Spring 的帝国

之

Spring 框架

叩丁狼教育：任小龙

春水初生，春林初盛，春风十里，不如你！——冯唐

1. 第一章 Spring 并天下

1.1. 丑陋的代码

1.1.1.代码耦合度比较高

```
public class EmployeeServiceImpl implements IEmployeeService {
  private IEmployeeDAO dao;

  public EmployeeServiceImpl() {
    //创建依赖对象
    dao = new EmployeeDAOJdbcImpl();
  }
  public void save() {
    //TODO
  }
}
```

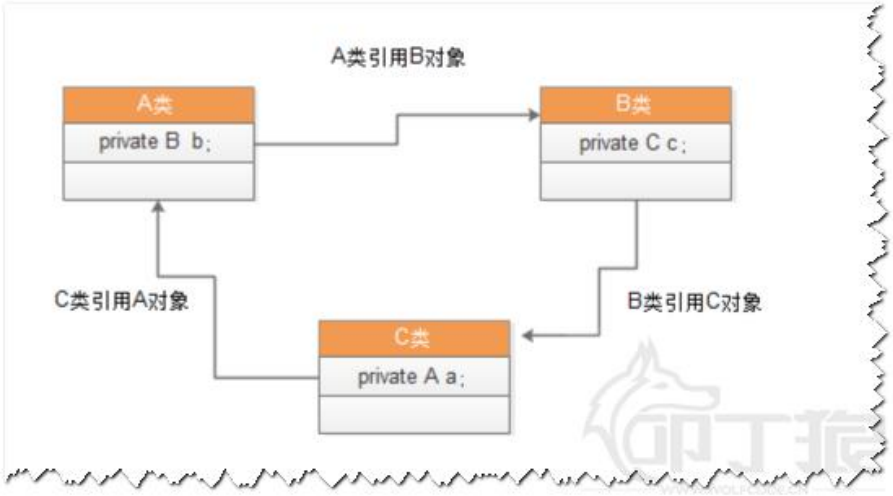
此时如果把 IEmployeeDAO 的实现类换成 EmployeeDAOMyBatisImpl，此时需要修改 EmployeeServiceImpl 的源代码，不符合开闭原则。

开闭原则：对于扩展是开放的，对于修改是关闭的。

1.1.2.循环依赖问题

循环依赖就是循环引用，指两个或多个 Bean 之间相互持有对方。

如在创建 A 类时，构造器需要 B 类，那将去创建 B，在创建 B 类时又发现需要 C 类，则又去创建 C，最终在创建 C 时发现又需要 A；从而形成一个环。



此时如何去管理对象的创建和相互的依赖关系？

1.1.3.控制事务繁琐

考虑一个应用场景：需要对系统中的某些业务方法做事务管理，拿简单的 save 和 update 操作举例。

没有加上事务控制的代码如下：

```
public class EmployeeServiceImpl implements IEmployeeService {
  public void save() {
    //保存操作
  }
}
```

修改源代码，加上事务控制之后：

```
public class EmployeeServiceImpl implements IEmployeeService {
  public void save() {
    //打开资源
    //开启事务
    try {
       //保存操作
       //提交事务
    } catch (Exception e) {
       //回滚事务
    } finally {
       //释放资源
    }
  }
}
```

此时我们发现，很明显有重复代码，这种重复明显是可以使用模板方法设计模式来消除的。

模板基类：

```
public class BaseServiceTemplate implements IEmployeeService {
  public void save() {
    //打开资源
    //开启事务
    try {
       this.doSave();
       //提交事务
    } catch (Exception e) {
       //回滚事务
    } finally {
       //释放资源
    }
  }
  protected void doSave() {
```

```
        //NOOP
      }
    }
```

子类代码:

```
    public class EmployeeServiceImpl extends BaseServiceTemplate implements IEmployeeService
{
      protected void doSave() {
        //保存操作
      }
    }
```

且先不论该设计也违背了开闭原则，假若我们不能更改 EmployeeServiceImpl 类的源代码，又该何去何从？

1.1.4. 第三方框架运用太麻烦

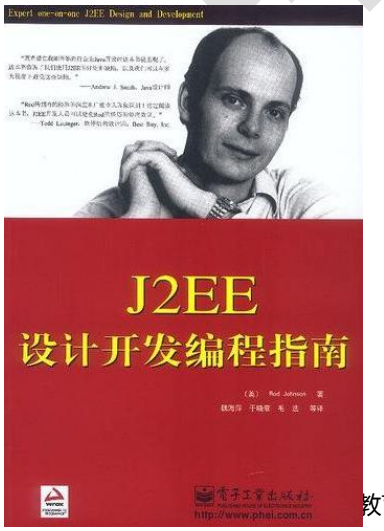
单独使用 MyBatis 框架的保存操作代码如下：

```
    Person p = new Person();
    SqlSession session = MyBatisUtil.getSession();
    PersonMapper personMapper = session.getMapper(PersonMapper.class);
    personMapper.insert(p);
    session.commit();
    session.close();
```

1.2. Spring 帝国

1.2.1. 帝国崛起-诞生

源于 Rod Johnson 在其著作《Expert one on one J2EE design and development》中阐述的部分理念和原型衍生而来。



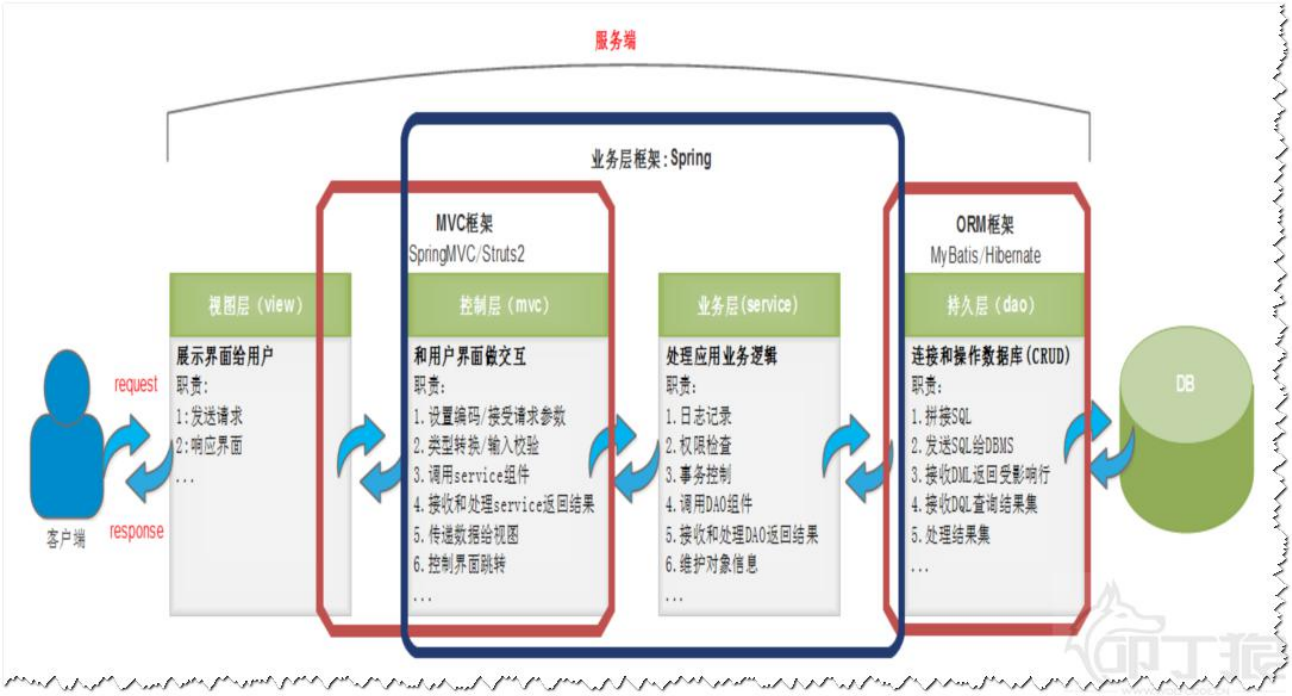
Rod Johnson，Spring Framework 创始人，著名作者。

Rod 在悉尼大学不仅获得了计算机学位，同时还获得了音乐学位。更令人吃惊的是在回到软件开发领域之前，他还获得了音乐学的博士学位。有着相当丰富的 C/C++技术背景的 Rod 早在 1996 年就开始了对 Java 服务器端技术的研究。他是一个在保险、电子商务和金融行业有着丰富经验的技术顾问，同时也是 JSR-154（Servlet2.4）和 JDO2.0 的规范专家、JCP 的积极成员，是 Java development community 中的杰出人物。

Spring 是一个轻量级的 DI/IoC 和 AOP 容器的开源框架，致力于构建致力于构建轻量级的 JavaEE 应用，简化应用开发，本身涵盖了传统应用开发，还拓展到移动端，大数据等领域。

什么是容器（Container）：从程序设计角度看就是装对象的的对象，因为存在放入、拿出等操作，所以容器还要管理对象的生命周期，如 Tomcat 就是 Servlet 和 JSP 的容器。

Java EE 的最佳实践就是在整个应用中按照功能职责不同的纵向划分为三层架构而不同的框架的目的就在于解决不同领域的问题。



Spring 提供了 Java EE 每一层的解决方案，所以我们也说 Spring 是 Java EE 的全栈式（full stack）框架。

1.2.2. 帝国意义-优势

Spring 除了不能帮我们写业务逻辑，其余的几乎什么都能帮助我们简化开发。

①.Spring 能帮我们低侵入/低耦合地根据配置文件创建及组装对象之间的依赖关系。

②.Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制等。

③.Spring 能非常简单的且强大的声明式事务管理。

④.Spring 提供了与第三方数据访问框架（如 Hibernate、JPA）无缝集成，且自己也提供了一套 JDBC 模板来方便数据库访问。

⑤.Spring 提供与第三方 Web（如 Struts1/2、JSF）框架无缝集成，且自己也提供了一套 Spring MVC 框架，来方便 Web 层搭建。

⑥.Spring 能方便的与如 Java Mail、任务调度、缓存框架等技术整合，降低开发难度。

1.2.3. 帝国版图-Spring 家族

Spring 主要产品：

Spring FrameWork:

Spring 帝国之核心，其他 Spring 其他产品都是基于 Spring 框架而来。

Spring Boot:

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

Spring Cloud:

Spring Cloud 微服务框架，为开发者提供了在分布式系统（配置管理，服务发现，熔断，路由，微代理，控

制总线，一次性 token，全居琐，leader 选举，分布式 session，集群状态)中快速构建的工具，使用 Spring Cloud 的开发者可以快速的启动服务或构建应用、同时能够快速和云平台资源进行对接。

Spring Cloud Data Flow:

Spring Cloud Data Flow 简化了专注于数据流处理的应用程序的开发和部署。通过 SpringBoot 启动应用，采用 Spring Cloud Stream、 Spring Cloud Task 完成微服务构建。

Spring Data:

Spring Data 用于简化数据库访问，并支持云服务的开源框架。旨在统一和简化对各类型持久化存储， 而不拘泥是关系型数据库还是 NoSQL 数据存储。

Spring Batch：

专门针对企业级系统中的日常批处理任务的轻量级框架，能够帮助开发者方便地开发出强壮、高效的批处理应用程序。

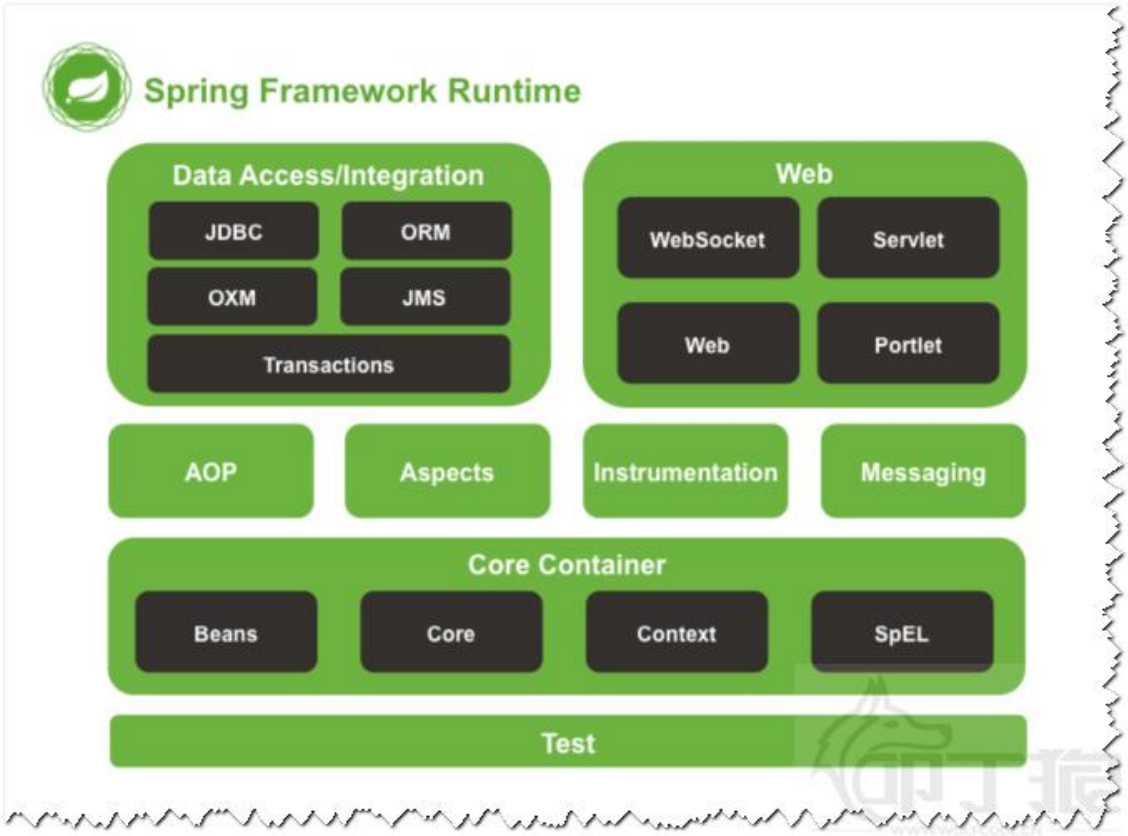
Spring Integration：

Spring Integration 为 Spring 编程模型提供了一个支持企业集成模式（Enterprise Integration Patterns）的扩展，在应用程序中提供轻量级的消息机制，可以通过声明式的适配器与外部系统进行集成。

Spring Security：

Spring Security 早期称为 Acegi ,基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。

1.2.4.帝国核心-Spring Framework



Spring 框架架构:

Core Container(核心容器)包含有 Beans、Core、Context 和 SpEL 模块。

Data Access/Integration 层包含有 JDBC、ORM、OXM、JMS 和 Transactions 模块。

Web 层包含了 Web、Servlet、WebSocket、Porlet 模块。

AOP 模块提供了遵循 AOP 联盟标准的面向切面编程的实现。

Test 模块支持使用 JUnit 和 TestNG 对 Spring 组件进行测试。

Test 模块： Spring 支持 jUnit 和 TestNG 测试框架，而且还额外提供了一些基于 Spring 的测试功能，如在 Web 环境中，模拟 HTTP 请求。

核心容器：包含有 Core、Beans、Context、SpEL 模块。

Core 模块：封装了框架依赖的最底层部分，包括资源访问、类型转换及一些常用工具类。

Beans 模块：提供了框架的基础部分，包括反转控制和依赖注入。其中 Bean Factory 是容器核心，本质是“工厂设计模式”的实现，所有应用中对象间关系由框架管理，这些依赖关系都由 BeanFactory 来维护。

Context 模块：以 Core 和 Beans 为基础，集成 Beans 模块功能并添加资源绑定、数据验证、国际化、Java EE 支持、容器生命周期、事件传播等；核心接口是 ApplicationContext。

EL 模块：提供强大的表达式语言支持，支持访问和修改属性值，方法调用，支持访问及修改数组、容器和

索引器，命名变量，支持算数和逻辑运算，支持从 Spring 容器获取 Bean，它也支持列表投影、选择和一般的列表聚合等。

切面编程模块：包含 AOP、Aspects。

AOP 模块：Spring AOP 模块提供了符合 AOP Alliance 规范的面向方面的编程现，提供比如日志记录、权限控制、性能统计等通用功能和业务逻辑分离的技术，降低业务逻辑和通用功能的耦合。

Aspects 模块：提供了对 AspectJ 的集成，AspectJ 提供了比 Spring AOP 更强大的功能。

Instrumentation 模块：提供了在特定服务器上的类加载器操作。

Messaging 模块：提供消息 API 和协议规范。

数据访问/集成模块：包含 JDBC、ORM、OXM、JMS 和事务管理。

事务模块：该模块用于 Spring 管理事务，支持编程和声明性的事务管理。

JDBC 模块：提供了一个 JBDC 的模板，简化 JDBC 开发。

ORM 模块：提供与流行的“对象-关系”映射框架的无缝集成，包括 Hibernate、JPA、iBatiss 等。而且可以使用 Spring 事务管理，无需额外控制事务。

OXM 模块：提供了一个对 Object/XML 映射实现，将 java 对象映射成 XML 数据，或者将 XML 数据映射成 java 对象，Object/XML 映射实现包括 JAXB、Castor、XMLBeans 和 XStream。

JMS 模块：用于 JMS(Java Messaging Service)，提供一套 “消息生产者、消息消费者”模板用于更加简单的使用 JMS，JMS 用于用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

Web 模块：Web/Remoting 模块包含了 Web、WeSocket、Servlet、Porlet 模块。

Web 模块：提供与 Struts1 和 Struts2 集成技术。

Servlet 模块：提供了一个 Spring MVC Web 框架实现。

WebSocket 模块：提供浏览器与服务端建立全双工的通信方式，解决 http 请求-响应带来过多的资源消耗，同时对特殊场景应用提供了全新的实现方式，比如聊天、股票交易、游戏等对对实时性要求较高的行业领域。

