技术层次把这两个实现做了非常好的实现（用Java语言）。

1 IoC

1.1 什么是IoC

IoC（Inversion of Control）是一个技术思想，不是技术实现。

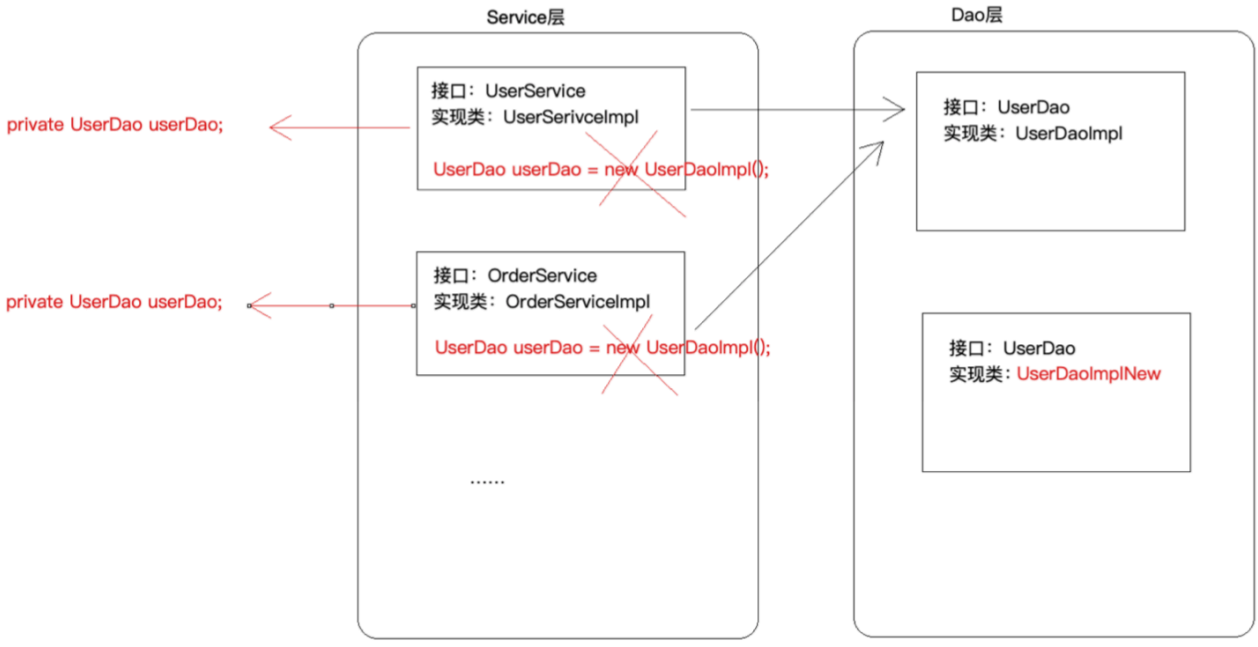
- 针对：Java开发领域对象的创建和管理
  - 传统方式：类A依赖于类B，就在类A中new一个B的对象来用
  - IoC思想：不用自己new，而是由IoC容器来实例化对象并存储、管理它，我们需要用什么对象，就直接去找IoC容器取。我们不再需要考虑对象的创建和管理
- 为什么叫IoC?
  - Control：指的是对象创建（实例化、管理）的权利
  - Inversion：控制权交给了外部环境（Spring框架IoC容器）了

1.2 IoC解决了什么问题

解决了对象之间的耦合问题。例如（下图场景）：

按传统方式，会将UserServiceImpl 和 UserDaoImpl 耦合，OrderServiceImpl 和 UserDaoImpl 耦合，如果此时create了一个新实现类 UserDaoImplNew在Dao层，那么在Service层，如果要改成用Dao层的新实现类，就要一个一个去把new出来的UserDaoImpl改成UserDaoImplNew，如果有一万个new，此改动会很麻烦且易出错。

用IoC的话，由于可以在IoC中管理要用哪个实现类去实现Interface，所以，在Service层，只需声明用哪个Interface即可，而不用管其背后的实现类。



