## 四、Spring框架

### 1、spring

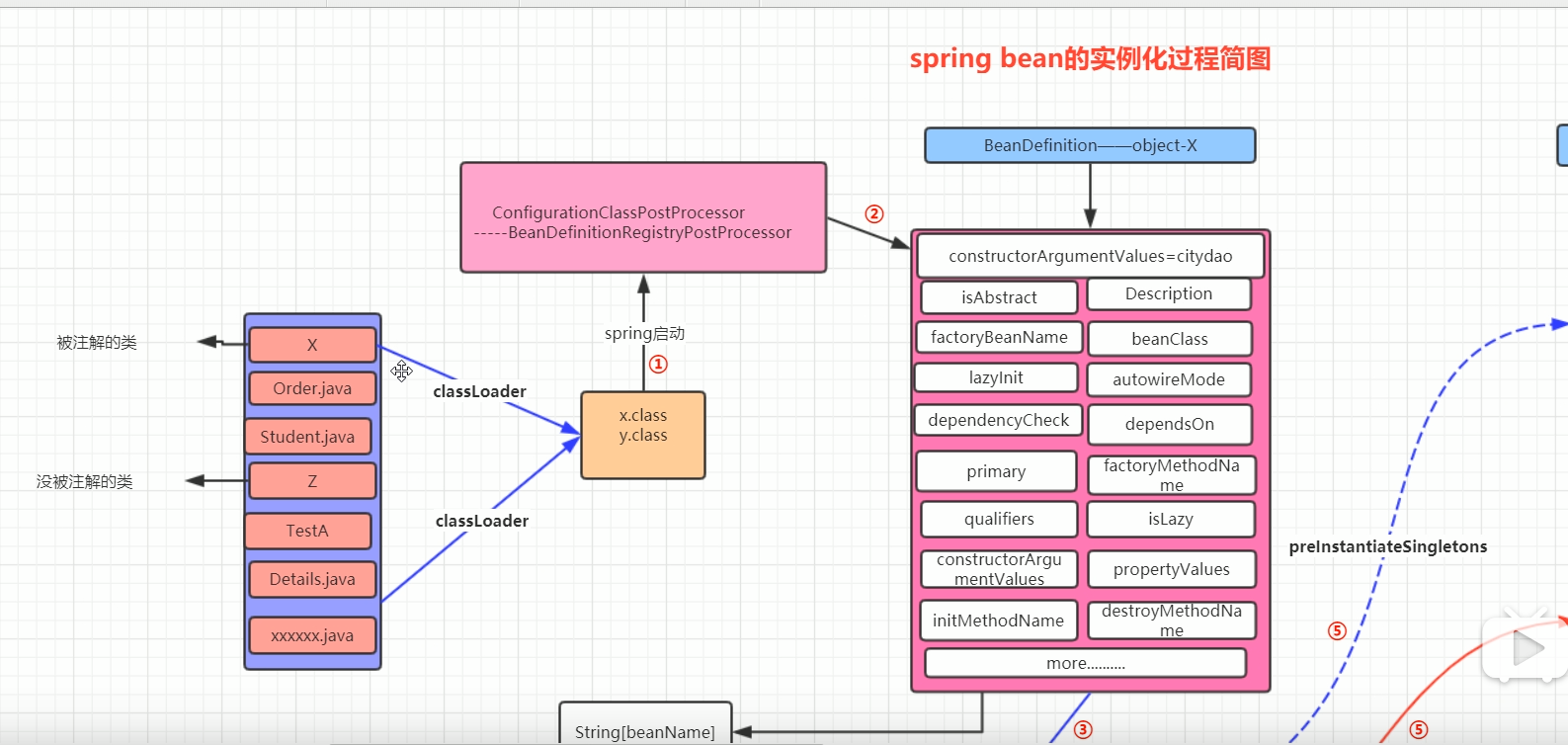