Portlet：提供 Portlet 组件和容器的支持和实现功能，用于构建 Portlet 应用。

1.2.5. 帝国诞生-下载

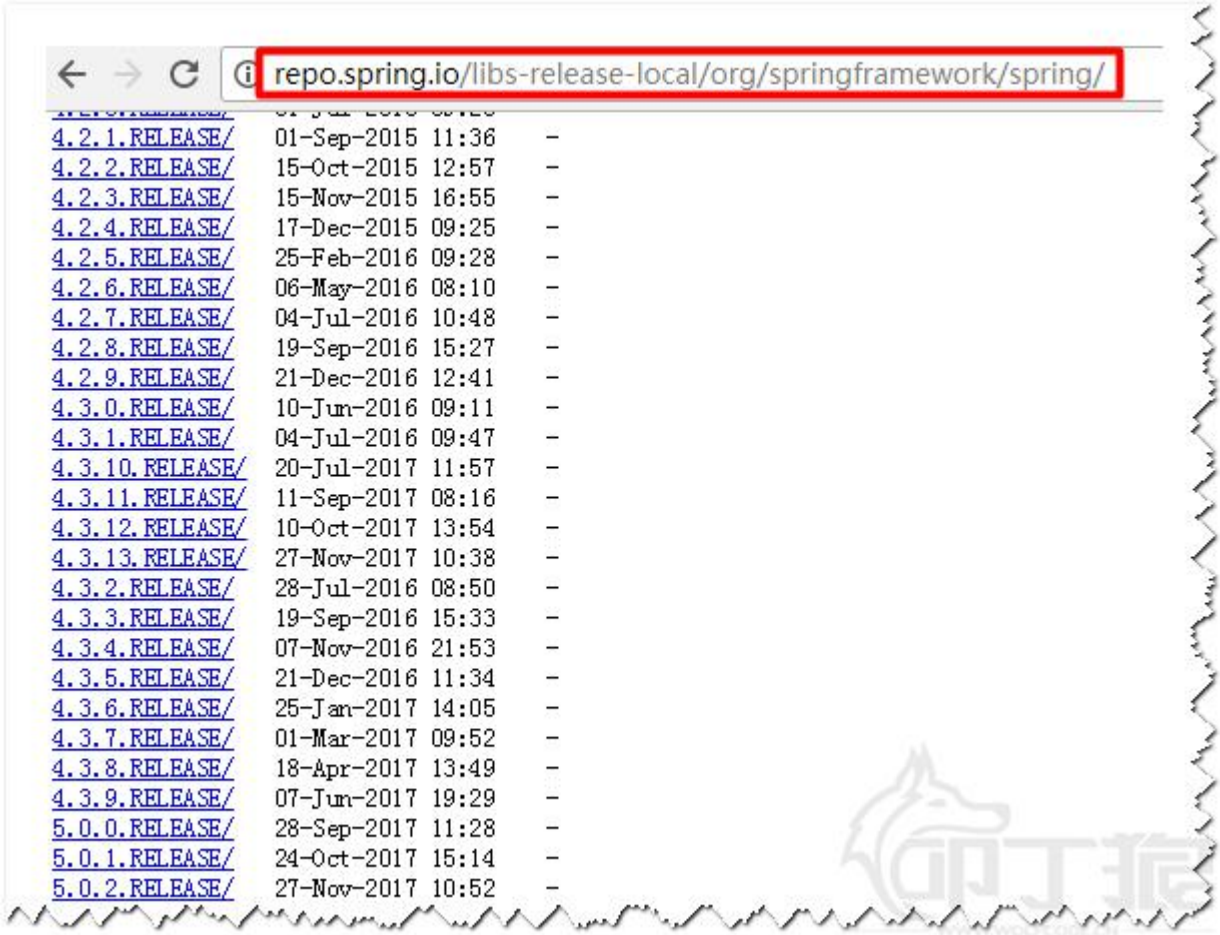
Spring 下载：

一：使用 Maven 或 Gradle 方式根据配置下载依赖。

二：直接访问 URL 下载。

直接下载地址

下载地址：http://repo.spring.io/libs-release-local/org/springframework/spring/



Spring 的新特性：

Spring4.x 新特性：

1，泛型限定式依赖注入

2，核心容器的其他改进

3，Web 开发的增强

4，集成 Bean Validation 1.1(JSR-349)到 SpringMVC

5，Groovy Bean 定义 DSL

6，更好的 Java 泛型操作 API

7，JSR310 日期 API 的支持

8，注解、脚本、任务、MVC 等其他特性改进

Spring FrameWork 5.0 新的功能

JDK 8+和 Java EE7+以上版本

整个框架的代码基于 java8，运行时兼容 JDK9

许多不建议使用的类和方法在代码库中被删除

核心特性

1，JDK8 的增强：

Spring 5.0 框架自带了通用的日志封装

2，核心容器

支持候选组件索引(也可以支持环境变量扫描)

支持@Nullable 注解

函数式风格 GenericApplicationContext/AnnotationConfigApplicationContext

基本支持 bean API 注册

在接口层面使用 CGLIB 动态代理的时候，提供事物，缓存，异步注解检测

3，Spring WebMVC

全部的 Servlet 3.1 签名支持在 Spring-provied Filter 实现

在 Spring MVC Controller 方法里支持 Servlet4.0 PushBuilder 参数

SpringWebFlux

新的 spring-webflux 模块，一个基于 reactive 的 spring-webmvc，完全的异步非阻塞，旨在使用 enent-loop 执行模型和传统的线程池模型。

在 spring-web 包里包含 HttpMessageReade 和 HttpMessageWrite

4，测试方面的改进

支持 JUnit 5

SpringExtension:是 JUnit 多个可拓展 API 的一个实现，提供了对现存 Spring TestContext Framework 的支持，使用@ExtendWith(SpringExtension.class)注解引用。

@SpringJunitConfig:一个复合注解

@ExtendWith(SpringExtension.class) 来源于 Junit Jupit

@ContextConfiguration 来源于 Spring TestContext 框架

@DisabledIf 如果提供的该属性值为 true 的表达或占位符，信号：注解的测试类或测试方法被禁用

在 Spring TestContext 框架中支持并行测试

1.2.6. 帝国之器-STS

工欲善其事必先利其器，Spring 开发利器除了 IDEA，STS（Spring Tool Suite）绝对够霸道。

下载地址：https://spring.io/tools/sts

he Spring Tool Suite is an Eclipse-based development environment that is customized for developing Spring applications. It provides a ready-to-use environment to implement, debug, run, and deploy your Spring applications, including integrations for Pivotal tc Server, Pivotal Cloud Foundry, Git, Maven, AspectJ, and comes on top of the latest Eclipse releases.

1.3. 帝国第一战

1.3.1. IoC 和 DI 思想

IoC：Inversion of Control（控制反转）:读作“反转控制”，更好理解，不是什么新技术，而是一种设计思想，好比于 MVC。

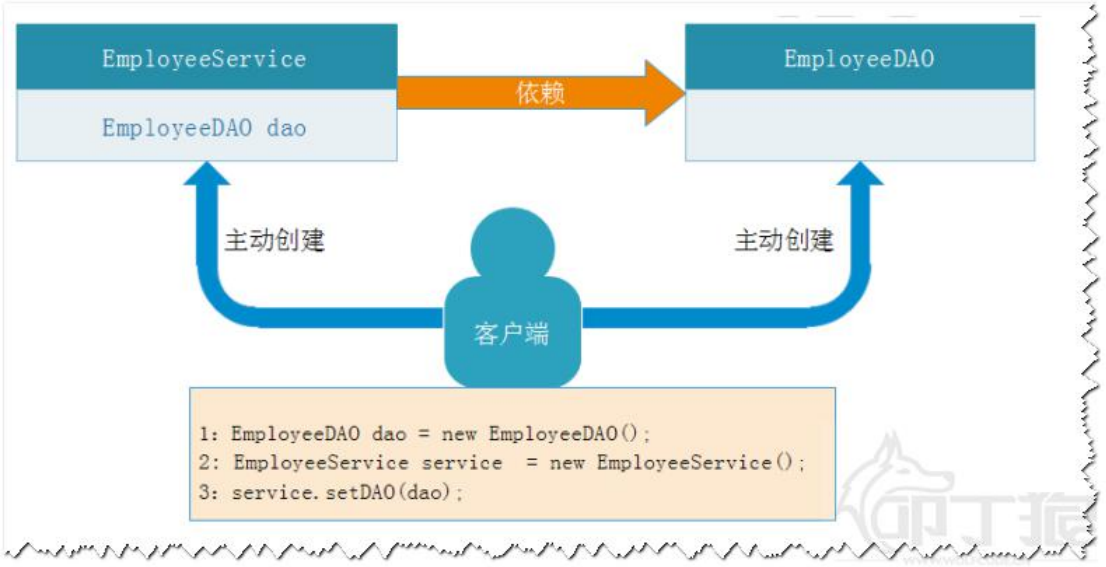
其本意是是将原本在程序中手动创建对象的控制权，交由 Spring 框架来管理。

为了弄明白“反控”的概念，我们先假设存在“正控”，即没有 IoC 之前，我们是怎么操作的。

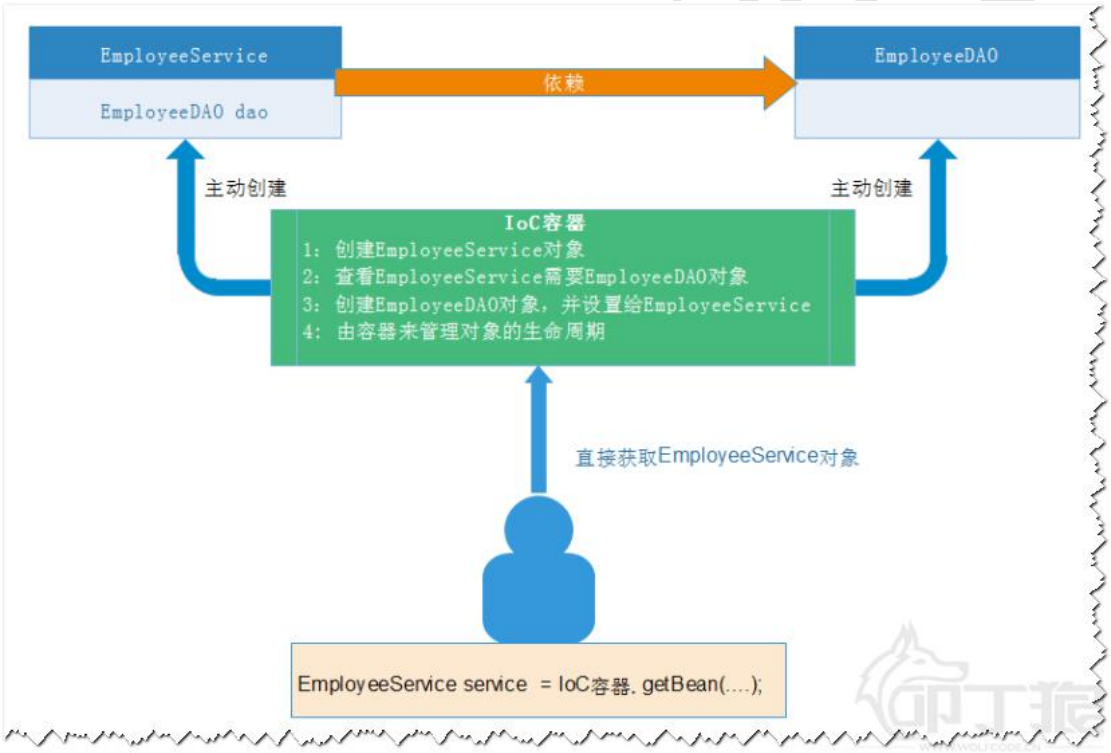
需求：

创建一个 EmployeeDAO 对象和 EmployeeService 对象，并把 EmployeeDAO 对象设置给 EmployeeService 对象。

正控：若调用者需要使用某个对象，其自身就得负责该对象及该对象所依赖对象的创建和组装。



反控：调用者只管负责从 Spring 容器中获取需要使用的对象，不关心对象的创建过程，也不关心该对象依赖对象的创建以及依赖关系的组装，也就是把创建对象的控制权反转给了 Spring 框架。

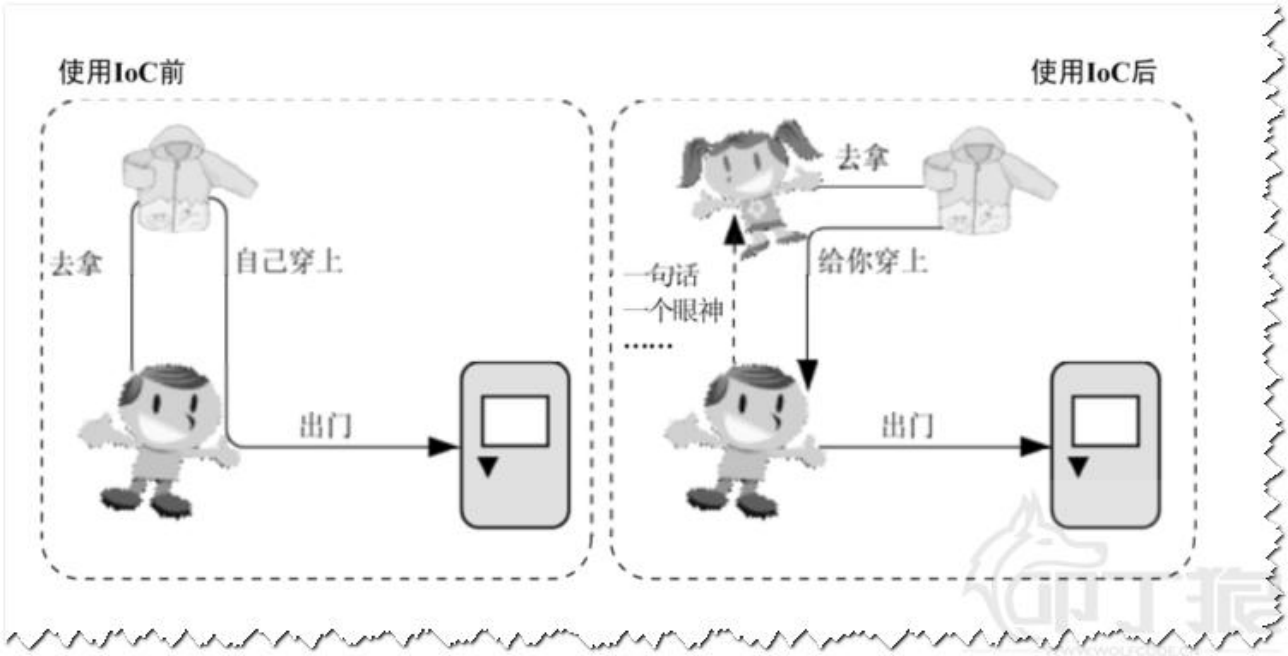


这里完美地体现了好莱坞法则（Don’ t call me ,I’ ll call you ）

DI：Dependency Injection（依赖注入）从字面上分析：IoC：指将对象的创建权，反转给了 Spring 容器；DI ：指 Spring 创建对象的过程中，将对象依赖属性（常量，对象，集合）通过配置设值给该对象。

IoC 从字面意思上很难体现出谁来维护对象之间的关系，Martin Fowler 提出一个新的概念——DI，更明确描述了“被注入对象（Service 对象）依赖 IoC 容器来配置依赖对象（DAO 对象）”。

正控和反控对比：



上述图片来源于互联网

1.3.2.第一滴血

需求：使用 Spring 先做一个 IoC 案例，再在其基础之上增加 DI 操作。

依赖的 jar:

spring-beans-版本.RELEASE.jar

spring-core-版本.RELEASE.jar

报错再添加：

com.springsource.org.apache.commons.logging-1.版本.jar

代码：

```
public class HelloWorld {
  private String name;
  private int age;

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public void sayHello() {
```

```
        System.out.println(name + "你好,年龄" + age);
      }
    }
```

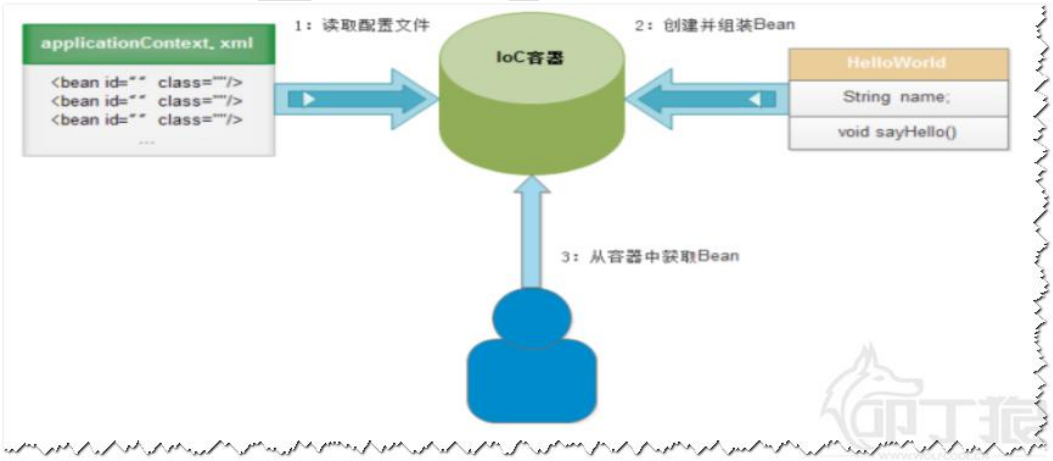
XML 配置：

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

      <bean id="helloWorld" class="cn.wolfcode.day1._02_hello.HelloWorld">
        <property name="name" value="will" />  对应 HelloWorld 中的 setName 方法
        <property name="age" value="17" />     对应 HelloWorld 中的 setAge 方法
      </bean>
```

测试类：

```
    HelloWorld world = null;
    //加载 Spring 配置文件 applicationContext.xml
    Resource resource = new ClassPathResource("applicationContext.xml");
    //创建 Spring 容器对象
    BeanFactory factory = new XmlBeanFactory(resource);
    //从 Spring 容器中获取制定名为 helloWorld 的 bean
    world = (HelloWorld) factory.getBean("helloWorld");
    world.sayHello();
```

例子示意图：



1.3.3.第一滴血背后

什么是 BeanFactory：

BeanFactory 是 Spring 最古老的接口，表示 Spring IoC 容器——生产 bean 对象的工厂，负责配置，创

建和管理 bean。

被 Spring IoC 容器管理的对象称之为 bean。

Spring IoC 容器如何知道哪些是它管理的对象:

此时需要配置文件，Spring IoC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。

元数据的配置有三种方式（后讲）：

- XML-based configuration
- Annotation-based configuration
- Java-based configuration

Spring IoC 管理 bean 的原理：

1、通过 Resource 对象加载配置文件

2、解析配置文件，得到指定名称的 bean

3、解析 bean 元素，id 作为 bean 的名字，class 用于反射得到 bean 的实例：

注意：此时，bean 类必须存在一个无参数构造器(和访问权限无关)；

4、调用 getBean 方法的时候，从容器中返回对象实例；

结论：就是把代码从 JAVA 文件中转移到了 XML 中。

```
HelloWorld wolrd = null;
String className="cn.wolfcode.day1._02_hello.HelloWorld";
//-----------------------------------------------------
Class clz = Class.forName(className);
Constructor con = clz.getDeclaredConstructor();
con.setAccessible(true);
wolrd = (HelloWorld)con.newInstance();
BeanInfo beanInfo = Introspector.getBeanInfo(wolrd.getClass());
PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();
  for (PropertyDescriptor pd : pds) {
    String propertyName = pd.getName();
    Method setterMethod =pd.getWriteMethod();
    if("name".equals(propertyName)){
       setterMethod.invoke(wolrd, "Will");
    }else if("age".equals(propertyName)) {
       setterMethod.invoke(wolrd, 17);
    }
  }
//-----------------------------------------------------
wolrd.sayHello();
```

1.3.4.getBean 方法签名

1,按照 bean 的名字拿 bean,按照名字拿 bean 不太安全