1.3 IoC和DI的区别

DI（Dependency Injection）和IoC针对的是同一件事情（即对象的创建和管理），只是角度不同。

DI强调：容器为了把某对象实例化成功，会把其依赖的其他对象都注入给它。

IoC强调：对象实例化和管理的权利都交个了容器。

DI的概念在IoC之后。

2. AOP

2.1 什么是AOP

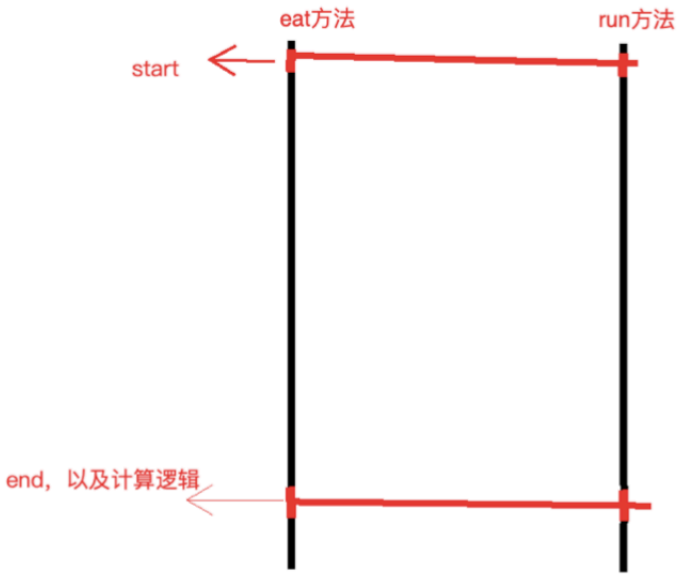
AOP（Aspect Oriented Programming 面向切面编程）是OOP的延续，从OOP说起。OOP三大特征：封装、继承和多态。OOP是一种垂直纵向的继承体系，子类extends父类的属性和方法等，从而解决了代码重复的问题。

- 但是，如果在顶级父类class中的多个方法内出现了有着相同功能代码（比如监控每一个方法的性能等），OOP就解决不了。

```java
*/
public class Animal {

    private String height; // 高度
    private float weight;  // 体重

    public void eat(){

        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can eat...");

        // 性能监控代码
        long end = System.currentTimeMillis();
        System.out.println("执行时长: " + (end-start)/1000f + "s");
    }

    public void run(){
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can run...");

        // 性能监控代码
        long end = System.currentTimeMillis();
        System.out.println("执行时长: " + (end-start)/1000f + "s");
    }

}
```

- 所以，这里先引入横切逻辑代码：

在多个纵向（顺序）执行的流程中出现的相同子流程代码（即这里的监控性能），即为横切逻辑代码。其使用场景很有限，一般是事务的控制（打开、提交、close、回滚等）、权限校验、日志打印等。



横切逻辑代码也存在问题：

1. 横切逻辑代码重复问题
2. 横切逻辑代码和业务代码混在一起，代码臃肿，不便维护。

- 所以，这里再引入AOP：

AOP提出横向抽取机制，将横切逻辑从业务逻辑中拆分出去，然后再合进来。这里的合不是代码的整合，而是最终执行效果的整合。AOP运用动态代理完成了效果整合（之后讲）。