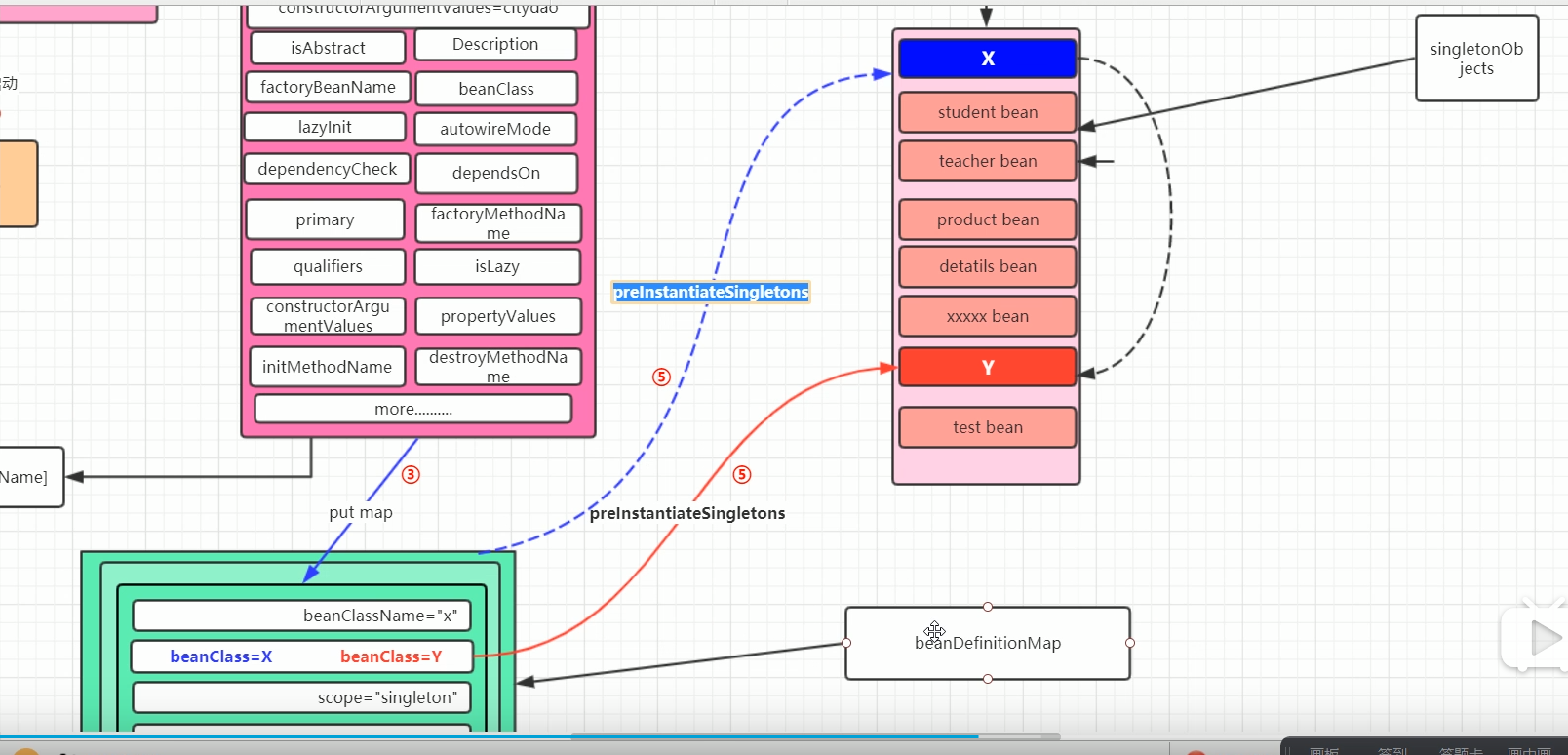
#### ★Spring框架【DI、AOP】