```
world = (HelloWorld) factory.getBean("helloWorld");
```

2,按照类型拿 bean,要求在 Spring 中只配置一个这种类型的实例

```
world = factory.getBean(HelloWorld.class);
```

3,按照名字和类型:(推荐)

```
world = factory.getBean("helloWorld",HelloWorld.class);
```

常见错误：

错误 1：NoSuchBeanDefinitionException: No bean named 'helloWorld2' available

按照 bean 名称去获取 bean 时，不存在名称为 helloWorld2 的 bean.

错误 2 ：BeanDefinitionParsingException: Configuration problem: Bean name 'helloWorld' is already used in this <beans> element Offending resource: class path resource [applicationContext.xml]

在 applicationContext.xml 文件中，多个 bean 元素的名称是 helloWorld.

错误 3：NoUniqueBeanDefinitionException: No qualifying bean of type 'cn.wolfcode.day1._02_hello.HelloWorld' available: expected single matching bean but found 2: helloWorld,helloWorld2

按照 HelloWorld'类型去获取 bean 时，期望找到该类型唯一的一个 bean，可是此时找到了两个。

1.3.5.Eclipse 提示 XML 语法

XML 需要导入 schema 约束，约束指向网络路径：

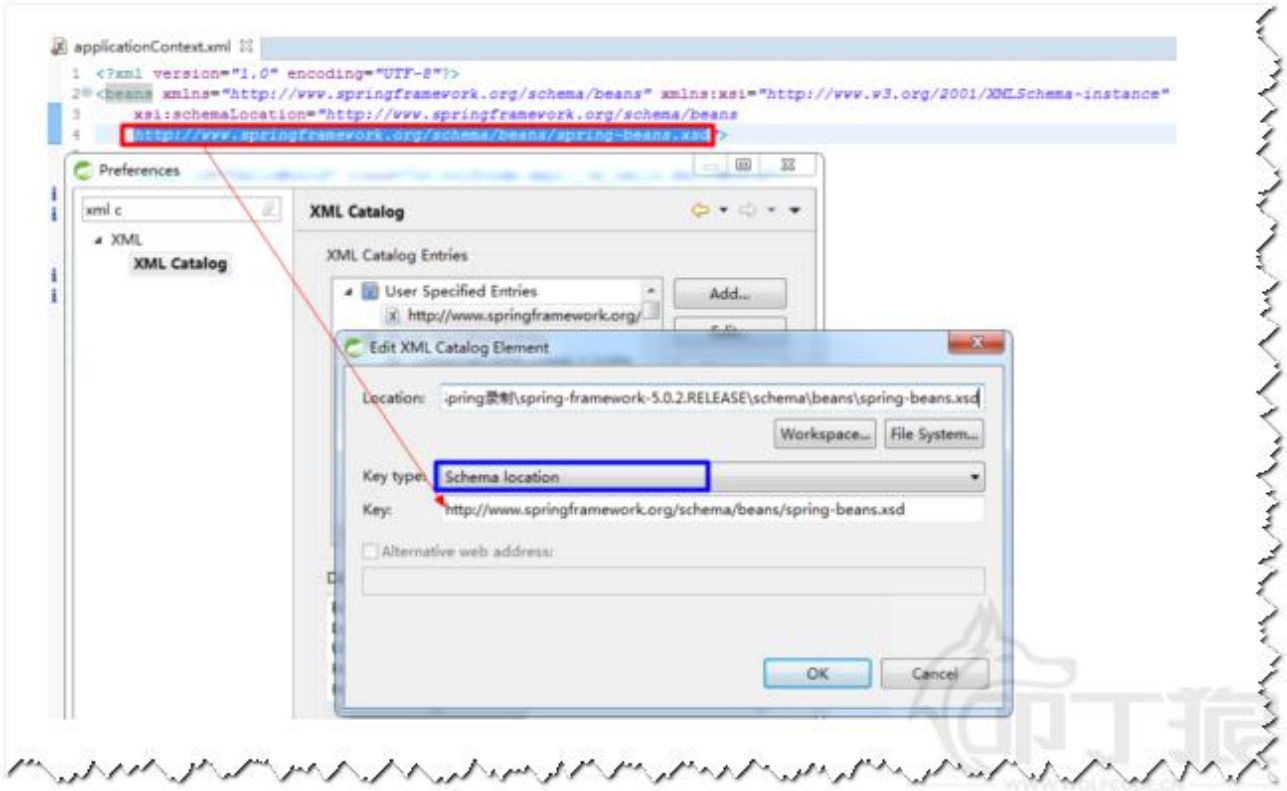
方式一、联网后，自动缓存路径文件到本地，提供提示功能；

方式二、无法联网 ，需要配置 xsd schema 文件位置（操作如下图）：

①到解压 spring/schemas/beans/spring-beans.xsd 。

②选择 schema location 方式 。

③复制网络路径 http://www.springframework.org/schema/beans/spring-beans.xsd。



1.3.6.Spring 基本配置

在 bean 元素中 id 和 name 属性:

在 Spring 配置中，id 和 name 属性都可以定义 bean 元素的名称，不同的是：lid 属性，遵守 XML 语法 ID 约束。

必须以字母开始，可以使用字母、数字、连字符、下划线、句话、冒号，不能以“/”开头。lname 属性，就可以使用很多特殊字符，比如在 Spring 和 Struts1，就得使用 name 属性来的定义 bean 的名称。

```
<bean name="/login" class="cn.woldcode.crm.web.action.LoginAction" />
```

注意：从 Spring3.1 开始，id 属性不再是 ID 类型了，而是 String 类型，也就是说 id 属性也可以使用“/”开头了，而 bean 元素的 id 的唯一性由容器负责检查。

当然也可以使用 name 属性为 bean 元素起多个别名，多个别名之间使用逗号或空格隔开，在代码中依然使用 BeanFactory 对象.getBean(...)方法获取。

```
<bean name="hello,hi" class="cn.woldcode.crm.day1._02_hello.HelloWorld"/>
或则<bean name="hello hi" class="com._520it.day1._01_hello.HelloWorld"/>
```

建议：bean 起名尽量规范，不要搞一些非主流的名字，尽量使用 id。

<import resource=""/>元素：

在开发中，随着应用规模的增加，系统中<bean>元素配置的数量也会大量增加，导致 applicationContext.xml 配置文件变得非常臃肿。

为提高其可读性，我们可以将一个 applicationContext.xml 文件分解成多个配置文件，然后在

applicationContext.xml 文件中包含其他配置文件即可。

语法如下：

```
<import resource="classpath:cn/wolfcode/day1/_02_hello/hello.xml"/>
```

使用 import 元素注意：

1、默认情况下，从 classpath 的跟路径寻找。

2、可以使用前缀来定位文件的基础位置：

①：[classpath:]：后面的文件从 classpath 路径开始找(推荐)；

②：[file:]：后面的文件使用文件系统的路径开始找；

注意：只有当框架中实现了 Resource 接口才能够识别上述的前缀标识符。

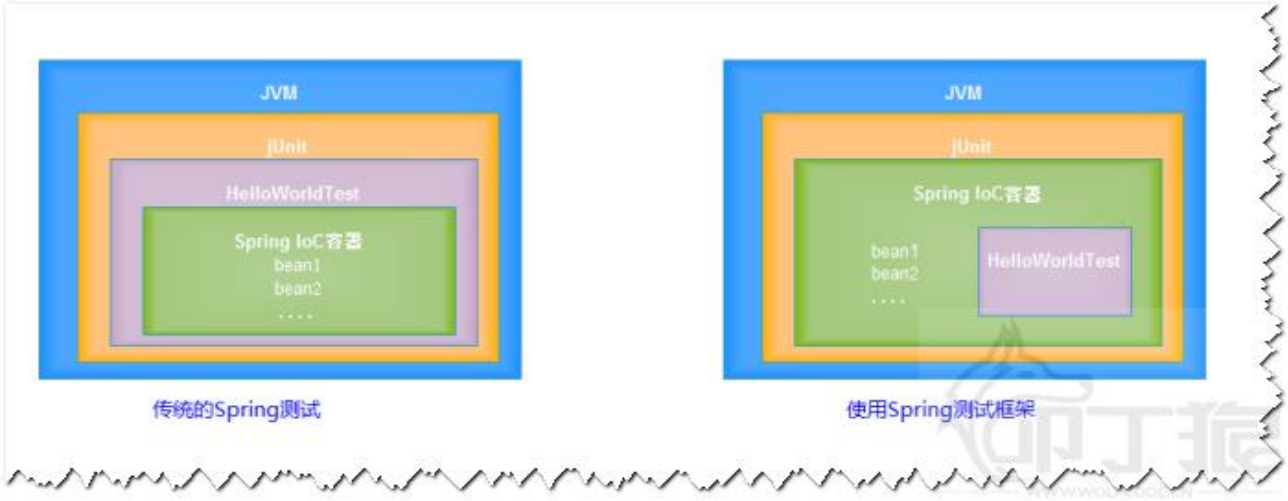
1.3.7.Spring 测试框架

传统测试存在的问题：

1，每个测试都要重新启动 Spring，启动容器的开销大，测试效率低下。

2，不应该是测试代码管理 Spring 容器，应该是 Spring 容器在管理测试代码。

3，不能正常的关闭 Spring 容器，Spring 容器生命周期非正常退出。



如何使用 Spring 的测试框架：

依赖的 jar:

```
spring-test-版本.RELEASE.jar
spring-context-版本.RELEASE.jar
spring-aop-版本.RELEASE.jar
spring-expression-版本.RELEASE.jar
```

基于 jUnit4 的测试：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:springTest.xml")
public class SpringTestTest {

  @Autowired
  private HelloWorld world;

  @Test
  public void test1() throws Exception {
    world.sayHello();
  }
}
```

若把@ContextConfiguration("classpath:springTest.xml") 写成@ContextConfiguration

默认去找的当前测试类名-context.xml 配置文件,如:SpringTestTest-context.xml

@Autowired 注解 先记住表示从 Spring IoC 容器中根据类型找到对应的 bean 并自动注入到某个字段上。

基于 jUnit5 的测试：

```
@SpringJUnitConfig
public class SpringTestTest {

  @Autowired
  private HelloWorld world;

  @Test
  void test1() throws Exception {
    world.sayHello();
  }
}
```

jUnit 版本问题：

从 Spring4.x 需要依赖的单元测试得是最新的 junit4.12，如果 Eclipse 自带的 jUnit 是 4.8 版本，则不支持，测试手动加入 jUnit 的库。

junit-4.12.jar

hamcrest-core-1.3.jar

2. 第二章 Spring 帝国之剑-IoC

2.1. IoC 核心（基于 XML）

2.1.1.IoC 容器

Spring IoC 容器（Container）：

BeanFactory：Spring 最底层的接口，只提供了的 IoC 功能，负责创建、组装、管理 bean，在应用中，一般不使用 BeanFactory，而推荐使用 ApplicationContext（应用上下文）。

ApplicationContext 接口继承了 BeanFactory，除此之外还提供 AOP 集成、国际化处理、事件传播、统一资源价值等功能。

```
public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory,
                HierarchicalBeanFactory,MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
}
```

bean 的创建时机（此时不使用 Spring Test）：

1、BeanFactory 需要等到获取某一个 bean 的时候才会创建该 bean--延迟初始化。

2、ApplicationContext 在启动 Spring 容器的时候就会创建所有的 bean（Web 应用建议）。

Beanactory：

```
    @Test
    void testBeanFactory() throws Exception {
        Resource resource =
    new ClassPathResource("cn/wolfcode/day1/_04_container/ContainerTest-context.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        System.out.println("--------before --------");
        factory.getBean("someBean", SomeBean.class);
        System.out.println("--------after --------");
      }
```

输出结果

```
输出结果：
--------before --------
创建 SomeBean 对象
```

ApplicationContext：

```
@Test
 void    testBeanApplicationContext() throws Exception {
```

```
    ApplicationContext ctx = new ClassPathXmlApplicationContext("cn/wolfcode/day1/_04_container/ContainerTest-context.xml");
    System.out.println("--------before --------");
    ctx.getBean("someBean", SomeBean.class);
    System.out.println("--------after --------");
  }
```

输出结果：

```
创建 SomeBean 对象
--------before --------
--------after --------
```

延迟初始化配置（了解）：

针对于当前 xml 中所有的 bean。

```
<beans default-lazy-init="default | false | true">
```

针对于指定的 bean:

```
<bean lazy-init="default | false | true">
```

再讲使用 Spring Test 注入 BeanFactory、ApplicationContext 对象

```
@SpringJUnitConfig
public class ContainerTest {
  @Autowired
  private BeanFactory factory;
  @Autowired
  private ApplicationContext ctx;
}
```

2.1.2. Bean 实例化方式

①.构造器实例化（无参数构造器），最标准，使用最多。

②.静态工厂方法实例化：解决系统遗留问题。

③.实例工厂方法实例化：解决系统遗留问题。

④.实现 FactoryBean 接口实例化：实例工厂变种。

如集成 MyBatis 框架使用：org.mybatis.spring.SqlSessionFactoryBean

1. 构造器实例化（无参数构造器），最标准，使用最多。

```
public class SomeBean1 {
  public SomeBean1() {
    System.out.println("SomeBean1 构造器");
  }
}
```

```
<bean id="someBean1" class="cn.wolfcode.day1._05_create_bean._1constructor.SomeBean1"/>
```

2. 静态工厂方法实例化：解决系统遗留问题。

```
public class SomeBean2 {
}

public class SomeBean2Factory {
  public static SomeBean2 createInstance() {
    //TODO
    return new SomeBean2();
  }
}

<bean id="someBean2"
     class="cn.wolfcode.day1._05_create_bean._2staticfactory.SomeBean2Factory"
     factory-method="createInstance"/>
```

3. 实例工厂方法实例化：解决系统遗留问题。

```
public class SomeBean3 {
}

public class SomeBean3Factory {

  public SomeBean3 createInstance() {
    //TODO
    return new SomeBean3();
  }
}

<bean id="factory"
     class="cn.wolfcode.day1._05_create_bean._3instancefactory.SomeBean3Factory"/>

<bean id="someBean3" factory-bean="factory" factory-method="createInstance"/>
```

4. 实现 FactoryBean 接口实例化：实例工厂变种。

```
public class SomeBean4 {
}

public class SomeBean4FactoryBean implements FactoryBean<SomeBean4>{
  public SomeBean4 getObject() throws Exception {
    //TODO
    return new SomeBean4();
  }

  public Class<?> getObjectType() {
```

```
        return SomeBean4.class;
      }
    }


    <bean id="someBean4"
         class="cn.wolfcode.day1._05_create_bean._4factorybean.SomeBean4FactoryBean"/>
```

演示给 SomeBean4FactoryBean 注入属性操作。

2.1.3. Bean 作用域

在 Spring 容器中是指其创建的 Bean 对象相对于其他 Bean 对象的请求可见范围，定于语法格式：

```
    <bean id="" class="" scope="作用域"/>
```

singleton: 单例 ，在 Spring IoC 容器中仅存在一个 Bean 实例 （默认的 scope ）。

prototype: 多例 ，每次从容器中调用 Bean 时，都返回一个新的实例，即每次调用 getBean()时 ，相当于执行 new XxxBean()，不会在容器启动时创建对象。

request: 用于 web 开发，将 Bean 放入 request 范围 ，request.setAttribute("xxx") ， 在同一个 request 获得同一个 Bean。

session: 用于 web 开发，将 Bean 放入 Session 范围，在同一个 Session 获得同一个 Bean 。

globalSession: 一般用于 Porlet 应用环境 ，分布式系统存在全局 session 概念（单点登录），如果不是 porlet 环境，globalSession 等同于 Session 。

application ：Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext。

websocket ： Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext。

在开发中主要使用 scope="singleton"、 scope="prototype" 。

总结 :对于 Struts1 只的 Action 使用 request ,Struts2 中的 Action 使用 prototype 类型 ,其他的如 DAO、Service、Controller 使用 singleton。

2.1.4. Bean 初始化和销毁

比如 DataSource,SqlSessionFactory 最终都需要关闭资源:在 Bean 销毁之前,都要调用 close 方法 ,Spring IoC 容器可以帮我们管理对象的创建，如何帮我们管理对象在创建之后的初始化操作和回收之前的扫尾操

作，语法如下：

```
<bean id="someBean" class="....."
        init-method="该类中初始化方法名"
        destroy-method="该类中销毁方法名">
</bean>
```

init-method：bean 生命周期初始化方法,对象创建后就进行调用

destroy-method:容器被正常销毁的时候，如果 bean 被容器管理，会调用该方法。

代码：

```
public class SomeBean {
  public SomeBean() {
    System.out.println("执行构造器");
  }

  public void open() {
    System.out.println("打开资源");
  }

  public void close() {
    System.out.println("释放资源");
  }

  public void doWork() {
    System.out.println("执行工作");
  }
}
```

XML 配置：

```
<bean id="someBean" class="cn.wolfcode.day1._06_lifecycle.SomeBean"
    init-method="open"
    destroy-method="close" />
```

测试类：

```
@Test
  void test() throws Exception {
    ApplicationContext ctx =
         new ClassPathXmlApplicationContext
                  ("cn/wolfcode/day1/_06_lifecycle/LifeCycleTest-context.xml");
    SomeBean bean = ctx.getBean("someBean", SomeBean.class);
    bean.doWork();
  }
```

此时因为没有正常关闭 Spring 容器，所以不会执行释放资源方法。

演示 Lombok 的@Cleanup 注解

```
最好的方式:把 Spring 线程作为 JVM 的子线程: ctx.registerShutdownHook();
```

分析原理:

如果 bean 的 scope="prototype",那么容器只负责创建和初始化，它并不会被 spring 容器管理。交给用户自己调用.