2.2 AOP解决了什么问题

在不改变原有业务逻辑的情况下，将横切逻辑代码解耦出来，避免横切逻辑代码的重复。

2.3 为什么叫AOP（面向切面编程）

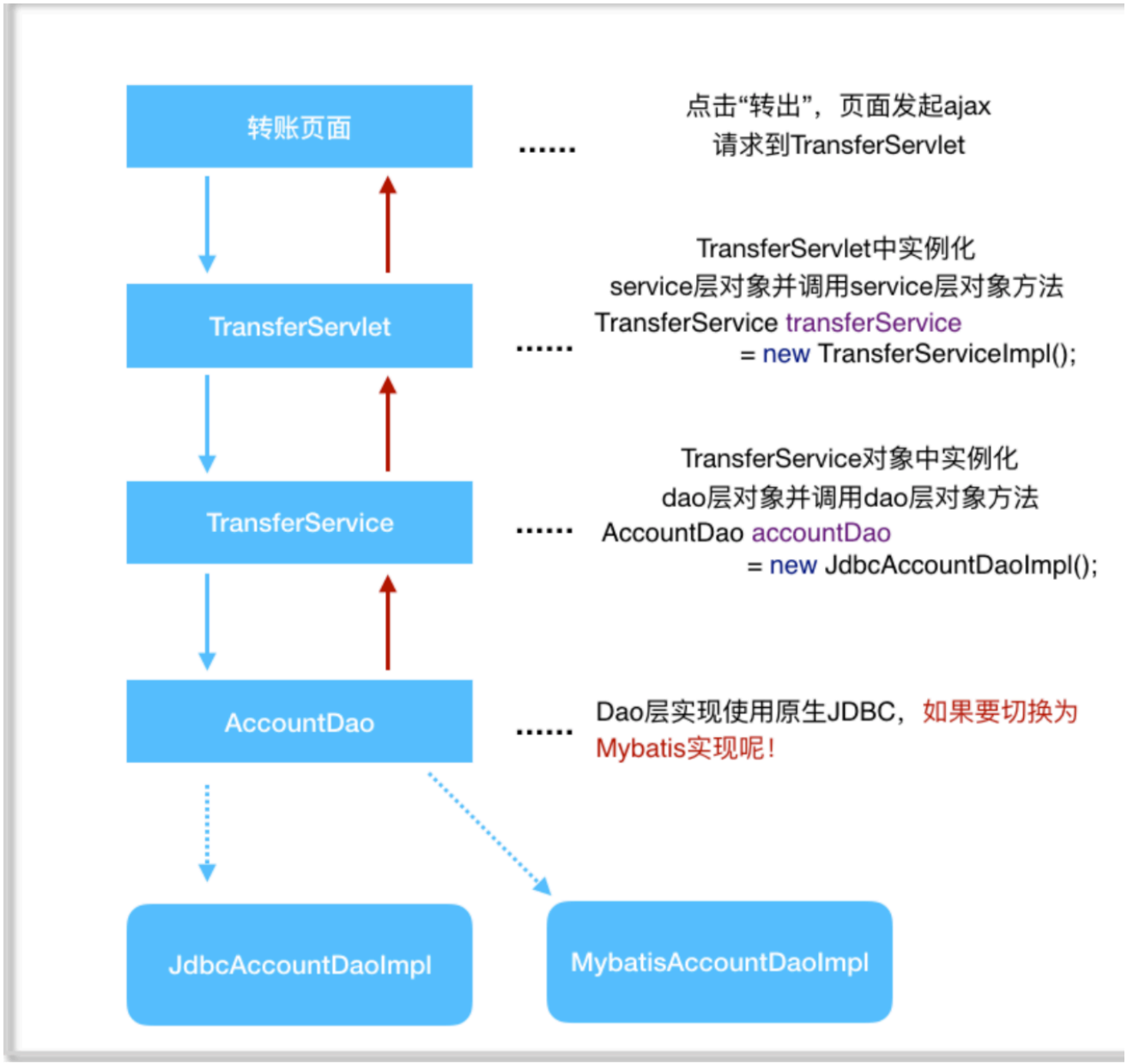
【切】：即横切逻辑。原有业务逻辑不能动，所以只能面向横切逻辑。

【面】：横切逻辑代码往往要影响很多个方法，每个方法都如同一个点，点连起来成线，线动起来成面，所以可见其影响范围之广。

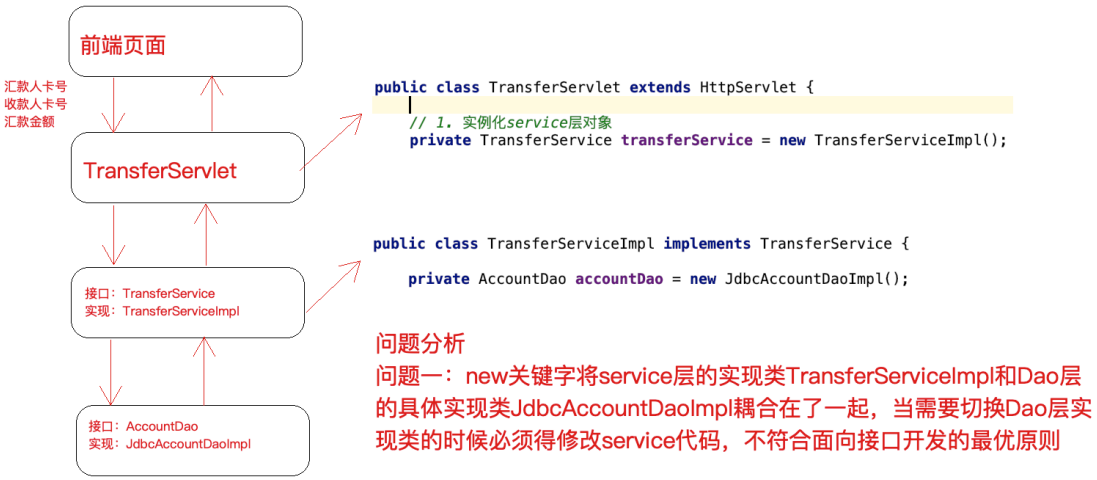
第三部分 手写实现IoC和AOP

开`lagou-transfer`project，然后点击左边maven，去`Plugins/tomcat7`下面，点击tomcat7:run来启动tomcat Server，启动后就可以browser里用`localhost:8080`看到界面。