##### 1）Spring底层实现

###### A）Spring Bean实例化过程

BeanDefinition：SpringBean的建模对象，描述SpringBean对象，JVM中的Class类，bd----bean

AbstractBeanDefinition 

代码：**【最终实例化bean是通过反射得到的】**

初始化：AnnotationConfigApplicationContext

第一步：

// 1. 首先初始化Spring的7个内置Bean后置处理器，并放到 DefaultListableBeanFactory 类型的对象 beanFactory 中

// 2. 创建Spring的注解解析器 Component

this();

this.beanFactory = new DefaultListableBeanFactory();//父类的，初始化bean工厂

//解析@Configuration

this.reader = new AnnotatedBeanDefinitionReader(this);

//一般是扩展Spring用

this.scanner = new ClassPathBeanDefinitionScanner(this);

第二步：

register(annotatedClasses);

在执行完register(annotatedClasses);方法后，我们传入的配置类信息就会保存到Spring容器的beanFactory中。至此，Spring容器中包含了7个类信息。（6个内置后置处理器类，一个配置类）。

第三步：

refresh();

Spring bean的扫描是在这里面完成的（并且还完成了beanFactoryPostProcesser后置处理器的工作）

invokeBeanFactoryPostProcessors(beanFactory);

//这里仅仅是完成了扫描

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

调用程序员自己实现的beanFactoryPostProcesser后置处理器，并完成里面的操作

invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

###### Spring AOP的源码解析【AOP：解决横切性问题】

**AOP介绍：**

**OOP的代码逻辑是自上而下的，在自上而下的过程中会产生切面性问题，AOP的编程思想是把这些横切性的问题和业务逻辑进行分离，达到解耦的目的**

**AOP应用场景：**

场景一：记录日志

场景二：监控方法运行时间 （监控性能）

场景三：权限控制

场景四：缓存优化（第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五：事务管理 （调用方法前开启事务， 调用方法后提交关闭事务 ）