总结:bean 的生命周期过程

2.1.5. Bean 实例化过程

2.2. DI 核心（基于 XML）

什么是注入操作？

怎么注入

setter 方法注入

构造器注入

注入值类型

1) 常量值（简单类型）：value 元素
2) 对象 ：ref 元素
3) 集合 ：对应集合类型元素

2.2.1. XML 自动装配（不推荐）

设置：<bean />元素的：autowire 属性

```
<bean id="somebean" class="SomeBean 全限定名" autowire="byType"/>
```

autowire 属性:让 Spring 按照一定的规则方式自己去找合适的对象,并完成 DI 操作。

```
- default:不要自动注入，缺省 default 表示 no
- no:不要自动注入
- byName:按照名字注入(按照属性的名字在 spring 中找 bean) factory.getBean(String beanName)
- byType:按照依赖对象的类型注入(factory.getBean(Class requiredType))
- constructor:按照对象的构造器上面的参数类型注入
```

注意：

1,如果按照 byName 自动注入,要求所有的属性名字和 id 的名字必须保证一种规范的命名方式;

2,如果按照 byType 注入,如果 spring 中同一个类型有多个实例-->报 bean 不是唯一类型错误;

该方式不推荐，了解即可。

```
public class OtherBean {
}

public class SomeBean {
  private OtherBean other;

  public void setOther(OtherBean other) {
    this.other = other;
  }

  public String toString() {
    return "SomeBean [other=" + other + "]";
  }
}

<bean id="other" class="cn.wolfcode.day1._07_di._01xml_auto.OtherBean" />
<bean id="someBean"
      class="cn.wolfcode.day1._07_di._01xml_auto.SomeBean" autowire="byType"/>
```

2.2.2.Setter 方法注入

其实就是通过对象的 setter 方法来完成对象的设置操作：

1，使用 bean 元素的<property>子元素设置；

1，常量类型，直接使用 value 赋值；

2，引用类型，使用 ref 赋值；

3，集合类型，直接使用对应的集合类型元素即可。

2，Spring 通过属性的 setter 方法注入值；

3，在配置文件中配置的值都是 String，Spring 可以自动的完成类型的转换

4，属性的设置值是在 init 方法执行之前完成的

5，改进 Spring 的测试，直接在测试类里面注入需要测试的对象

2.2.2.1. 注入常量

注入常量值，也称之为注入简单类型，语法：

```
<property name="对象属性名称">
      <value>需要注入的值</value>
</property>
```

简写：

```
    <property name="对象属性名称" value="需要注入的值">
```

Java 代码：

```
    public class Employee {
      private String name;
      private Integer age;
      private BigDecimal salary;
      private URL url;

      public void setName(String name) {
        this.name = name;
      }
      public void setAge(Integer age) {
        this.age = age;
      }
      public void setSalary(BigDecimal salary) {
        this.salary = salary;
      }
      public void setUrl(URL url) {
        this.url = url;
      }
      public String toString() {
        return "Employee [name=" + name + ", age=" + age + ", salary=" + salary + ", url=" +
url + "]";
      }
    }
```

XML 配置：

```
    <bean id="employee" class="cn.wolfcode.day1._07_di._2setter.Employee">
       <property name="name"  value="will"/>
       <property name="age" value="17"/>
       <property name="salary" value="800"/>
       <property name="url" value="http://www.wolfcode.cn"/>
      </bean>
```

2.2.2.2. 注入对象

注入对象，就是把一个对象，通过 setter 方法设置给另一个对象，语法：

```
    <property name="对象属性名称">
          <ref bean="被注入对象的 bean 的 id" />
     </property>
```

简写：

```
    <property name="dao" ref="employeeDAO" />
```

2.2.2.3. 注入集合

注入集合，就是把一个集合类型的数据，通过 setter 方法设置给一个对象：

Java 代码：

```
public class CollectionBean {
  private Set<String> set;
  private List<String> list;
  private String[] array;
  private Map<String, String> map;
  private Properties prop;

  public void setSet(Set<String> set) {
    this.set = set;
  }

  public void setList(List<String> list) {
    this.list = list;
  }

  public void setArray(String[] array) {
    this.array = array;
  }

  public void setMap(Map<String, String> map) {
    this.map = map;
  }

  public void setProp(Properties prop) {
    this.prop = prop;
  }

  public String toString() {
    return "CollectionBean [set=" + set + ", list=" + list + ", array=" + Arrays.toString(array)
+ ", map=" + map + ", prop=" + prop + "]";
  }
}
```

XML 配置：

```
<bean id="collectionBean" class="cn.wolfcode.day1._07_di._2setter.CollectionBean">
   <property name="set">
     <set>
       <value>set1</value>
       <value>set2</value>
```

```
        </set>
      </property>
      <property name="list">
        <list>
          <value>list1</value>
          <value>list2</value>
        </list>
      </property>
      <property name="array">
        <list>
          <value>array1</value>
          <value>array2</value>
        </list>
      </property>
      <property name="map">
        <map>
          <entry key="key1" value="value1" />
          <entry key="key2" value="value2" />
        </map>
      </property>
      <property name="prop">
        <props>
          <prop key="pKey1">pVal1</prop>
          <prop key="pKey2">pVal2</prop>
        </props>
      </property>
  </bean>
```

对于 Properties 类型的简单配置（常用）：

```
  <property name="prop">
        <value>
          pKey1=pValue1
          pKey2=pValue2
        </value>
  </property>
```

2.2.3.构造器注入

其实就是通过构造器，在完成创建对象的时候，同时初始化对象设置数据的操作：

setter 方式注入使用：<property />元素

构造器方式注入使用：<constructor-arg />元素

1，注入常量使用 value,注入对象使用 ref，注入集合使用对应的集合类型元素即可。

2，默认情况下 constructor-arg 的顺序就是构造器参数的顺序（不建议）。

3，调整构造器中设置参数顺序：

1.index:构造器中的参数位置，从 0 开始。（不建议）

2.type:构造器中的参数的类型。（不建议）

3.name:构造器中按照构造器的参数名字设置值 。（建议）

Java 代码：

```
public class SomeBean {
  private String name;
  private int age;
  private OtherBean other;
  private Properties prop;

  public SomeBean(String name, int age, OtherBean other, Properties prop) {
    this.name = name;
    this.age = age;
    this.other = other;
    this.prop = prop;
  }

  public String toString() {
    return "SomeBean [name=" + name + ", age=" + age + ", other=" + other + ", prop=" + prop
+ "]";
  }
}
```

XML 配置：

```
<bean id="otherBean" class="cn.wolfcode.day1._07_di._3constructor.OtherBean"/>

<bean id="someBean" class="cn.wolfcode.day1._07_di._3constructor.SomeBean">
   <constructor-arg name="name" value="will"/>
   <constructor-arg name="age" value="17" />
   <constructor-arg name="other" ref="otherBean"/>
   <constructor-arg name="prop">
     <value>
       pKey1=pValue1
       pKey2=pValue2
     </value>
   </constructor-arg>
```

```
    </bean>
```

演示注入空值（<null/>）和内部 bean。

2.2.4.方法注入

其实就是通过配置方式覆盖或拦截指定的方法，通常通过代理模式实现。

Spring 提供两种方法注入：查找方法注入和方法替换注入。

因为 Spring 是通过 CGLIB 动态代理方式实现方法注入，也就是通过动态修改类的字节码来实现的，本质就是生成需方法注入的类的子类方式实现。

方法定义格式：必须保证被子类覆盖

1：查找方法的类和被重载的方法必须为非 final

2：访问级别必须是 public 或 protected

3：可以是抽象方法，必须有返回值，必须是无参数方法

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

一、查找方法注入：又称为 lookup 方法注入，用于注入方法返回结果，也就是说能通过配置方式替换方法返回结果。

使用<lookup-method name="方法名" bean="bean 名字"/>配置；其中 name 属性指定方法名，bean 属性指定方法需返回的 Bean。

二、替换方法注入：也叫“MethodReplacer”注入，和查找注入方法不一样的是，他主要用来替换方法体，通过实现 MethodReplacer 接口。

2.2.5.SpEL 注入

Spring 表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于 Struts2x 中使用的 OGNL 表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与 Spring 功能完美整合，如能用来配置 Bean 定义。

表达式语言给静态 Java 语言增加了动态功能。

SpEL 是单独模块，只依赖于 core 模块，不依赖于其他模块，可以单独使用。

表达式语言一般是用最简单的形式完成最主要的工作，减少我们的工作量。

SpEL 支持如下表达式：

一、基本表达式：字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及 Elivis 表达式、正则表达式、括号优先级表达式；

二、类相关表达式：类类型表达式、类实例化、instanceof 表达式、变量定义及引用、赋值表达式、自定

义函数、对象属性存取及安全导航表达式、对象方法调用、Bean 引用；

三、集合相关表达式：内联 List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、其他表达式：模板表达式。

注：SpEL 表达式中的关键字是不区分大小写的。

2.2.6. Bean 元素继承

多个 bean 元素共同配置的抽取，实则是 bean 配置的拷贝，和 Java 的继承不同。

Java 的继承： 把多个类共同的代码抽取到父类中。

bean 元素的继承： 把多个 bean 元素共同的属性配置抽取到另一个公用的 bean 元素中。

原来的配置：

```
    <bean id="someBean1" class="cn.wolfcode.day2._01_bean_tag_inheritance.SomeBean1">
       <property name="name" value="will" />
       <property name="age" value="17" />
    </bean>

    <bean id="someBean2" class="cn.wolfcode.day2._01_bean_tag_inheritance.SomeBean2">
       <property name="name" value="lucy" />
       <property name="age" value="18" />
       <property name="color" value="red" />
    </bean>
```

存在继承后：

```
    <bean id="base" abstract="true">
       <property name="name" value="default" />
       <property name="age" value="100" />
    </bean>

    <bean    id="someBean1"    class="cn.wolfcode.day2._01_bean_tag_inheritance.SomeBean1"
parent="base"/>

    <bean    id="someBean2"    class="cn.wolfcode.day2._01_bean_tag_inheritance.SomeBean2"
parent="base">
       <property name="age" value="18"/>
       <property name="color" value="red" />
    </bean>
```

2.2.7.注册案例

2.3. 属性占位符

2.3.1.配置连接池

拷贝 MySQL 和 druid 库

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"
destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/springdemo"/>
        <property name="username" value="root"/>
        <property name="password" value="admin"/>
        <property name="initialSize" value="5"/>
    </bean>
```

2.3.2.Property Place Holder

db.properties 文件

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql:///springdemo
jdbc.username=root
jdbc.password=admin
jdbc.initialSize=5
```

在 applicationContext.xml 文件中增加 context 命名空间：

1：手动配置

2：使用 STS 工具

```
    <context:property-placeholder location="classpath:db.properties"/>

    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"
destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
        <property name="initialSize" value="${jdbc.initialSize}"/>
    </bean>
```

演示 system-properties-mode 属性问题

2.4. 注解配置 IoC 和 DI

通过 XML 完成 Spring 的配置确实很清晰，而使用注解配置现在在企业中越来越受青睐。

2.4.1. DI 注解

2.4.1.1. Autowired 注解和 Qualifier 注解

Spring 规范提供

1，可以让 Spring 自动的把属性需要的对象找出来,并注入到对象

2，可以贴在字段或者 setter 方法上面

3，可以同时注入多个对象

```
    @Autowired
    public void setter(OtherBean otherBean, OtherBean other2) {
    }
```

4，可以注入一些 Spring 内置的重要对象,比如 BeanFactory,ApplicationContext，ServletContext 等;

5，默认情况下 Autowired 注解必须要能找到对应的对象,否则报错。

通过 required=false 来避免这个问题：@Autowired(required=false)

6，注解解析器，Spring3.0 之前,需要手动配置 Autowired 注解的解析程序，，在 Web 开发中必须配置。

```
<context:annotation-config />
```

7，Autowired 注解寻找 bean 的方式：

1)，首先按照依赖对象的类型找，如果找到，就是用 setter 方法或者字段直接注入；

2)，如果在 Spring 上下文中找到多个匹配的类型，再按照名字去找，如果没有匹配报错；

3)，可以通过使用@Qualifier("other")标签来规定依赖对象按照 bean 的 id 和类型的组合方式去找；

Java 代码：

```
public class SomeBean {
  @Autowired
  @Qualifier("otherBean1")
  private OtherBean other1;
  @Autowired
  @Qualifier("otherBean2")
  private OtherBean other2;
```

```
  public String toString() {
    return "SomeBean [other1=" + other1 + ", other2=" + other2 + "]";
  }
}
```

XML 配置：

```
<context:annotation-config/>

<bean id="otherBean1" class="cn.wolfcode.day2._02_di_anno.OtherBean"/>
<bean id="otherBean2" class="cn.wolfcode.day2._02_di_anno.OtherBean"/>

<bean id="someBean" class="cn.wolfcode.day2._02_di_anno.SomeBean"/>
```

2.4.1.2. Resource 注解

JavaEE 规范提供

1，可以让 Spring 自动的把属性需要的对象找出来，并注入到对象。

2，可以贴在字段或者 setter 方法上面。

3，可以注入一些 Spring 内置的重要对象,比如 BeanFactory,ApplicationContext，ServletContext 等。

4，Resource 注解必须要能找到对应的对象,否则报错。

5，注解解析器，Spring3.0 之前,需要手动配置 Resource 注解的解析程序，在 Web 开发中必须配置。

```
<context:annotation-config />
```

6，Resource 注解寻找 bean 的方式：

1)，首先按照名字去找,如果找到,就使用 setter 方法或者字段注入。

2)，如果按照名字找不到,再按照类型去找,但如果找到多个匹配类型，报错。

3)，可以直接使用 name 属性指定 bean 的名称（@Resource(name="")）;但是,如果指定的 name,就只能按照 name 去找,如果找不到,就不会再按照类型去找。

Java 代码：

```
public class SomeBean {
  @Resource(name="otherBean1")
  private OtherBean other1;
  @Resource(name="otherBean2")
  private OtherBean other2;

  public String toString() {
    return "SomeBean [other1=" + other1 + ", other2=" + other2 + "]";
  }
}
```

XML 配置和 Autowired 相同

2.4.1.3. Value 注解

Autowired 和 Resource 注解用于注入对象，Value 注解用于注入常量数据（简单类型数据）。

server.properties 文件

```
server.port=8888
server.path=/
```

Java 代码：

```
@Value("${server.port}")
private int port;
```

引入配置文件：

```
<context:property-placeholder
        location="classpath:db.properties,classpath:server.properties"/>
```

或

```
<context:property-placeholder
        location="classpath:db.properties" ignore-unresolvable="true"/>
<context:property-placeholder
        location="classpath:server.properties" ignore-unresolvable="true"/>
```

2.4.2.IoC 注解

bean 组件版型：四个组件的功能是相同的，只是用于标注不同类型的组件。

@Component 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Repository 用于标注数据访问组件，即 DAO 组件。

@Service 用于标注业务层组件。

@Controller 用于标注控制层组件（如 struts 中的 Ation，SpringMVC 的 Controller）。

此时需要配置 IoC 注解的解析器：

使用<context:component-scan base-package=""/>

表示去哪些包中及其子包中去扫描组件注解。

Java 代码：

```
@Component
public class OtherBean {
}

@Component
public class SomeBean {
```

```
  @Autowired
  private OtherBean other;

  public String toString() {
    return "SomeBean [other=" + other + "]";
  }
}
```

XML 配置：

```
<!-- DI 注解解释器 -->
  <context:annotation-config/>
<!-- IoC 注解解释器 -->
<context:component-scan base-package="cn.wolfcode.day2._03_ioc_anno"/>
```

2.4.3.作用域注解

使用 Scope 注解，设置 Bean 的作用域。

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class OtherBean {
}
```

2.4.4.初始化和销毁注解

@PostConstruct 用于贴在初始化方法上

@PreDestroy 用于贴在销毁方法上

```
@Component
public class SomeBean {
  public SomeBean() {
    System.out.println("创建 SomeBean 对象");
  }
  @PostConstruct
  public void open() {
    System.out.println("初始化方法");
  }
  @PreDestroy
  public void close() {
    System.out.println("销毁方法");
  }
  public void doWork() {
    System.out.println("工作");
  }
}
```

2.4.5. 注册案例

2.5. Java 代码配置 IoC 和 DI

2.5.1. Configuration 注解

2.5.2. Bean 注解

2.5.3. ComponentScan 注解

2.5.4. Import 注解

2.5.5. ImportResource 注解

2.5.6. PropertySource 注解

2.5.7. Profile 注解

2.5.8. ActiveProfile 注解

3. 第三章 Spring 帝国之军-AOP

3.1. 案例分析

3.1.1.引出问题

回到 Spring 之初控制事务繁琐的问题。

考虑一个应用场景：需要对系统中的某些业务方法做事务管理，拿简单的 save 和 update 操作举例。没有加上事务控制的代码如下。

```
public class EmployeeServiceImpl implements IEmployeeService {
  public void save() {
    //保存操作
  }
}
```

修改源代码，加上事务控制之后：