1. 银行转账案例代码调用关系



2. 银行转账案例代码问题分析



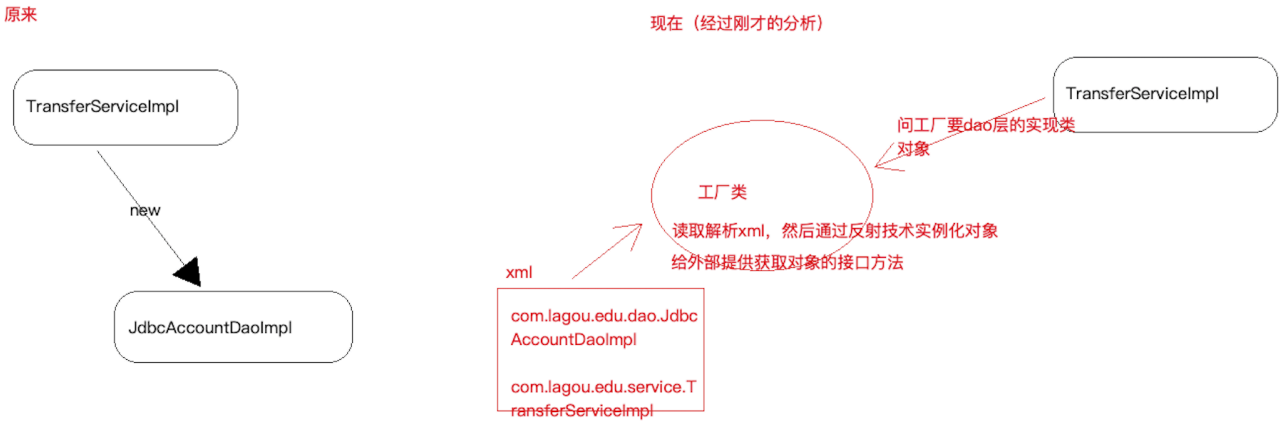
关于问题一，面向接口开发即只想new一个dao interface，而不是它的某个特定实现类。

3. 问题解决思路

3.1 针对问题一 （源于new关键字）

思考：

- 实例化对象的方式，除了new，还有什么技术？答：反射。Class.forName("全类名 com.lagou.edu.dao.JdbcAccountDaoImpl")，为了方便修改，把全类名放在xml文件中（xml不会重复被编译）。
- 使用工厂设计模式来通过反射技术创建对象从而解耦合。
    - 工厂类做2件事情：（1）读取解析xml，通过反射实例化对象 （2）给外部提供获取对象的接口方法
-



3.1.1 写代码(见`lagou-transfer-beanfactory`)