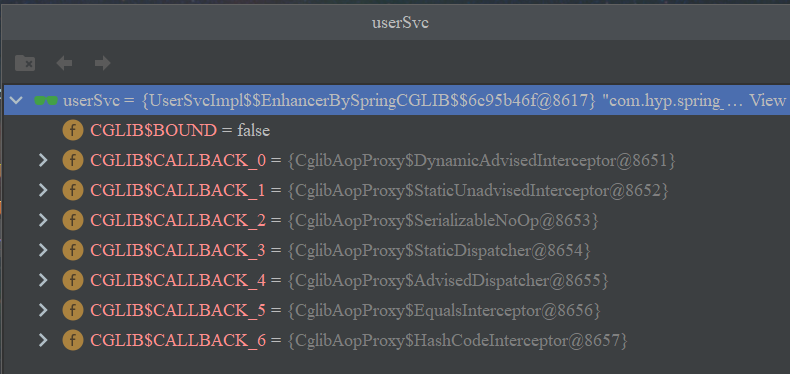
问题1：SpringAOP的本质是什么？SpringAOP的底层是什么？

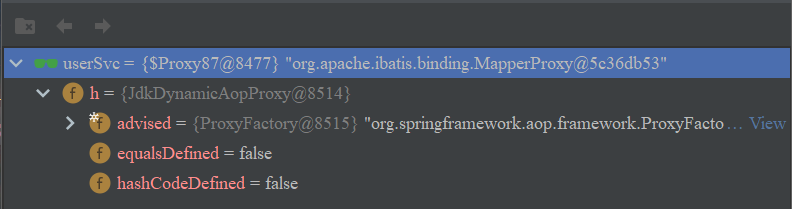
**底层实现：【区别：代理对象是否为接口，实现方式有两种】**

**Java动态代理或者CGLIB**

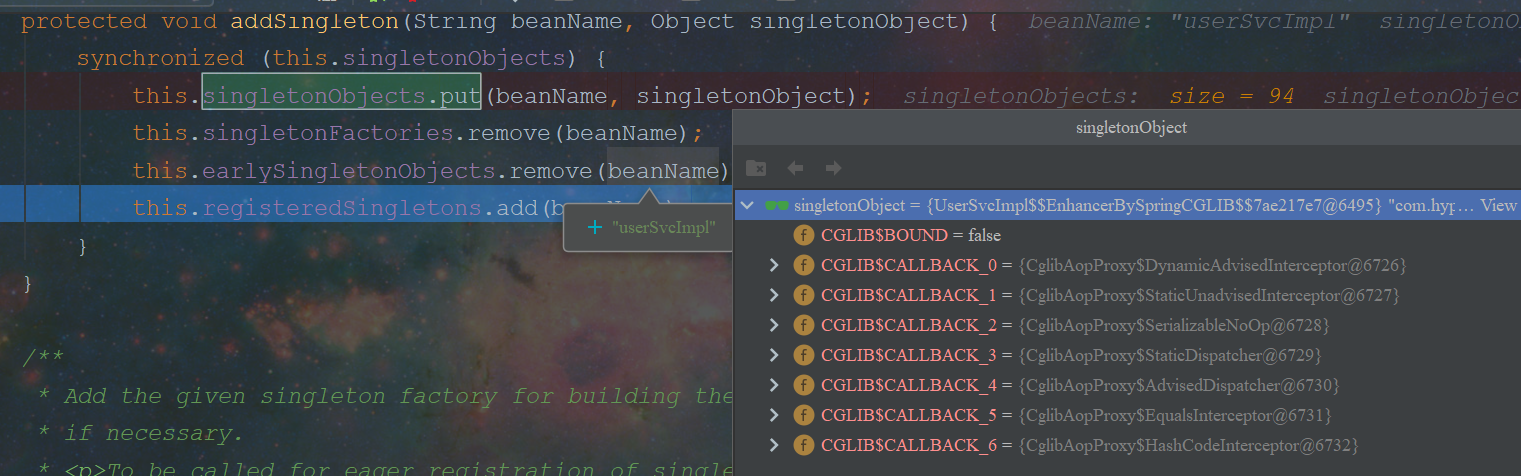
动态代理的底层是通过字节码实现的

**代理对象不是接口：用CGLIB，代理对象是接口：用Java动态代理**





**在put时，已经是CGLIB动态代理**



**这个就是IOC，一个map，存放beanName和实例对象**

**private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);**

**★★★★★**

**补充：Spring框架看源代码技巧：**

1. **条件断点，断点右键，输入condition**
2. **带着问题看源代码**
3. **Map get之前一定有put**
4. **找出代理对象，返回对象**

AOP源码在此处决定是用动态代理还是CGLIB？

**org.springframework.aop.framework.DefaultAopProxyFactory#createAopProxy**

**return (AopProxy)(!targetClass.isInterface() && !Proxy.isProxyClass(targetClass) ?**

**new ObjenesisCglibAopProxy(config) : new JdkDynamicAopProxy(config));**

问题2：Java动态代理CGLIB原理是什么？

CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理。

问题3：为什么Java动态代理必须是接口？

Java类是单继承多实现

##### 2）Spring如何解决循环依赖

##### 3）Tomcat容器启动时Spring的启动流程

##### 4）SpringAOP原理

##### 5）Spring常用注解大全

###### 1、声明bean的注解

★★@Component 组件，没有明确的角色

★★@Service 在业务逻辑层使用（service层）

★★@Repository 在数据访问层使用（dao层）、@Mapper注解

@ Repository和@Mapper的区别：

@Repository需要在Spring中配置扫描地址，然后生成Dao层的Bean才能被注入到Service层中，@Mapper不需要配置扫描地址，通过xml里面的namespace里面的接口地址，生成了Bean后注入到Service层中。也就是@Repository多了一个配置扫描地址的步骤

★★@Controller 在展现层使用，控制器的声明（C）

###### 2、注入bean的注解

★★@Autowired：由Spring提供

@Inject：由JSR-330提供

★★@Resource：由JSR-250提供

（1）：相同点

@Resource的作用相当于@Autowired，均可标注在字段或属性的setter方法上。

（2）：不同点

a：提供方 @Autowired是Spring的注解，@Resource是javax.annotation注解，而是来自于JSR-250，J2EE提供，需要JDK1.6及以上。

b ：注入方式 @Autowired只按照Type 注入；@Resource默认按Name自动注入，也提供按照Type 注入；

c：属性

@Autowired注解可用于为类的属性、构造器、方法进行注值。默认情况下，其依赖的对象必须存在（bean可用），如果需要改变这种默认方式，可以设置其required属性为false。