```
public class EmployeeServiceImpl implements IEmployeeService {
  public void save() {
    //打开资源
    //开启事务
    try {
       //保存操作
       //提交事务
    } catch (Exception e) {
       //回滚事务
    } finally {
       //释放资源
    }
  }
}
```

上述问题：在我们的业务层中每一个业务方法都得处理事务(繁琐的 try-catch)。

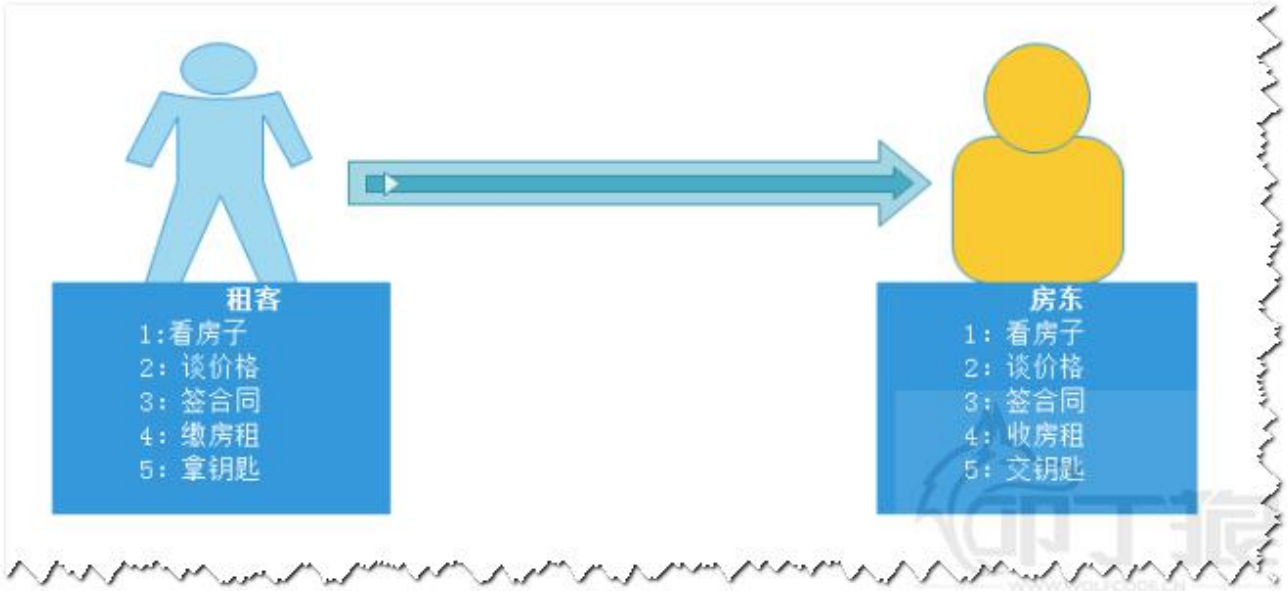
在设计上存在两个很严重问题：

1，责任不分离.业务方法只需要关心如何完成该业务功能,不需要去关系事务管理/日志管理/权限管理等等。

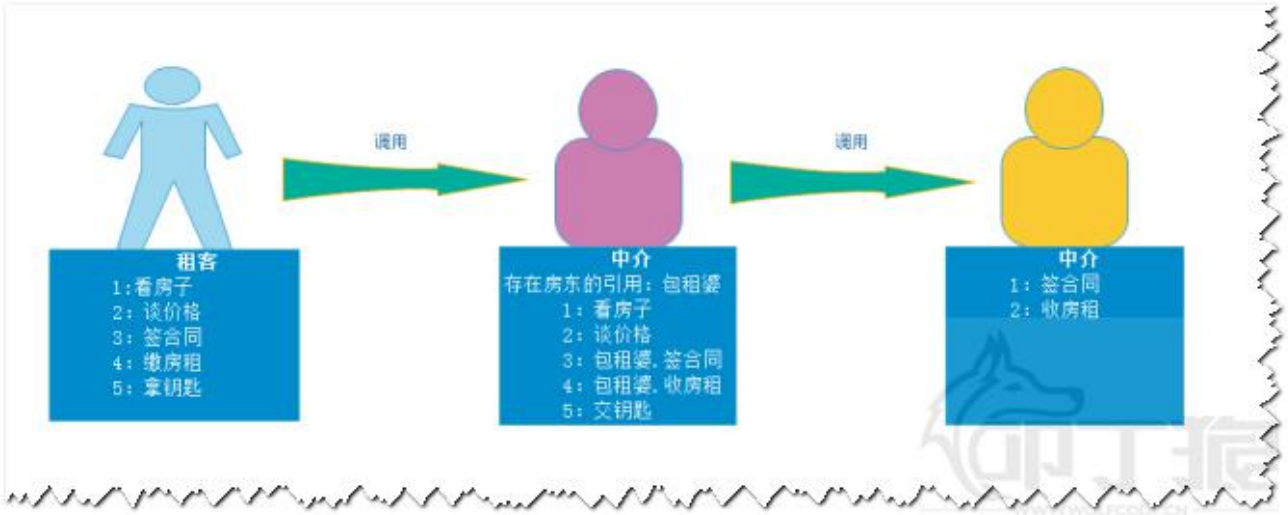
2，代码结构重复.在开发中不要重复代码,重复就意味着维护成本增大。

3.1.2.房屋租赁的启示

租房情况一：直接联系房东：



租房情况二：通过中介：



3.2. 静态代理

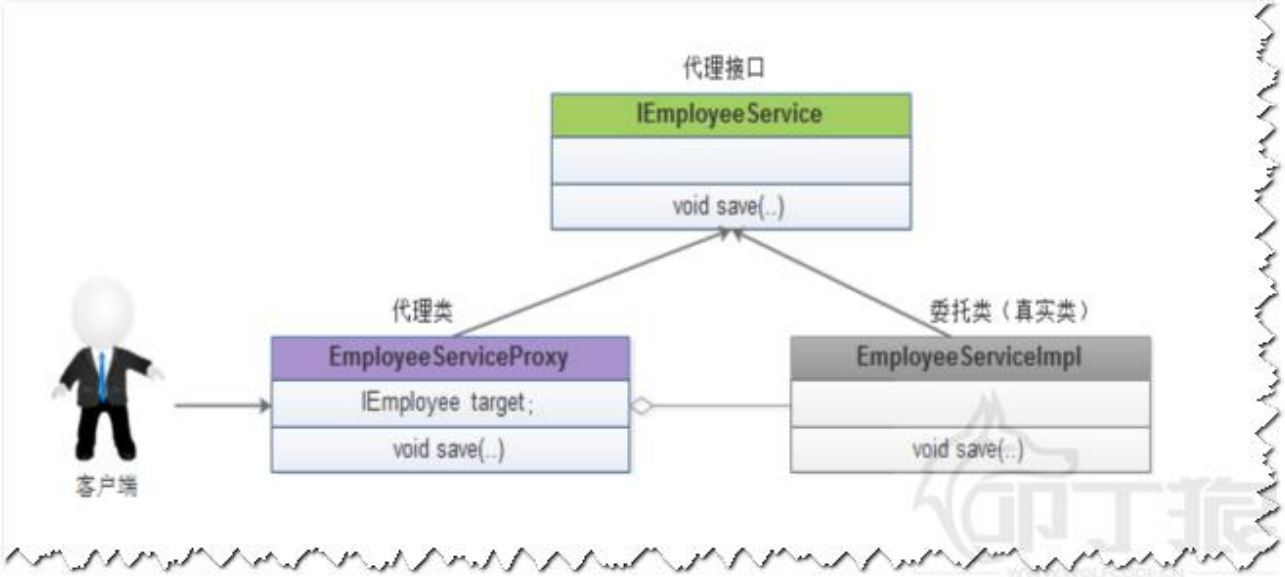
3.2.1.代理实现

代理模式：客户端直接使用的都是代理对象，不知道真实对象是谁，此时代理对象可以在客户端和真实对象之间起到中介的作用。

1、代理对象完全包含真实对象，客户端使用的都是代理对象的方法，和真实对象没有直接关系；

2、代理模式的职责：把不是真实对象该做的事情从真实对象上撇开——职责清晰；

静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和真实对象的关系在运行前就确定了。



Java 代码：

```
public class EmployeeServiceProxy implements IEmployeeService {
  private IEmployeeService target;
  private TransctionManager txManager;

  public void setTarget(IEmployeeService target) {
    this.target = target;
  }

  public void setTxManager(TransctionManager txManager) {
    this.txManager = txManager;
  }

  public void save(Employee e) {
    txManager.begin();
    try {
      target.save(e);
      txManager.commit();
    } catch (Exception ex) {
      ex.printStackTrace();
      txManager.rollback();
    }
  }

  public void update(Employee e) {
```

```
        txManager.begin();
        try {
           target.update(e);
           txManager.commit();
        } catch (Exception ex) {
           ex.printStackTrace();
           txManager.rollback();
        }
      }
    }
```

XML 配置：

```
    <bean id="transctionManager" class="cn.wolfcode.wms.tx.TransctionManager" />

    <bean id="employeeDAO" class="cn.wolfcode.wms.dao.impl.EmployeeDAOImpl" />

    <bean id="employeeServiceProxy" class="cn.wolfcode.wms.proxy.EmployeeServiceProxy">
      <property name="txManager" ref="transctionManager" />
      <property name="target">
        <bean class="cn.wolfcode.wms.service.impl.EmployeeServiceImpl">
           <property name="dao" ref="employeeDAO" />
        </bean>
      </property>
    </bean>
```

3.2.2.问题分析

静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和真实对象的关系在运行前就确定了。

优点：

1，业务类只需要关注业务逻辑本身，保证了业务类的重用性。

2，把真实对象隐藏起来了,保护真实对象

缺点：

1，代理对象的某个接口只服务于某一种类型的对象，也就是说每一个真实对象都得创建一个代理对象。

2，如果需要代理的方法很多，则要为每一种方法都进行代理处理。

3，如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。

3.3. 动态代理

代理模式：客户端直接使用的都是代理对象，不知道真实对象是谁，此时代理对象可以在客户端和真实对象之间起到中介的作用。

1、代理对象完全包含真实对象，客户端使用的都是代理对象的方法，和真实对象没有直接关系；

2、代理模式的职责：把不是真实对象该做的事情从真实对象上撇开——职责清晰；

静态代理：在程序运行前就已经存在代理类的字节码文件，代理对象和真实对象的关系在运行前就确定了。

动态代理：动态代理类是在程序运行期间由 JVM 通过反射等机制动态的生成的，所以不存在代理类的字节码文件，代理对象和真实对象的关系是在程序运行时期才确定的。

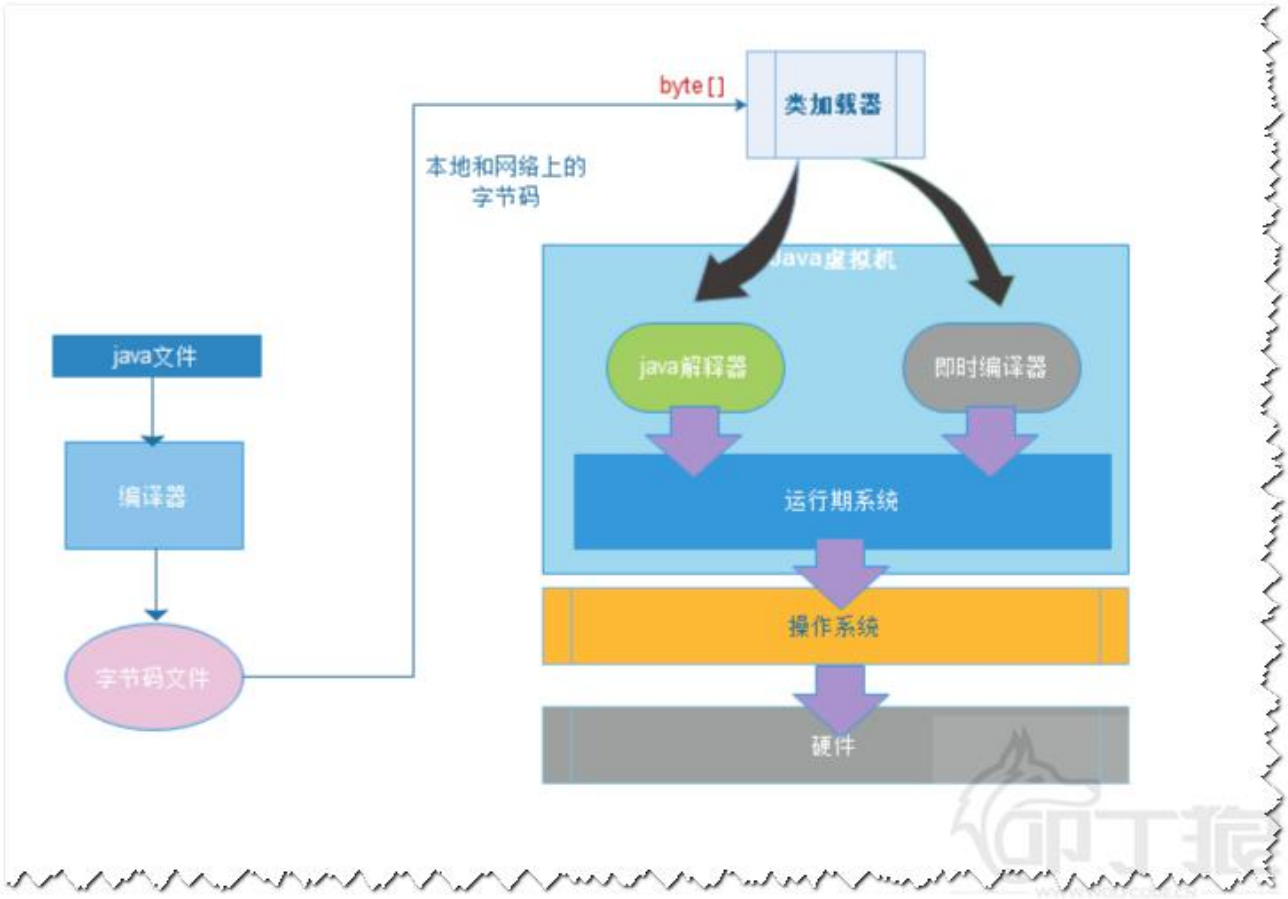
如何实现动态代理：

1）：针对有接口：使用 JDK 动态代理

2）：针对无接口：使用 CGLIB 或 Javassist 组件

3.3.1.字节码动态加载

Java 运行原理和字节码加载过程：



如何动态的加载一份字节码：

由于 JVM 通过字节码的二进制信息加载类的，如果我们在运行期系统中，遵循 Java 编译系统组织.class 文件的格式和结构，生成相应的二进制数据，然后再把这个二进制数据加载转换成对应的类。

如此，就完成了在代码中动态创建一个类的能力了。

3.3.2.JDK 动态代理

JDK 动态代理 API 分析:(必须要求真实对象是有接口)

1、java.lang.reflect.Proxy 类:Java 动态代理机制生成的所有动态代理类的父类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。

主要方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler hanlder)
```

方法职责：为指定类加载器、一组接口及调用处理器生成动态代理类实例

参数:

loader :类加载器，一般传递真实对象的类加载器

interfaces :代理类需要实现的接口

hanlder :代理对象如何做增强:

返回：创建的代理对象

2、java.lang.reflect.InvocationHandler 接口:

```
public Object invoke(Object proxy, Method method, Object[] args)
```

方法职责：负责集中处理动态代理类上的所有方法调用

参数:

proxy :生成的代理对象

method :当前调用的真实方法对象

args :当前调用方法的实参

返回: 真实方法的返回结果 n

jdk 动态代理操作步骤 ：

① 实现 InvocationHandler 接口，创建自己增强代码的处理器。

② 给 Proxy 类提供 ClassLoader 对象和代理接口类型数组，创建动态代理对象。

③ 在处理器中实现增强操作。

Java 代码：

```
public class TransctionManagerInvocationHandler
                        implements java.lang.reflect.InvocationHandler {
  private Object target;
  private TransctionManager txManager;

  public void setTarget(Object target) {
    this.target = target;
  }
```

```
    public void setTxManager(TransctionManager txManager) {
      this.txManager = txManager;
    }

    public <T> T getProxyObject() {
      return (T) Proxy.newProxyInstance(target.getClass().getClassLoader(),
          target.getClass().getInterfaces(),
          this);
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
      Object ret = null;
      txManager.begin();
      try {
         ret = method.invoke(target, args);
         txManager.commit();
      } catch (Exception ex) {
         ex.printStackTrace();
         txManager.rollback();
      }
      return ret;
    }
  }
```

XML 配置：

3.3.3.JDK 动态代理原理

通过 DynamicProxyClassGenerator 生成动态代理的字节码，再通过反编译工具查看。

一，生成动态代理字节码。

```
  public class DynamicProxyClassGenerator {
    public static void main(String[] args) throws Exception {
      generateClassFile(EmployeeServiceImpl.class, "EmployeeServiceProxy");
    }
    public static void generateClassFile(Class targetClass, String proxyName)
        throws Exception {
      //根据类信息和提供的代理类名称，生成字节码
      byte[] classFile =
            ProxyGenerator.generateProxyClass(proxyName, targetClass.getInterfaces());
      String path = targetClass.getResource(".").getPath();
      System.out.println(path);
      FileOutputStream out = null;
      //保留到硬盘中
      out = new FileOutputStream(path + proxyName + ".class");
```

```
      out.write(classFile);
      out.close();
    }
  }
```

二，通过反编译工具，查看字节码文件（删除异常处理代码）：

```
  public final class EmployeeServiceProxy extends Proxy implements IEmployeeService {
    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m4;
    private static Method m0;

    public EmployeeServiceProxy(InvocationHandler paramInvocationHandler) {
      super(paramInvocationHandler);
    }

    public final int hashCode() {
      return ((Integer) this.h.invoke(this, m0, null)).intValue();
    }

    public final String toString() {
      return (String) this.h.invoke(this, m2, null);
    }

    public final boolean equals(Object paramObject) {
      return       ((Boolean)       this.h.invoke(this,       m1,       new       Object[]
{ paramObject })).booleanValue();
    }

    public final void save(Employee paramEmployee) {
      this.h.invoke(this, m4, new Object[] { paramEmployee });
    }

    public final void update(Employee paramEmployee) {
      this.h.invoke(this, m3, new Object[] { paramEmployee });
    }

    static {
      m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
      m1 = Class.forName("java.lang.Object").getMethod("equals",
                          new Class[] { Class.forName("java.lang.Object") });
      m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
```

```
      String className = "cn.wolfcode.wms.service.IEmployeeService";
      String argType = "cn.wolfcode.wms.domain.Employee";
      m3 = Class.forName(className).getMethod("update",
                         new Class[] { Class.forName(argType) });
      m4 = Class.forName(className).getMethod("save",
                         new Class[] { Class.forName(argType) });
    }
  }
```

观察：save 方法，发现底层其实依然在执行 InvocationHandler 中的 invoke 方法。

```
  public final void save(Employee paramEmployee) {
    this.h.invoke(this, m4, new Object[] { paramEmployee });
  }
```

演示打印代理对象的死循环和调用 getClass 方法。

3.3.4. CGLIB 动态代理

使用 JDK 的动态代理，只能针对于目标对象存在接口的情况，如果目标对象没有接口，此时可以考虑使用 CGLIB 的动态代理方式。

Java 代码：

```
public class TransctionManagerInvocationHandler
                          implements org.springframework.cglib.proxy.InvocationHandler {
  private Object target;
  private TransctionManager txManager;

  public void setTarget(Object target) {
    this.target = target;
  }

  public void setTxManager(TransctionManager txManager) {
    this.txManager = txManager;
  }

  public <T> T getProxyObject() {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(target.getClass());
    enhancer.setCallback(this);
    return (T) enhancer.create();
  }

  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object ret = null;
```

```
        txManager.begin();
        try {
           ret = method.invoke(target, args);
           txManager.commit();
        } catch (Exception ex) {
           ex.printStackTrace();
           txManager.rollback();
        }
        return ret;
      }
    }
```

3.3.5. CGLIB 动态代理原理

观察 CGLIB 生成的动态代理字节码文件(使用 xjad 反编译工具)：

System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,"C:/test");

CGLIB 生成的动态代理字节码文件，阅读比较复杂，不容易读，我们见到看一下，下面是经过优化处理的代码。

```
    public class EmployeeServiceImpl$$EnhancerByCGLIB$$12345
                              extends EmployeeServiceImpl implements Factory {
      private boolean CGLIB$BOUND;
      public static Object CGLIB$FACTORY_DATA;
      private static ThreadLocal CGLIB$THREAD_CALLBACKS;
      private static Callback CGLIB$STATIC_CALLBACKS[];
      private InvocationHandler CGLIB$CALLBACK_0;
      private static Object CGLIB$CALLBACK_FILTER;
      private static Method CGLIB$update$0;
      private static Method CGLIB$save$1;
      private static Method CGLIB$setDao$2;
      private static Method CGLIB$equals$3;
      private static Method CGLIB$toString$4;
      private static Method CGLIB$hashCode$5;
      private static Method CGLIB$clone$6;

      static {
        try {
           CGLIB$THREAD_CALLBACKS = new ThreadLocal();
           String className = "cn.wolfcode.wms.service.impl.EmployeeServiceImpl";
           String argType = "cn.wolfcode.wms.domain.Employee";
           CGLIB$update$0 = Class.forName(className).getDeclaredMethod("update",
              new Class[] { Class.forName(argType) });
           CGLIB$save$1  =  Class.forName(className).getDeclaredMethod("save",  new  Class[]
{ Class.forName(argType) });
```