第一步：添加xml文件`beans.xml`

```
<beans>
    <bean id="accountDao" class="com.lagou.edu.dao.impl.JdbcTemplateDaoImpl">
        <property name="ConnectionUtils" ref="connectionUtils"/>
    </bean>
    <bean id="transferService"
class="com.lagou.edu.service.impl.TransferServiceImpl">
        <!--set+ name 之后锁定到传值的set方法了，通过反射技术可以调用该方法传入对应的值-->
        <property name="AccountDao" ref="accountDao"></property>
    </bean>

</beans>
```

第二步：添加工厂类`com.lagou.edu.factory.BeanFactory`做2件事：

```
public class BeanFactory {
    private static Map<String,Object> map = new HashMap<>();  // 存储对象

    // 任务一：读取解析xml，通过反射技术实例化对象并且存储待用（map集合）
    static {
        // 加载xml
        InputStream resourceAsStream =
BeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");
        // 解析xml
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read(resourceAsStream);
            Element rootElement = document.getRootElement();
            List<Element> beanList = rootElement.selectNodes("//bean"); // 两个杠
可以找所有<bean>标签而不考虑层级
            for (int i = 0; i < beanList.size(); i++) {
                Element element =  beanList.get(i);
                // 处理每个bean元素，获取到该元素的id 和 class 属性
                String id = element.attributeValue("id");        // accountDao
                String clazz = element.attributeValue("class");  //
com.lagou.edu.dao.impl.JdbcAccountDaoImpl
                // 通过反射技术实例化对象
                Class<?> aClass = Class.forName(clazz);
                Object o = aClass.newInstance();  // 实例化之后的对象
```

```
                // 存储到map中待用
                map.put(id,o);
            }
        }catch {
            ...
        }
    }

    // 任务二：对外提供获取实例对象的接口（根据id获取）
    public static  Object getBean(String id) {
        return map.get(id);
    }
```

第三步：

- 在ServiceImpl里面，调用BeanFactory方法获得dao：

```
public class TransferServiceImpl implements TransferService {

    //private AccountDao accountDao = new JdbcAccountDaoImpl();

    private AccountDao accountDao = (AccountDao)
BeanFactory.getBean("accountDao");
```

- 在Servlet里面，调用BeanFactory方法获得dao：

```
public class TransferServlet extends HttpServlet {
  // 1. 实例化service层对象
  //private TransferService transferService = new TransferServiceImpl();
  private TransferService transferService = (TransferService)
BeanFactory.getBean("transferService");
```

> servlet是暴露给外部容器来启动的，比如tomcat，jetty等

第四步：再优化成 只声明一个Interface，而不去getBean:

```
public class TransferServiceImpl implements TransferService {

   /*
     * 在某个对象中 new 出另一个类的对象：
     * 法一：直接new
     * 法二：getBean
```

```
     * 法三：构造函数传进来 或者 setter
     * */
    //private AccountDao accountDao = new JdbcAccountDaoImpl();
    //private AccountDao accountDao = (AccountDao)
BeanFactory.getBean("accountDao");

    private AccountDao accountDao;  // 最佳状态

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
```

并且，在BeanFactory里面，把各个bean实例化完成后，继续维护bean之间的依赖关系，即检查哪些对象需要传值进入其他的bean，根据property标签，来传入相应的bean对象。

3.2 针对问题二（缺少事务分析）

问题：没有添加事务控制时候，如果在转账过程中出现异常，那么可能会造成一方已经转了钱但另一方未收到钱 或者 一方转钱失败但另一方却收到了钱 等情况。这种inconsistency不是我们想要的。

3.2.1 深入分析：数据库事务控制归根结底是Connection的事务控制

connection.commit() -- 提交事务

connection.close() -- 回滚事务

而connect 的 autoCommit 属性默认为true。

3.2.2 Root cause

- 在TransferServiceImpl中，做了2次accountDao.updateAccountByCardNo分别为发钱方和收钱方，这2次update，由于connect的commit是自动的，即说明这2次update就使用了2个DB connect，所以，就不属于同一个事务控制了。
- 当前事务控制发生在了Dao层，而不是发生在Service层。