还有一个比较重要的点就是，@Autowired注解默认按照类型装配，如果容器中包含多个同一类型的Bean，那么启动容器时会报找不到指定类型bean的异常，解决办法是结合\*\*@Qualifier\*\*注解进行限定，指定注入的bean名称。

@Resource有两个中重要的属性：name和type。name属性指定byName，如果没有指定name属性，当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象，当注解标注在属性的setter方法上，即默认取属性名作为bean名称寻找依赖对象。

需要注意的是，@Resource如果没有指定name属性，并且按照默认的名称仍然找不到依赖对象时， @Resource注解会回退到按类型装配。但一旦指定了name属性，就只能按名称装配了。

d：@Resource注解的使用性更为灵活，可指定名称，也可以指定类型；@Autowired注解进行装配容易抛出异常，特别是装配的bean类型有多个的时候，而解决的办法是需要在增加@Qualifier进行限定。

###### 3、java配置类相关注解

★★@Configuration 声明当前类为配置类，相当于xml形式的Spring配置（类上）

★★@Bean 注解在方法上，声明当前方法的返回值为一个bean，替代xml中的方式（方法上）

@Configuration 声明当前类为配置类，其中内部组合了@Component注解，表明这个类是一个bean（类上）

★★@ComponentScan 用于对Component进行扫描，相当于xml中的（类上）

@WishlyConfiguration 为@Configuration与@ComponentScan的组合注解，可以替代这两个注解

@Configuration和@Component的区别：

使用Configuration时在driver和spring容器之中的是同一个对象，而使用Component时是不同的对象。 虽然Component注解也会当做配置类，但是并不会为其生成CGLIB代理Class，所以在生成Driver对象时和生成Car对象时调用car()方法执行了两次new操作，所以是不同的对象。当时Configuration注解时，生成当前对象的子类Class，并对方法拦截，第二次调用car()方法时直接从BeanFactory之中获取对象，所以得到的是同一个对象。

###### 4、切面（AOP）相关注解

Spring支持AspectJ的注解式切面编程。

@Aspect 声明一个切面（类上）

使用@After、@Before、@Around定义建言（advice），可直接将拦截规则（切点）作为参数。

@After 在方法执行之后执行（方法上）

@Before 在方法执行之前执行（方法上）

@Around 在方法执行之前与之后执行（方法上）

@PointCut 声明切点

在java配置类中使用@EnableAspectJAutoProxy注解开启Spring对AspectJ代理的支持（类上）

###### 5、@Bean的属性支持

★★@Scope 设置Spring容器如何新建Bean实例（方法上，得有@Bean）

其设置类型包括：

Singleton （单例,一个Spring容器中只有一个bean实例，默认模式）,

Protetype （每次调用新建一个bean）,

Request （web项目中，给每个http request新建一个bean实例）,

Session （web项目中，给每个http session新建一个bean实例）,

GlobalSession（给每一个 global http session新建一个Bean实例）

@StepScope 在Spring Batch中还有涉及

@PostConstruct 由JSR-250提供，在构造函数执行完之后执行，等价于xml配置文件中bean的initMethod

@PreDestory 由JSR-250提供，在Bean销毁之前执行，等价于xml配置文件中bean的destroyMethod

###### 6、@Value注解

@Value 为属性注入值（属性上）支持如下方式的注入：》注入普通字符

###### 7、环境切换

@Profile 通过设定Environment的ActiveProfiles来设定当前context需要使用的配置环境。（类或方法上）

@Conditional Spring4中可以使用此注解定义条件话的bean，通过实现Condition接口，并重写matches方法，从而决定该bean是否被实例化。（方法上）