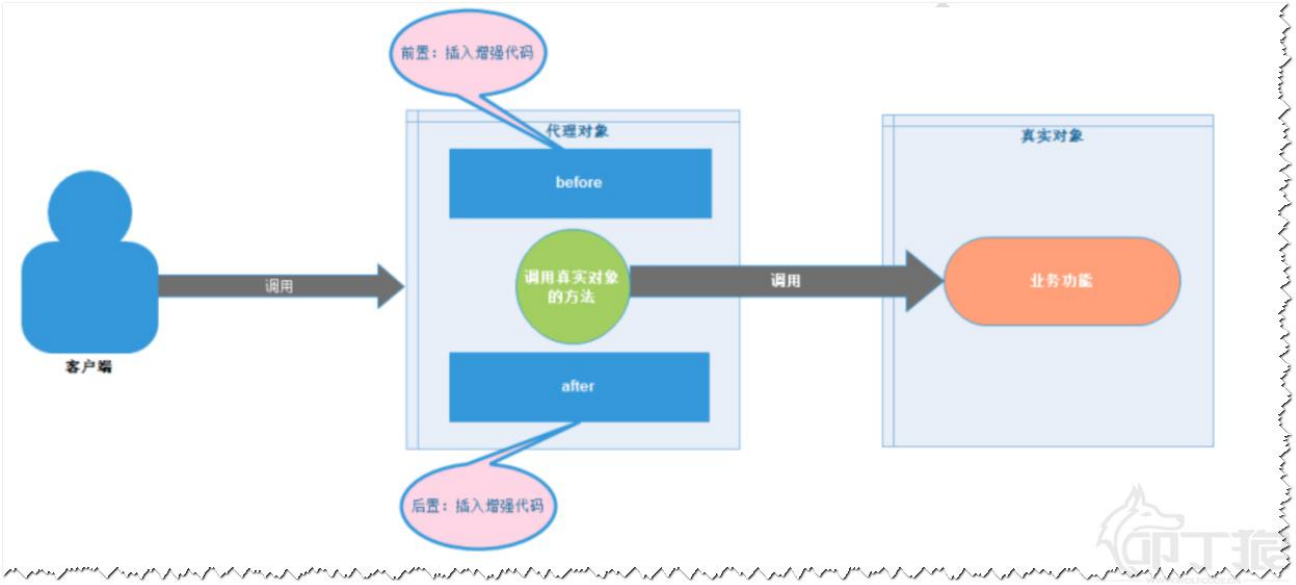
```
            CGLIB$setDao$2 = Class.forName(className).getDeclaredMethod("setDao",
                new Class[] { Class.forName("cn.wolfcode.wms.dao.IEmployeeDAO") });
            CGLIB$equals$3 = Class.forName("java.lang.Object").getDeclaredMethod("equals",
                new Class[] { Class.forName("java.lang.Object") });
            CGLIB$toString$4 = Class.forName("java.lang.Object").getDeclaredMethod("toString",
new Class[0]);
            CGLIB$hashCode$5 = Class.forName("java.lang.Object").getDeclaredMethod("hashCode",
new Class[0]);
            CGLIB$clone$6 = Class.forName("java.lang.Object").getDeclaredMethod("clone", new
Class[0]);
        } catch (Exception e) {

        }
      }

      public final void save(Employee employee)
      {
        CGLIB$CALLBACK_0;
        if (CGLIB$CALLBACK_0 != null) goto _L2; else goto _L1
    _L1:
        JVM INSTR pop ;
        CGLIB$BIND_CALLBACKS(this);
        CGLIB$CALLBACK_0;
    _L2:
        this;
        CGLIB$save$1;
        new Object[] {
          employee
        };
        invoke();
        return;
        throw ;
        JVM INSTR new #55  <Class UndeclaredThrowableException>;
        JVM INSTR dup_x1 ;
        JVM INSTR swap ;
        UndeclaredThrowableException();
        throw ;
      }
    }
```

观察：可以看出 CGLIB 是通过生成代理类，然后继承于目标类，再对目标类中可以继承的方法做覆盖，并在该方法中做功能增强的，因为多台的关系，实则调用的是子类中的方法。

3.3.6.拦截器思想

3.4. 代理总结

3.4.1.动态代理原理总结



JDK 动态代理总结：

1，JAVA 动态代理是使用 java.lang.reflect 包中的 Proxy 类与 InvocationHandler 接口这两个来完成的。

2，要使用 JDK 动态代理，委托必须要定义接口。

3，JDK 动态代理将会拦截所有 pubic 的方法（因为只能调用接口中定义的方法），这样即使在接口中增加了新的方法，不用修改代码也会被拦截。

4，动态代理的最小单位是类(所有类中的方法都会被处理)，如果只想拦截一部分方法，可以在 invoke 方法中对要执行的方法名进行判断。

CGLIB 代理总结：

1,CGLIB 可以生成委托类的子类，并重写父类非 final 修饰符的方法。

2,要求类不能是 final 的，要拦截的方法要是非 final、非 static、非 private 的。

3,动态代理的最小单位是类(所有类中的方法都会被处理);

3.4.2. 性能和选择

JDK 动态代理是基于实现接口的，CGLIB 和 Javassit 是基于继承委托类的。

从性能上考虑：Javassit > CGLIB > JDK

Struts2 的拦截器和 Hibernate 延迟加载对象，采用的是 Javassit 的方式.

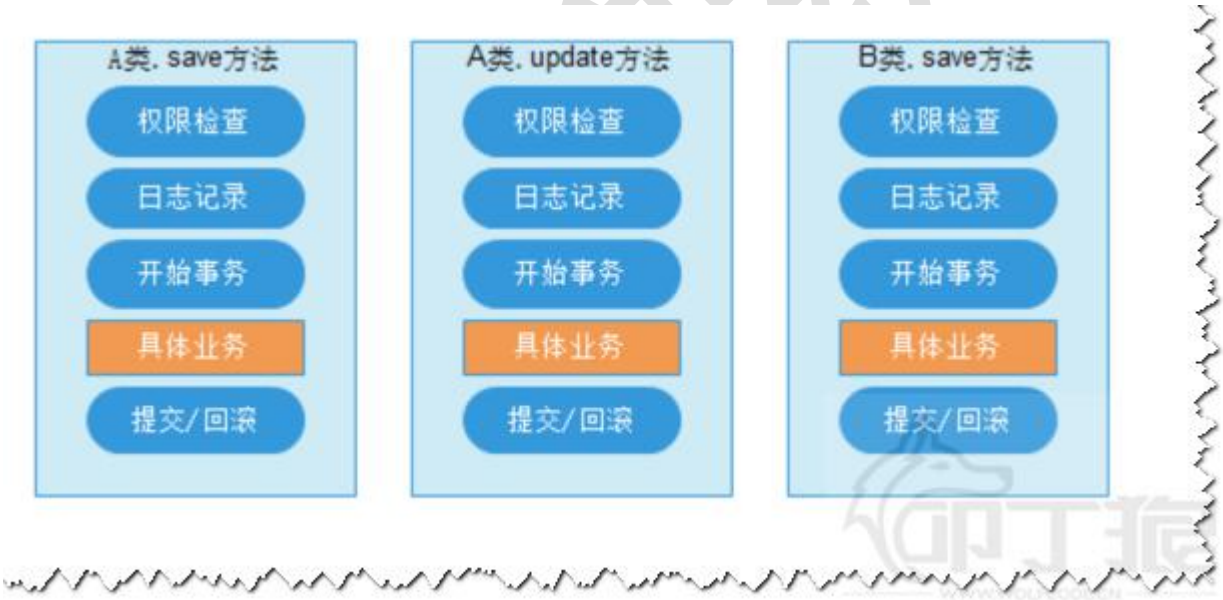
对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统，也更符合面向接口编程规范。

若委托对象实现了干接口，优先选用 JDK 动态代理。

若委托对象没有实现任何接口，使用 Javassit 和 CGLIB 动态代理。

3.5. AOP 思想

3.5.1. 从 OOP 到 AOP



在开发中，为了给业务方法中增加日志记录，权限检查，事务控制等功能，此时我们需要去修改业务方法代码，考虑到代码的重用性，

我们可以考虑使用 OOP 的继承或组合关系来消除重复，但是无论怎么样，我们都会在业务方法中纵向地增加这些功能方法的调用代码。

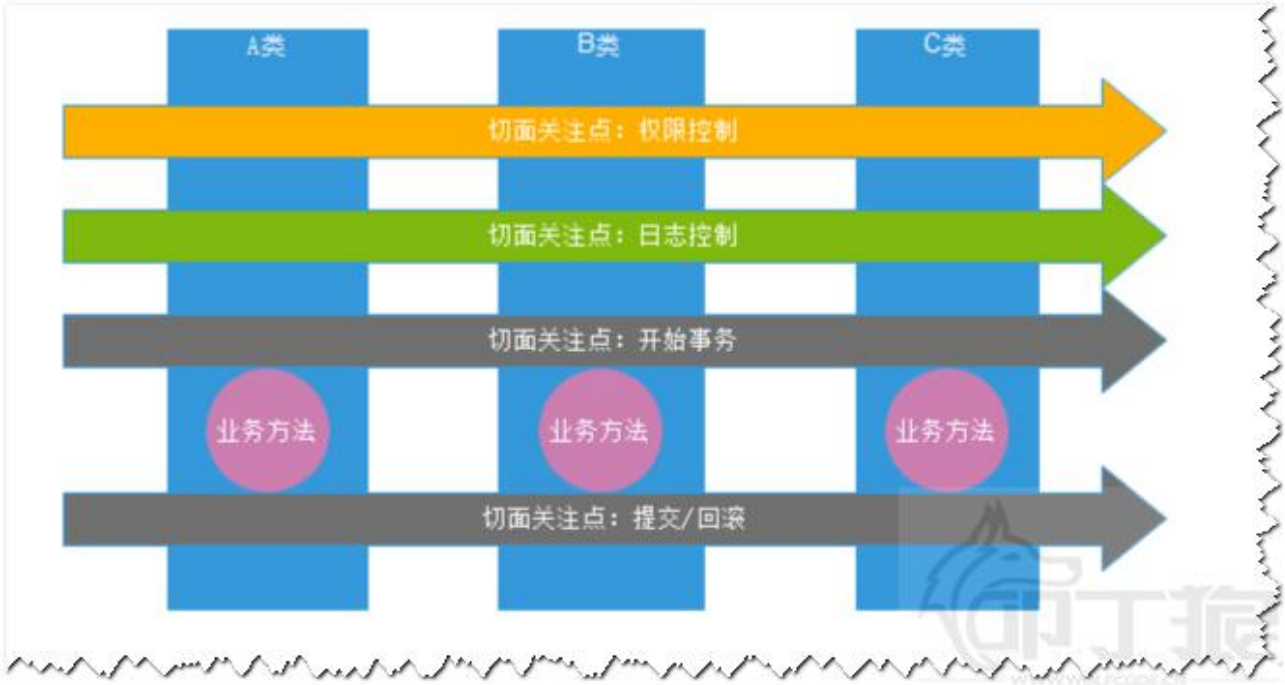
此时，既不遵循开闭原则，也会为后期系统的维护带来很大的麻烦。

这些零散存在于业务方法中的功能代码，我们称之为横切面关注点，横切面关注点不属于业务范围，应该从业务代码中剥离出来。

为了解决该问题，OOP 思想是不行了，得使用 AOP 思想。

AOP(Aspect Oritention Programming):

把一个个的横切关注点放到某个模块中去，称之为切面。那么每一个的切面都能影响业务的某一种功能，切面的目的就是功能增强，如日志切面就是一个横切关注点，应用中许多方法需要做日志记录的只需要插入日志的切面即可。



这种面向切面编程的思想就是 AOP 思想了。

3.5.2. AOP 术语

Joinpoint:连接点，被拦截到需要被增强的方法。where：去哪里做增强

Pointcut：切入点，哪些包中的哪些类中的哪些方法，可认为是连接点的集合。where:去哪些地方做增强

Advice：增强，当拦截到 Joinpoint 之后，在方法执行的什么时机（when）做什么样(what)的增强。根据时机分为：前置增强、后置增强、异常增强、最终增强、环绕增强

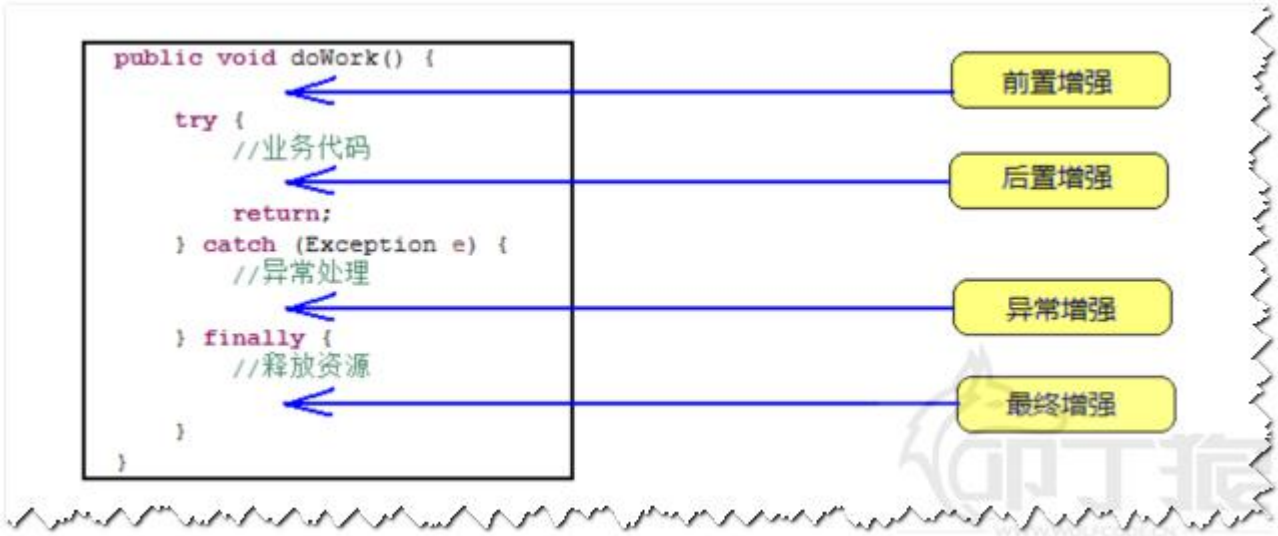
Aspect：切面，Pointcut+Advice，去哪些地方+在什么时机+做什么增强

Target：目标对象,被代理的目标对象

Weaving：织入，把 Advice 加到 Target 上之后，创建出 Proxy 对象的过程。

Proxy：一个类被 AOP 织入增强后，产生的代理类

Advice（增强）执行时机：



3.5.3. Pointcot 语法

AOP 的规范本应该由 SUN 公司提出，但是被 AOP 联盟捷足先登。AOP 联盟在制定 AOP 规范时，首先就要解决一个问题——怎么表示切入点。也就是说怎么表达需要在哪些方法上做增强。这是一个如何表示 WHERE 的问题，应该有一些思考。

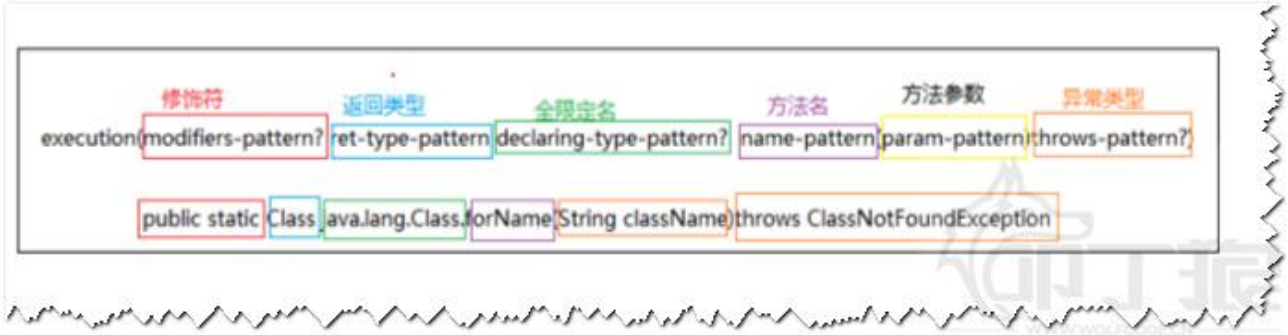
AOP 规范知道后，面向切面编程的框架 AspectJ 也就应运而生了，同时也确定如何去表达这个 WHERE。

AspectJ 切入点语法如下(表示在哪些包下的哪些类中的哪些方法上做切入增强)：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?
                                                    name-pattern(param-pattern)throws-pattern?)
```

翻译成中文：

```
execution(<修饰符>? <返回类型> <声明类型>? <方法名>(<参数>) <异常>?)
```



举例：

```
public static Class java.lang.Class.forName(String className)throws ClassNotFoundException
```

切入点表达式中的通配符：

*：匹配任何部分，但是只能表示一个 world。

..：可用于全限定名中和方法参数中，分别表示子包和 0 到 N 个参数。

常见的写法：

```
execution(* cn.wolfcode.wms.service.*.*(..))
execution(* cn.wolfcode.wms.service.*Service.*(..))
execution(* cn.wolfcode..service.*Service.*(..))
```

3.6. AOP 开发

依赖的 jar:

```
spring-aop-版本.RELEASE.jar
com.springsource.org.aopalliance-1.0.0.jar
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
```

spring5 开始在 spring-aop 库中纳入了 AOP 联盟的 API，不再需要拷贝 aopalliance-1.0.0.jar。

AOP 配置三部曲：where-when-what

3.6.1. 使用 JDK 动态代理

使用 JDK 动态代理（推荐）。