3.2.3 解决问题 （见`lagou-transfer-transaction`）

1. 让2次update（对发钱方和收钱方）使用同一个connection。让2次update属于同一个thread内执行调用，可以给当前线程绑定一个Connection，然后，和当前线程有关系的DB操作都去用这个绑定的Connection（从当前thread中拿）
    1. 在Utils pkg 中Create ConnectionUtils class。不过第一次使用时，由于当前thread没有

Connect，所以需要从连接池获取一个connection然后绑定到当前Connection上。

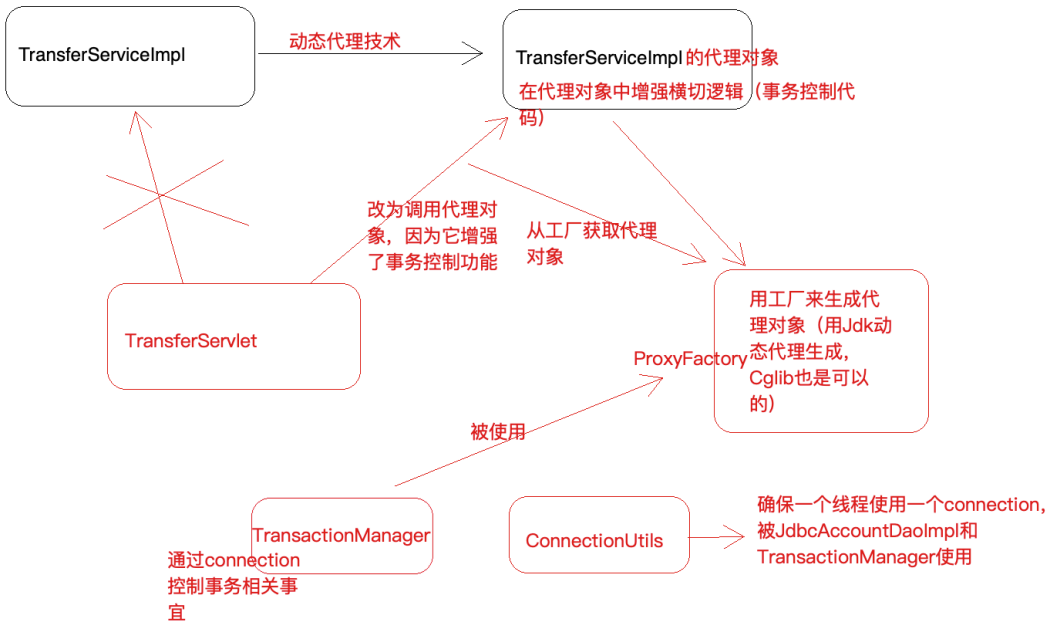
2. 把事务控制放在Service层
    1. Create一个饿汉单例模式的TransactionManager来负责事务的开启、提交和回滚。
    2. **注意：在Service transfer方法中，catch了Exception之后要throw 出去，这样上层（servlet）才能捕获到异常。如果这里不throw，虽然后台依然会报异常，但servlet由于拿不到异常，就会在UI上显示"转账成功"**

3.2.4 优化事务控制

- 考虑场景一：如果Service层中有成百上千个方法都需要进行事务管理，难道要在每个方法中逐一添加TransactionManager的方法来控制吗?
- 考虑场景二：这里把事务控制逻辑与业务逻辑混在了一起。如果事务逻辑很多很复杂，code可读性降低。（其实，对于某个类中的业务逻辑来说，其中的事务逻辑就是增强/Aspect逻辑）
- 解决方案：AOP + 动态代理模式

  创建ProxyFactory，生成Service的代理对象，并且让Servlet改用代理对象。这样一来，无论在Service中添加任何方法，都会走代理对象中的增强/切面逻辑

3.3 最终解决方案



3.4 优化最终解决方案：用beans.xml来管理class + 各个class之间的依赖关系

Step 1: 主Class添加setter，取消单例化

- TransanctionManager 和 JdbcAccountDaoImpl 依赖于 ConnectionUtils，所以可用setter来set 依赖的类。
- 取消单例的生成ConnectionUtils，因为使用了beans.xml来管理对象生成和对象之间的依赖关系。
    - ProxyFactory 依赖于TransactionManager，所以可用setter来set依赖的类
    - 取消单例的生成TransactionManager
    - 取消单例的生成ProxyFactory，因为beans.xml会通过反射机制生成ProxyFactory对象，其背后会用到ProxyFactory的无参constructor。