###### 8、异步相关

@EnableAsync 配置类中，通过此注解开启对异步任务的支持，叙事性AsyncConfigurer接口（类上）

@Async 在实际执行的bean方法使用该注解来申明其是一个异步任务（方法上或类上所有的方法都将异步，需要@EnableAsync开启异步任务）

###### 9、定时任务相关

★★@EnableScheduling 在配置类上使用，开启计划任务的支持（类上）

★★@Scheduled 来申明这是一个任务，包括cron,fixDelay,fixRate等类型（方法上，需先开启计划任务的支持）

###### 10、@Enable\*注解说明

这些注解主要用来开启对xxx的支持。

@EnableAspectJAutoProxy 开启对AspectJ自动代理的支持

@EnableAsync 开启异步方法的支持

@EnableScheduling 开启计划任务的支持

★★@EnableWebMvc 开启Web MVC的配置支持

@EnableConfigurationProperties 开启对@ConfigurationProperties注解配置Bean的支持

@EnableJpaRepositories 开启对SpringData JPA Repository的支持

★★@EnableTransactionManagement 开启注解式事务的支持

★★@EnableCaching 开启注解式的缓存支持

###### 11、测试相关注解

@RunWith 运行器，Spring中通常用于对JUnit的支持

在这里插入图片描述

@ContextConfiguration 用来加载配置ApplicationContext，其中classes属性用来加载配置类

###### 12、SpringMVC部分

★★@EnableWebMvc 在配置类中开启WebMVC的配置支持，如一些ViewResolver或者MessageConverter等，若无此句，重写WebMvcConfigurerAdapter方法（用于对SpringMVC的配置）

@Controller 声明该类为SpringMVC中的Controller

★★@RequestMapping 用于映射Web请求，包括访问路径和参数（类或方法上）

★★@ResponseBody 支持将返回值放在response内，而不是一个页面，通常用户返回json数据（返回值旁或方法上）

★★@RequestBody 允许request的参数在request体中，而不是在直接连接在地址后面。（放在参数前）

★★@PathVariable 用于接收路径参数，比如@RequestMapping(“/hello/{name}”)申明的路径，将注解放在参数中前，即可获取该值，通常作为Restful的接口实现方法。

★★@RestController 该注解为一个组合注解，相当于@Controller和@ResponseBody的组合，注解在类上，意味着，该Controller的所有方法都默认加上了@ResponseBody。

@ControllerAdvice 通过该注解，我们可以将对于控制器的全局配置放置在同一个位置，注解了@Controller的类的方法可使用@ExceptionHandler、@InitBinder、@ModelAttribute注解到方法上，这对所有注解了 @RequestMapping的控制器内的方法有效。

★★@ExceptionHandler 用于全局处理控制器里的异常

@InitBinder 用来设置WebDataBinder，WebDataBinder用来自动绑定前台请求参数到Model中。

@ModelAttribute 本来的作用是绑定键值对到Model里，在@ControllerAdvice中是让全局的@RequestMapping都能获得在此处设置的键值对。