```
<!-- 配置 AOP-->
<aop:config>
  <!--what:增强了什么 -->
  <aop:aspect ref="transctionManager">
    <!--where:哪些包下的哪些类中的哪些方法上增强 -->
    <aop:pointcut expression="
                execution(* cn.wolfcode.wms.service.*Service.*(..))" id="txPoint" />
    <!--when:在方法的前/后增强 -->
    <aop:before method="begin" pointcut-ref="txPoint" />
    <aop:after-returning method="commit" pointcut-ref="txPoint" />
    <aop:after-throwing method="rollback" pointcut-ref="txPoint" />
  </aop:aspect>
</aop:config>
```

3.6.2. 使用 CGLIB 动态代理

强制使用 CGLIB 去做动态代理。

设置 proxy-target-class 属性为 true，同时切入点表达式需要修改。

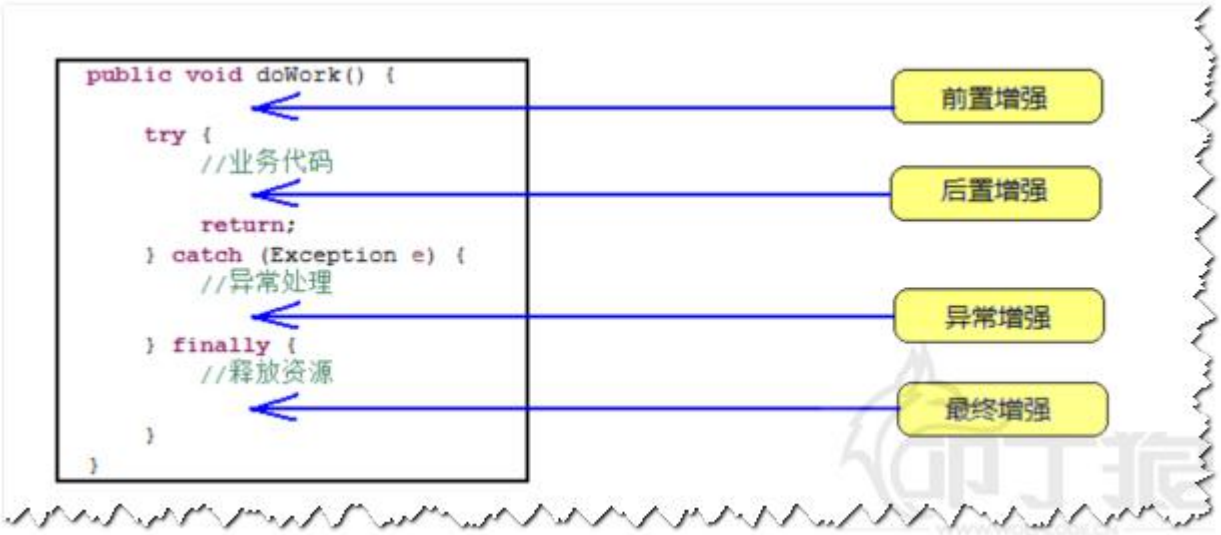
```
<aop:config proxy-target-class="true">
   <aop:aspect ref="transctionManager">
     <aop:pointcut expression="
               execution(* cn.wolfcode.wms.service..*(..))" id="txPoint" />
```

```
        <aop:before method="begin" pointcut-ref="txPoint" />
        <aop:after-returning method="commit" pointcut-ref="txPoint" />
        <aop:after-throwing method="rollback" pointcut-ref="txPoint" />
    </aop:aspect>
</aop:config>
```

3.7. 增强细节

3.7.1.各种时机的增强

增强的时机：



根据增强的时机不同，化为为多种增强机制。

增强	备注
aop:before（前置增强）	在方法执行之前执行增强
aop:after-returning（后置增强）	在方法正常执行完成之后执行增强(中间没有遇到任何异常)
aop:throwing（异常增强）	在方法抛出异常退出时执行增强
aop:after（最终增强）	在方法执行之后执行，相当于在finally里面执行；可以通过配置throwing来获得拦截到的异常信息
aop:around（环绕增强）	最强大的一种增强类型，环绕增强可以在方法调用前/后完成自定义的行为，环绕增强有两个要求： 1，方法需要要返回一个Object（返回的结果） 2，方法的第一个参数必须是ProceedingJoinPoint（可以继续向下传递的连接点）

各种增强的应用场景。

增强类型	应用场景
前置增强（Before advice）	权限控制、 记录调用日志
后置增强（After returning advice）	统计分析结果数据
异常增强（After throwing advice）	通过日志记录方法异常信息
最终增强（After finally advice）	释放资源
环绕增强（Around Advice）	缓存、性能日志、权限、事务管理

Java 代码：

```
public class TransctionManager {
  public void begin() {
    System.out.println("开启事务");
  }

  public void commit() {
    System.out.println("提交事务");
  }

  public void close() {
    System.out.println("释放资源");
  }

  public void rollback() {
    System.out.println("回滚事务");
  }

  public Object around() {
    Object ret = null;
    System.out.println("开启事务");
    try {
       //执行目标方法
       System.out.println("执行目标方法");
       System.out.println("提交事务");
    } catch (Throwable ex) {
       System.out.println("回滚事务");
    } finally {
       System.out.println("释放资源");
    }
    return ret;
```

```
    }
}
```

XML 配置：配置 around 增强时先把其他四个增强注释掉。

```
<bean id="transctionManager" class="cn.wolfcode.wms.tx.TransctionManager" />
  <aop:config>
    <aop:aspect ref="transctionManager">
      <aop:pointcut expression="
                  execution(* cn.wolfcode.wms.service.*Service.*(..))" id="txPoint" />
      <aop:before method="begin" pointcut-ref="txPoint" />
      <aop:after-returning method="commit" pointcut-ref="txPoint" />
      <aop:after-throwing method="rollback" pointcut-ref="txPoint" />
      <aop:after method="close" pointcut-ref="txPoint" />
      <aop:around method="around"  pointcut-ref="txPoint"/>
    </aop:aspect>
</aop:config>
```

3.7.2.增强丰富参数配置

一，在增强方法中获取异常信息

XML 配置：

```
<aop:after-throwing method="rollback" pointcut-ref="txPoint" throwing="ex" />
```

Java 代码：

```
public void rollback(Throwable ex) {
   System.out.println("回滚事务" + ex);
}
```

二，获取被增强方法信息，并传递给增强方法

Spring AOP 提供 org.aspectj.lang.JoinPoint 类，作为增强方法的第一个参数。

JoinPoint：提供访问当前被增强方法的真实对象、代理对象、方法参数等数据。

ProceedingJoinPoint：JinPoint 子类，只用于环绕增强中，可以处理被增强方法。

使用 JoinPoint 类：

```
public void begin(JoinPoint jp) {
    System.out.println("代理对象: "+jp.getThis().getClass());
    System.out.println("目标对象: "+jp.getTarget().getClass());
    System.out.println("被增强方法参数: "+Arrays.toString(jp.getArgs()));
    System.out.println("当前连接点签名: "+jp.getSignature());
    System.out.println("当前连接点类型: "+jp.getKind());
    System.out.println("开启事务");
}
```

使用 ProceedingJoinPoint 类：

```
public Object around(ProceedingJoinPoint pjp) {
    Object ret = null;
    System.out.println("开启事务");
    try {
       //执行目标方法
       ret = pjp.proceed();
       System.out.println("提交事务");
    } catch (Throwable ex) {
       System.out.println("回滚事务");
    } finally {
       System.out.println("释放资源");
    }
    return ret;
}
```

3.8. 注解配置

3.8.1.使用 JDK 动态代理

Java 代码：

```
@Component
@Aspect
public class TransctionManager {
  @Pointcut("execution(* cn.wolfcode.wms.service.*Service.*(..))")
  public void txPoint() {
  }
  @org.aspectj.lang.annotation.Before("txPoint()")
  public void begin(JoinPoint jp) {
    System.out.println("开启事务");
  }
  @org.aspectj.lang.annotation.AfterReturning("txPoint()")
  public void commit() {
    System.out.println("提交事务");
  }
  @org.aspectj.lang.annotation.After("txPoint()")
  public void close() {
    System.out.println("释放资源");
  }
  @org.aspectj.lang.annotation.AfterThrowing(value = "txPoint()", throwing = "ex")
  public void rollback(Throwable ex) {
    System.out.println("回滚事务" + ex);
```

```
    }
    @org.aspectj.lang.annotation.Around(value = "txPoint()")
    public Object around(ProceedingJoinPoint pjp) {
      Object ret = null;
      System.out.println("开启事务");
      try {
         //执行目标方法
         ret = pjp.proceed();
         System.out.println("提交事务");
      } catch (Throwable ex) {
         System.out.println("回滚事务");
      } finally {
         System.out.println("释放资源");
      }
      return ret;
    }
  }
```

XML 配置：

```
  <!-- IoC 注解解释器 -->
  <context:component-scan base-package="cn.wolfcode.wms" />
  <!-- DI 注解解释器 -->
  <context:annotation-config />
  <!-- AOP 注解解释器 -->
  <aop:aspectj-autoproxy />
```

通过运行会发现在注解中 AfterReturning 和 After 注解执行顺序 和 XML 中是相反的。

3.8.2. 使用 CGLIB 动态代理

默认使用的 JDK 动态代理方式，可以设置使用 CGLIB 方式。

XML 配置：

```
  <aop:aspectj-autoproxy proxy-target-class="true"/>
```

同样，修改切入点表达式。

```
  @Pointcut("execution(* cn.wolfcode.wms.service..*(..))")
  public void txPoint() {
  }
```

4. 第四章 Spring 帝国之仓-DAO

4.1. Spring 对持久层技术支持

4.1.1.为什么需要 Spring 对持久层提供支持

Spring 对持久层技术支持：

为什么需要使用 Spring 对持久层的支持？

1）：原生操作持久层 API 方式，麻烦。

2）：Spring 对事务支持非常优秀。

序号	具体操作	传统JDBC	Spring JDBC
01	获取JDBC连接	√	
02	获取预编译语句	√	
03	定义SQL	√	√
04	执行SQL	√	
05	处理结果集	√	√
06	提交事务	√	
07	处理异常	√	
08	回滚事务	√	
09	释放资源	√	

主要是Spring对事务的管理,强悍!

传统JDBC:
1. 代码臃肿,重复.
2. 处理异常
3. 控制事务

Spring JDBC:
1. 简洁,优雅,简单
2. 运行异常
3. Spring事务管理

4.1.2. Spring 支持的持久层技术

Spring 自身并没有提供持久层框架，但是提供了和持久层技术无缝整合的 API，和整合其他技术一样，刚开始 Spring 支持很多技术和框架，后来只支持一部分技术和框架，那么另一些技术和框架都只能自己提供和整合 Spring 的插件，比如 MyBatis 和 Struts2 等。

持久化技术	Spring 提供的对应模板类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate3	org.springframework.orm.hibernate3.HibernateTemplate
Hibernate4	org.springframework.orm.hibernate4.HibernateTemplate
iBatis	org.springframework.orm.ibatis.SqlMapClientTemplate(Spring4 不再提供)
JPA	org.springframework.orm.jpa.JpaTemplate(Spring4 不再提供)

持久化技术	Spring 提供的对应 DAO 基类
JDBC	org.springframework.jdbc.core.support.JdbcDaoSupport
Hibernate3	org.springframework.orm.hibernate3.support.HibernateDaoSupport
Hibernate4	org.springframework.orm.hibernate4.support.HibernateDaoSupport（Spring3 不提供）
iBatis	org.springframework.orm.ibatis.support.SqlMapClientDaoSupport(Spring4 不再提供)
JPA	org.springframework.orm.jpa.support.JpaDaoSupport(Spring4 不再提供)

从 Spring5 开始支持 Hibernate5 版本，不在支持 Hibernate3 和 Hibernate4。

4.2. Spring JDBC

4.2.1. JDBC 模板类

- JdbcTemplate 类

包含了 JDBC 操作的模板方法，简化开发。

public int update(String sql, @Nullable Object... args)

public <T> List<T> query(String sql, RowMapper<T> rowMapper)

- NamedParameterJdbcTemplate
- 包含了通过名称占位符的模板方法，更简化开发。

public int update(String sql, Map<String, ?> paramMap)

public <T> List<T> query(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper)

```
public class EmployeeDAOImplByParam implements IEmployeeDAO {
  private NamedParameterJdbcTemplate jdbcTemplate;
  public void setDataSource(DataSource ds) {
    this.jdbcTemplate = new NamedParameterJdbcTemplate(ds);
  }
  public void save(Employee e) {
    jdbcTemplate.update("INSERT INTO t_employee(name,age) VALUES(:ename,:eage)",
       new HashMap<String,Object>() {
       {
         this.put("ename", e.getName());
         this.put("eage", e.getAge());
       }
    });
  }
}
```

4.2.2. JdbcDaoSupport 类

在 Spring 中使用 JDBC 时，DAO 可以直接继承的基类。

```
public abstract class JdbcDaoSupport extends DaoSupport {
  @Nullable
  private JdbcTemplate jdbcTemplate;
  public final void setDataSource(DataSource dataSource) {
    if (this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {
       this.jdbcTemplate = createJdbcTemplate(dataSource);
       initTemplateConfig();
    }
  }
}
```

我们开发的 DAO 实现类，直接继承 JDBCDaoSupport 即可。

```
public class EmployeeDAOImpl extends JdbcDaoSupport implements IEmployeeDAO {
  public void save(Employee e) {
    super.getJdbcTemplate().update("INSERT INTO t_employee(name,age) VALUES(?,?)",
                                   e.getName(), e.getAge());
  }
}
```

DAO 的 bean 元素依然要配置 dataSource 属性。

4.2.3.JDBC 操作

4.2.3.1. DML 操作

```
public class EmployeeDAOImpl implements IEmployeeDAO {
  private JdbcTemplate jdbcTemplate;
  public void setDataSource(DataSource ds) {
    this.jdbcTemplate = new JdbcTemplate(ds);
  }

  public void save(Employee e) {
    jdbcTemplate.update("INSERT INTO t_employee(name,age) VALUES(?,?)",
                        e.getName(), e.getAge());
  }

  public void update(Employee e) {
    jdbcTemplate.update("UPDATE t_employee SET name = ?, age = ? WHERE id = ?",
                        e.getName(), e.getAge(), e.getId());
  }

  public void delete(Long id) {
    jdbcTemplate.update("DELETE FROM t_employee WHERE id = ?", id);
  }
}
```

XML 配置：

```
<context:property-placeholder location="classpath:db.properties" />
 <bean id="dataSource" init-method="init"  destroy-method="close"
        class="com.alibaba.druid.pool.DruidDataSource" >
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
    <property name="initialSize" value="${jdbc.initialSize}" />
 </bean>

 <bean id="employeeDAO" class="cn.wolfcode.wms.dao.impl.EmployeeDAOImpl">
    <property name="dataSource" ref="dataSource" />
 </bean>
```

4.2.3.2. DQL 操作

```
public Employee get(Long id) {
    List<Employee> list = jdbcTemplate.query(
     "SELECT id,name,age FROM t_employee WHERE id = ?",
     (rs, rowNum) -> {
       Employee e = new Employee();
       e.setId(rs.getLong("id"));
       e.setName(rs.getString("name"));
       e.setAge(rs.getInt("age"));
       return e;
    }, id);
    return list.size() == 1 ? list.get(0) : null;
  }
  public List<Employee> listAll() {
    return jdbcTemplate.query("SELECT id,name,age FROM t_employee", (rs, rowNum) -> {
       Employee e = new Employee();
       e.setId(rs.getLong("id"));
       e.setName(rs.getString("name"));
       e.setAge(rs.getInt("age"));
       return e;
    });
  }
```

分析调用 query 方法，如何处理结果集。

RowMapperResultSetExtractor 类：

```
public List<T> extractData(ResultSet rs) throws SQLException {
    List<T> results = (this.rowsExpected > 0 ?
                       new ArrayList<>(this.rowsExpected) : new ArrayList<>());
    int rowNum = 0;
    while (rs.next()) {
       results.add(this.rowMapper.mapRow(rs, rowNum++));
    }
    return results;
}
```

4.3. Spring ORM

4.3.1.和 MyBatis 整合

4.3.2.和 Hibernate 整合

4.3.3.和 JPA 整合

5. 第五章 Spring 帝国之盾-TX

5.1. 引出事务

5.1.1.银行转账案例

需求：从 ID 为 10086 账户给 ID 为 10010 账户转账 1000 元钱。

DAO 代码：

```
    public class AccountDAOImpl implements IAccountDAO {
      private JdbcTemplate jdbcTemplate;

      public void setDataSource(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
      }

      public void transOut(Long outId, int money) {
        jdbcTemplate.update("UPDATE account SET balance = balance - ? WHERE id = ?", money,
outId);
      }

      public void transIn(Long inId, int money) {
        jdbcTemplate.update("UPDATE account SET balance = balance + ? WHERE id = ?", money, inId);
      }
    }
```

Service 代码：