Step 2: 配置beans.xml

- 增加 ConnectionUtils, TransanctionManager，ProxyFactory 共3个bean
- 维护bean之间的依赖关系

```
<beans>
    <!--id标识对象，class是类的全限定类名-->
    <bean id="accountDao" class="com.lagou.edu.dao.impl.JdbcAccountDaoImpl">
        <property name="ConnectionUtils" ref="connectionUtils"></property>
    </bean>

    <bean id="transferService"
class="com.lagou.edu.service.impl.TransferServiceImpl">
        <!-- 这里会用class里的setter方法(set+Name)来传入ref指向的对象 -->
        <property name="AccountDao" ref="accountDao"></property>
    </bean>

    <!-- 优化最终解决方案：配置新增的3个bean -->
    <bean id="connectionUtils" class="com.lagou.edu.utils.ConnectionUtils">
</bean>

    <bean id="transactionManager" class="com.lagou.edu.utils.TransactionManager">
        <property name="ConnectionUtils" ref="connectionUtils"></property>
    </bean>

    <bean id="proxyFactory" class="com.lagou.edu.factory.ProxyFactory">
        <property name="TransactionManager" ref="transactionManager"></property>
    </bean>

</beans>
```

Step 3: Servlet 通过 BeanFactory 来获取 Service的代理对象

```
public class TransferServlet extends HttpServlet {

    // 首先，直接从BeanFactory中获取ProxyFactory的对象
    private ProxyFactory proxyFactory = (ProxyFactory)
BeanFactory.getBean("proxyFactory");
...
```

Bonus - Special Talks

Code 见`demo-others`。

1. 工厂设计模式

讲解：Vedio Task1-8 5:40

应用：第三部分3.1，BeanFactory

原理：

1. SimpleFactory：有一个Interface Noodles，只有一个工厂类，其可按照type来分别生产拉面、泡面、热干面

2. FactoryMethod：有一个Interface NoodlesFactory，工厂类都implements它，于是拉面工厂类只生产拉面，泡面工厂类只生产泡面，热干面工厂类只生产热干面

2. 单例模式

应用：第三部分3.2，ConnectionUtils，TransactionManager

原理：

1. 懒汉模式
2. 饿汉模式

   三步走：

   1. private Constructor
   2. static attribute = new Factory()
   3. static getInstance {return attribute}

3. 静态代理，动态代理

讲解：Vedio Task1-12 1:40，Vedio Task1-13 CGLIB

应用：第三部分3.2.4优化

静态代理原理

租客租房子，找了个中介（即代理人），把自己（的需求）传给中介。

对于租房业务，有相对应的租房中介。有别的业务，比如租车，则会有另外一个新的租车中介来handle。

JDK 动态代理原理

不需要显式地为每一个不同业务create一个新的中介，JDK自带Proxy类，用其来帮助某个委托对象生成它的代理对象，这个代理对象与委托对象同类。

然后，还可优化：通过工厂模式，创建单例化的ProxyFactory来生成不同委托对象的代理对象。这样，由于可以通过method知道当前是哪个接口的哪个方法，从而在invoke方法中，可针对性的添加增强逻辑。

CGLIB 动态代理

需要引入第三方jar包。其引入MethodInterceptor，通过Override intercept方法（4个参数：代理对象本身的reference， 委托对象的方法，args，当前执行方法的代理对象）。

在ProxyFactory 中，可以添加方法 getCglibProxy(委托对象) 来生成代理对象。

> JDK动态代理 vs. CGLIB 动态代理
>
> 在generate代理对象时，JDK动态代理需要传入委托对象的Interface，即委托对象是某个Interface的implementation class；
>
> 但CGLIB 动态代理不需要，即委托对象可以是个普通的class，而不一定是某个Interface的implementation class。