```
    public class AccountServiceImpl implements IAccountService {
      private IAccountDAO dao;

      public void setDao(IAccountDAO dao) {
        this.dao = dao;
      }

      public void trans(Long outId, Long inId, int money) {
        dao.transOut(outId, money);
        dao.transIn(inId, money);
      }
    }
```

5.1.2.转账流程分析

分析转账过程流程:

① 首先，获取 DataSource 对象；

② 其次，获取 DataSource 中的 Connection 对象；

③ 接着，设置取消事务的自动提交方式: connection.setAutoCommit(false)；

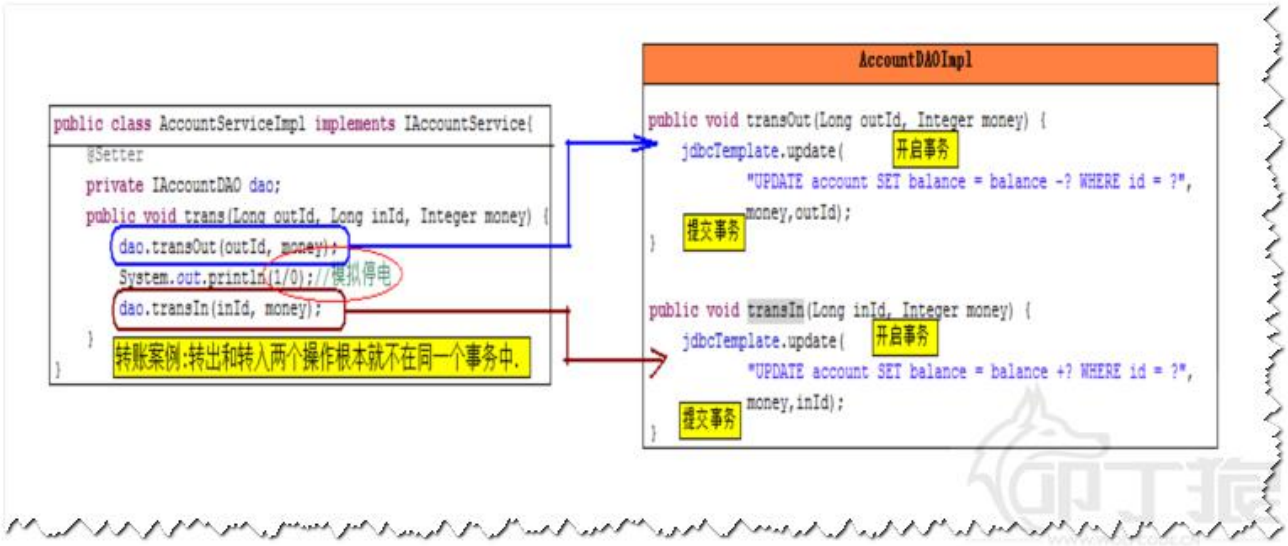
④ 然后，把 connection 绑定到当前线程中；

⑤ 从当前线程中获取 Connection 对象

⑥ 如果正常执行，则提交事务：提交事务：connection.commit()；

如果出现异常，则回滚事务：回滚事务：connection.rollback()；

出现问题的原因



怎么解决该问题？

5.2. 事务回顾

5.2.1.何为数据库事务

事务是一系列操作组成的工作单元，该工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做，这就是事务。

事务必需满足 ACID（原子性、一致性、隔离性和持久性）特性，缺一不可：

- 原子性（Atomicity）：事务是不可分割的最小工作单元，事务内的操作要么全做，要么全不做；
- 一致性（Consistency）：在事务执行前数据库的数据处于正确的状态，而事务执行完成后数据库的数

据依然处于正确的状态，即数据完整性约束没有被破坏，如 A 给 B 转账，不论转账成功与否转账之后的 A 和 B 的账户总额和转账之前是相同的。

- 隔离性（Isolation）：当多个事务处于并发访问同一个数据库资源时，事务之间相互影响程度，不同的隔离级别决定了各个事务对数据资源访问的不同行为。
- 持久性（Durability）：事务一旦执行成功，它对数据库的数据的改变是不可逆的。

5.2.2.数据库并发问题

并发问题类型	产生原因和效果
第一类丢失更新	两个事务更新相同数据，如果一个事务提交，另一个事务回滚，第一个事务的更新会被回滚
脏读	第二个事务查询到第一个事务未提交的更新数据，第二个事务根据该数据执行，但第一个事务回滚，第二个事务操作脏数据
虚读	一个事务查询到了另一个事务已经提交的新数据，导致多次查询数据不一致
不可重复读	一个事务查询到另一个事务已经修改的数据，导致多次查询数据不一致
第二类丢失更新	多个事务同时读取相同数据，并完成各自的事务提交，导致最后一个事务提交会覆盖前面所有事务对数据的改变

5.2.3.事务的隔离级别

为了解决这些并发问题，需要通过数据库隔离级别来解决，在标准 SQL 规范中定义了四种隔离级别：READ UNCOMMITED < READ COMMITED < REPEATABLE READ < SERIALIZABLE。

Oracle 支持 READ COMMITED（缺省）和 SERIALIZABLE。

MySQL 支持 四种隔离级别，缺省为 REPEATABLE READ。

隔离级别	脏读	不可重复读	幻读	第一类丢失更新	第二类丢失更新
READ UNCOMMITED	√	√	√	×	√
READ COMMITTED	×	√	√	√	√
REPREATABLE READ	×	×	√	√	√
SERIALIZABLE	×	×	×	×	×

√ 表示可能出现的情况

SQL92 推荐使用 REPREATABLE READ 以保证数据的读一致性，不过用户可以根据具体的需求选择适合的事务隔离级别。

默认情况下:MySQL不会出现幻读.
除非:select * from 表名 lock in share mode;
MySQl中锁基于索引机制,也不会出现第一类丢失更新.

如何选用：

隔离级别越高，数据库事务并发执行性能越差，能处理的操作越少。

因此在实际项目开发中为了考虑并发性能一般使用 READ COMMITED，它能避免丢失更新和脏读，尽管不可重复读和幻读不能避免，

更多的情况下使用悲观锁或乐观锁来解决。

5.2.4.事务类型

本地事务和分布式事务：

本地事务：就是普通事务，能保证单台数据库上的操作的 ACID，被限定在一台数据库上；

分布式事务：涉及多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的 ACID，使事务可以跨越多台数据库；

JDBC 事务和 JTA 事务：

JDBC 事务：就是数据库事务类型中的本地事务，通过 Connection 对象的控制来管理事务；

JTA 事务：JTA 指(Java Transaction API)，是 Java EE 数据库事务规范， JTA 只提供了事务管理接口，由应用程序服务器厂商提供实现，JTA 事务比 JDBC 更强大，支持分布式事务。

按是否通过编程实现事务有声明式事务和编程式事务；

编程式事务：通过编写代码来管理事务。

声明式事务： 通过注解或 XML 配置来管理事务；

5.3. Spring 事务管理

Spring 的事务管理主要包括 3 个接口：

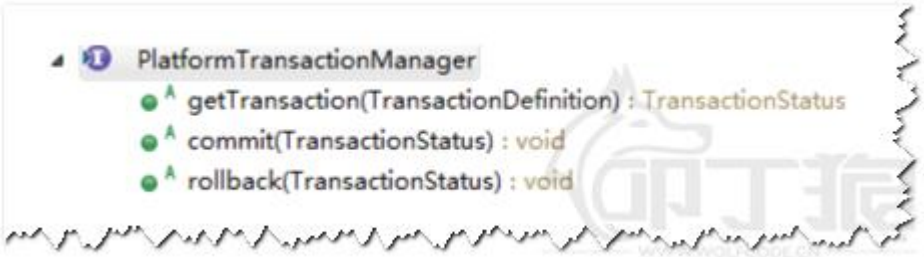
PlatformTransactionManager：根据 TransactionDefinition 提供的事务属性配置信息，创建事务。

TransactionDefinition：封装事务的隔离级别和超时时间，是否为只读事务和事务的隔离级别和传播规则等事务属性。

TransactionStatus：封装了事务的具体运行状态。如是否是新开启事务，是否已经提交事务，设置当前事务为 rollback-only 等。

5.3.1. PlatformTransactionManager

PlatformTransactionManager:接口统一抽象处理事务操作相关的方法，是其他事务管理器的规范；



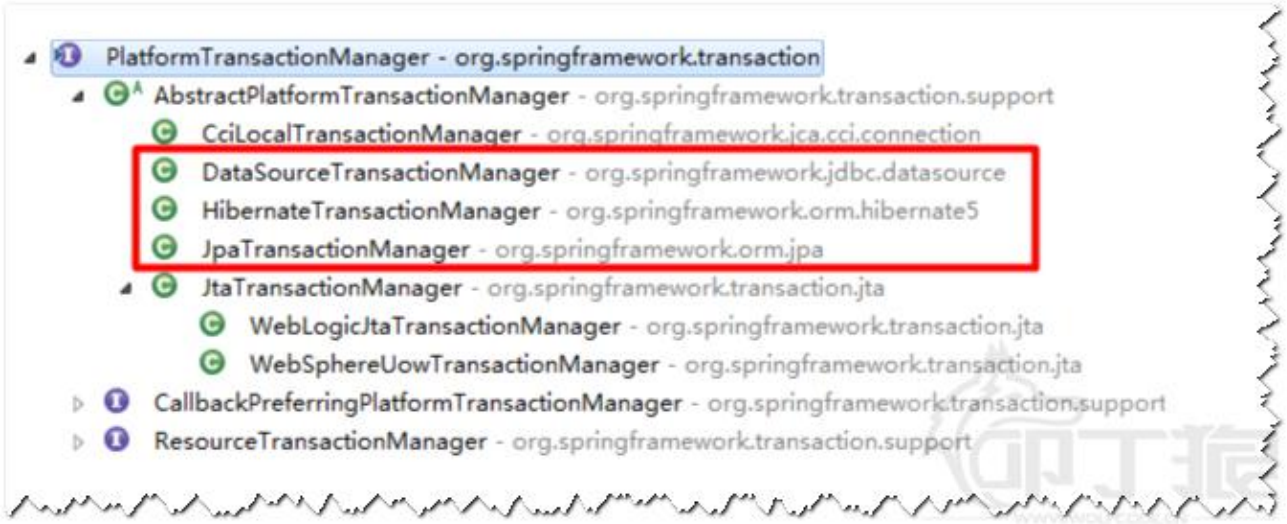
方法解释：

1):TransactionStatus getTransaction(TransactionDefinition definition)：根据事务定义信息从事务环境中返回一个已存在的事务，或者创建一个新的事务。

2):void commit(TransactionStatus status)：根据事务的状态提交事务，如果事务状态已经标识为 rollback-only，该方法执行回滚事务的操作。

3):void rollback(TransactionStatus status)：将事务回滚，当 commit 方法抛出异常时，rollback 会被隐式调用

使用 Spring 管理事务的时候，首先得告诉 Spring 使用哪一个事务管理器



常用的事务管理器：

DataSourceTransactionManager：支持 JDBC,MyBatis 器；

HibernateTransactionManager：支持 Hibernate；

5.3.2. TransactionDefinition

TransactionDefinition：封装事务的隔离级别和超时时间，是否为只读事务和事务的隔离级别和传播规则等事务属性。

事务隔离级别：用来解决并发事务时出现的问题：

ISOLATION_DEFAULT：默认隔离级别，即使用底层数据库默认的隔离级别；

ISOLATION_READ_UNCOMMITTED：未提交读；

ISOLATION_READ_COMMITTED：提交读，一般情况下我们使用这个；

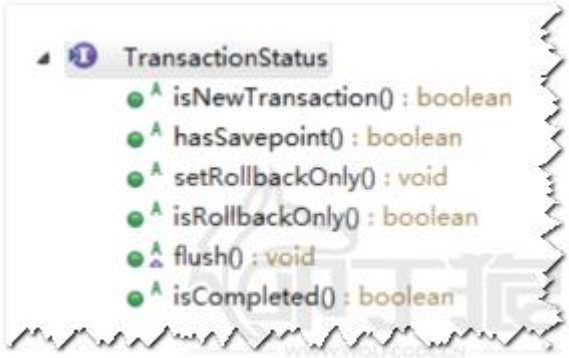
ISOLATION_REPEATABLE_READ：可重复读；

ISOLATION_SERIALIZABLE：序列化；

传播规则：

5.3.3. TransactionStatus

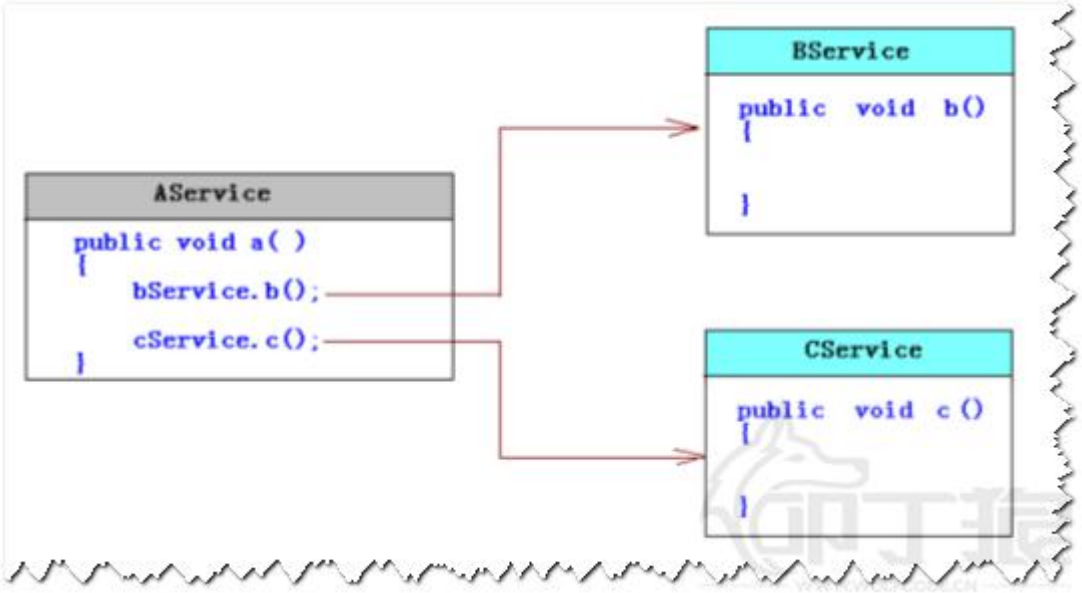
TransactionStatus：封装了事务的具体运行状态。如是否是新开启事务，是否已经提交事务，设置当前事务为 rollback-only 等。



5.3.4.事务传播规则

事务传播规则（事务传播行为）：

在一个事务方法中调用其他事务方法，此时事务该如何传播，按照什么规则传播，用谁的事务，还是都不用等等？，举个栗子。



事务是如何在这些方法间传播的，Spring 共支持 7 种传播行为：

情况一：遵从当前事务

REQUIRED：必须存在事务，如果当前存在一个事务，则加入该事务，否则将新建一个事务 （缺省）。

SUPPORTS：支持当前事务，指如果当前存在逻辑事务，就加入到该逻辑事务，如果当前没有逻辑事务，就以非事务方式执行。

MANDATORY ：必须有事务，使用当前事务执行，如果当前没有事务，则抛出异常 IllegalTransactionStateException。

情况二：不遵从当前事务

REQUIRES_NEW：不管当前是否存在事务，每次都创建新的事务

NOT_SUPPORTRD ：以非事务方式执行，如果当前存在逻辑事务，就把当前事务暂停，以非事务方式执行。

NEVER：不支持事务，如果当前存在是事务，则抛出异常，IllegalTransactionStateException

情况三：寄生事务（外部事务和寄生事务）

NESTED ：寄生事务，如果当前存在事务，则在内部事务内执行，如果当前不存在事务，则创建一个新的事务，嵌套事务使用数据库中的保存点来实现，即嵌套事务回滚不影响外部事务，但外部事务回滚将导致嵌套事务回滚。

DataSourceTransactionManager 默认支持，而 HibernateTransactionManager 默认不支持，需要手动

开启。

5.4. 事务配置

5.4.1.解决银行转账问题

```
<!-- 1，配置 JDBC 事务管理器 -->
  <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>
  <!-- 2，配置管理事务的增强 -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <!-- 设置需要事务的方法 -->
      <tx:method name="trans" />
    </tx:attributes>
  </tx:advice>
  <!-- 3，配置切面 -->
  <aop:config>
    <aop:pointcut  id="txPoint"
            expression="execution(* cn.wolfcode.wms.service.*Service.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPoint" />
  </aop:config>
```

5.4.2.事务方法属性配置

属性	必须	缺省值	描述
name	√		事务管理的方法，可使用通配符*
propagation	×	REQUIRED	事务传播规则，比如：SUPPORTS、REQUIRES_NEW
isolation	×	DEFAULT	事务隔离级别，DEFAULT:Spring 使用数据库默认的事务隔离级别、其他级别是 Spring 模拟的
read-only	×	false	事务是否只读，查询操作设置为只读事务
timeout	×	-1	事务超时时间（秒），若为-1，则有抵触事务系统决定
rollback-for	×	运行期异常	需要回滚的异常类型，多个使用逗号隔开
no-rollback-for	×	检查类型异常	不需要回滚的异常

1，name:匹配到的方法模式，必须配置；

2，read-only:如果为 true，开启一个只读事务，只读事务的性能较高，但是不能再只读事务中操作 DML；

3，isolation:代表数据库事务隔离级别(就使用默认)

DEFAULT：让 spring 使用数据库默认的事务隔离级别；

其他：Spring 模拟；

4，no-rollback-for: 如果遇到的异常是匹配的异常类型,就不回滚事务；

5，rollback-for:如果遇到的异常是指定匹配的异常类型,才回滚事务；

Spring 默认只会去回滚 RuntimeException 及其子类，不会回滚 Exception 和 Thowable

6，propagation:事务的传播方式(当一个方法已经在一个开启的事务当中了，应该怎么处理自身的事务)；

5.4.3.CRUD 通用事务配置

```
<tx:advice id="crudAdvice" transaction-manager="txManager">
    <tx:attributes>
      <!-- service 中查询方法 -->
      <tx:method name="get*" read-only="true"/>
      <tx:method name="list*" read-only="true"/>
      <tx:method name="query*" read-only="true"/>
      <!-- service 中其他方法 -->
      <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

5.5. 注解配置

5.5.1.注解的配置

Java 代码：

```
@Service
@Transactional
public class AccountServiceImpl implements IAccountService {
  @Autowired
  private IAccountDAO dao;
  public void trans(Long outId, Long inId, int money) {
    dao.transOut(outId, money);
    int a = 1 / 0;
    dao.transIn(inId, money);
  }
}
```

XML 配置：

```
<!-- IoC 注解解析器 -->
  <context:annotation-config />
  <!-- DI 注解解析器 -->
  <context:component-scan base-package="cn.wolfcode.wms" />
  <!-- TX 注解解析器 -->
  <tx:annotation-driven transaction-manager="txManager" />

  <!-- 配置 JDBC 事务管理器 -->
  <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <context:property-placeholder location="classpath:db.properties" />
  <bean id="dataSource" init-method="init"  destroy-method="close"
        class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
    <property name="initialSize" value="${jdbc.initialSize}" />
  </bean>
```

使用 CGLIB

```
<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>
```

5.5.2. 企业中的选择

5.6. Java Config 案例

5.6.1. JavaConfig 配置

连接池配置类：

```
@Configuration
@PropertySource("classpath:db.properties")
public class DataSourceConfig {
  @Value("${jdbc.driverClassName}")
  private String driverClassName;
  @Value("${jdbc.url}")
  private String url;
  @Value("${jdbc.username}")
  private String username;
  @Value("${jdbc.password}")
  private String password;
  @Value("${jdbc.initialSize}")
  private int initialSize;

  @Bean
  public DataSource dataSource() {
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName(driverClassName);
    ds.setUrl(url);
    ds.setUsername(username);
    ds.setPassword(password);
    ds.setInitialSize(initialSize);
    return ds;
  }
}
```

应用主配置类：

```
@Configuration
@Import(DataSourceConfig.class)
@EnableTransactionManagement
@ComponentScan
public class AppConfig {
```

```
    @Bean
    public DataSourceTransactionManager dataSourceTransactionManager(DataSource ds) {
      return new DataSourceTransactionManager(ds);
    }
  }
```

测试类：

```
  @SpringJUnitConfig(classes = AppConfig.class)
  public class App {
    @Autowired
    private IAccountService service;

    @Test
    void test() throws Exception {
      service.trans(10086L, 10010L, 1000);
    }
  }
```

5.6.2. 企业中的选择

Spring IoC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。

元数据的配置有三种方式（后讲）：

- 方式一，XML-based configuration
- 方式二，Annotation-based configuration
- 方式三，Java-based configuration

目前在企业中直接使用方式一的越来越少，更多的是使用方式二，或者方式二+方式一，或者方式三+方式二+方式一，SpringBoot 框架更倾向于方式三+方式二，如此更能提供开发效率，保证系统的安全性。

Spring 官方文档中写道：

Are annotations better than XML for configuring Spring?

The introduction of annotation-based configurations raised the question of whether this approach is better than XML. The short answer is it depends. The long answer is that each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It's worth pointing out that through its JavaConfig option, Spring allows annotations to be used in a non-invasive way, without touching the target components source code and that in terms of tooling, all configuration styles are supported by the Spring Tool Suite.