#### ★Spring MVC框架

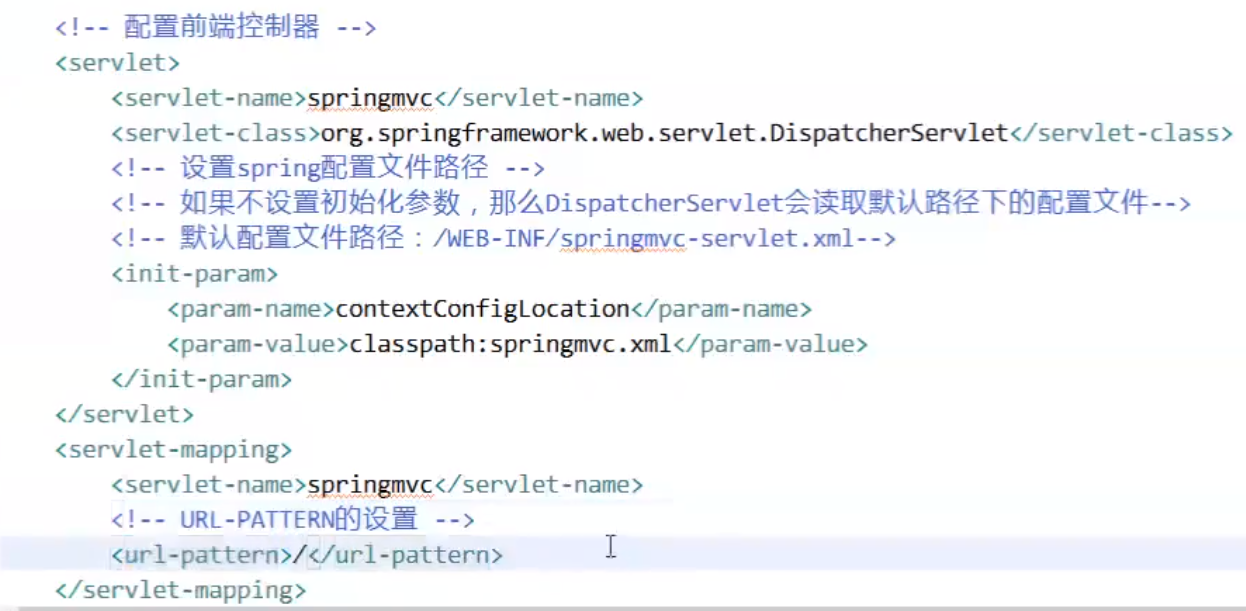
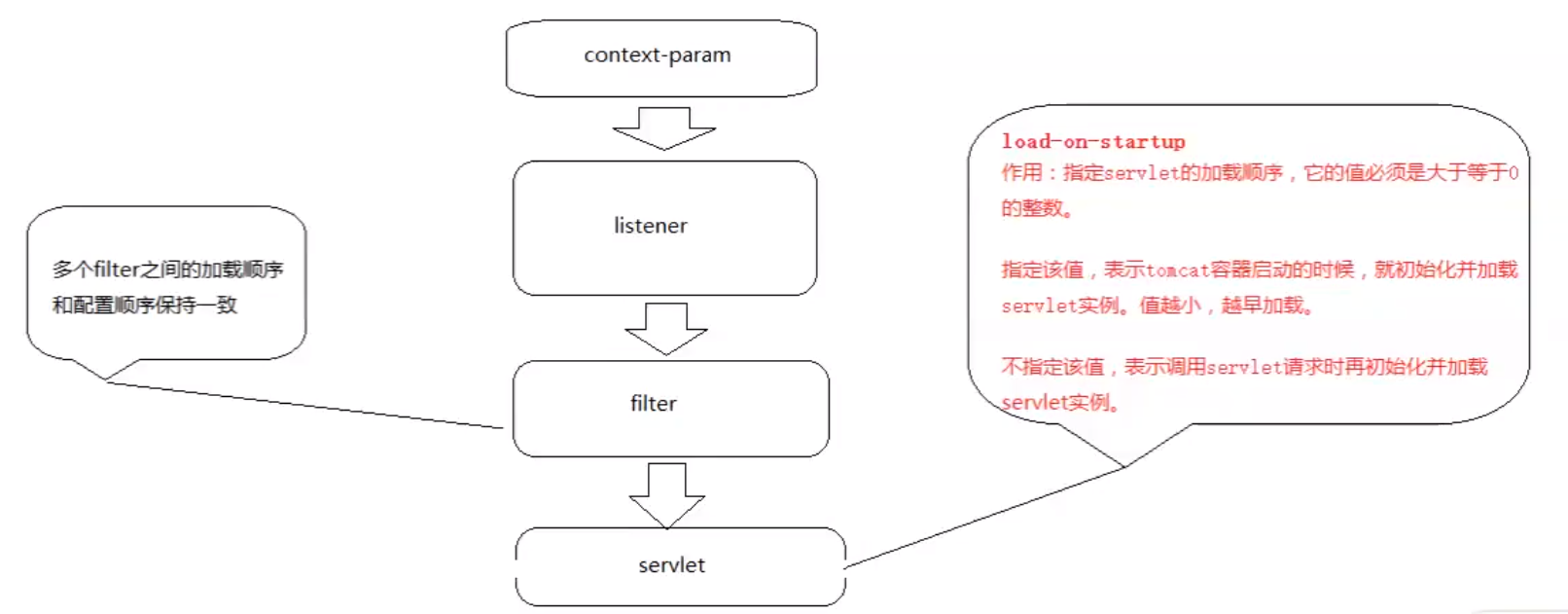
##### 1、原理

SpringMVC深入解析：

SpringMVC是基于MVC模式设计的，Model（模型）View（视图）Controller（控制器）

SpringMVC作用：处理Http请求

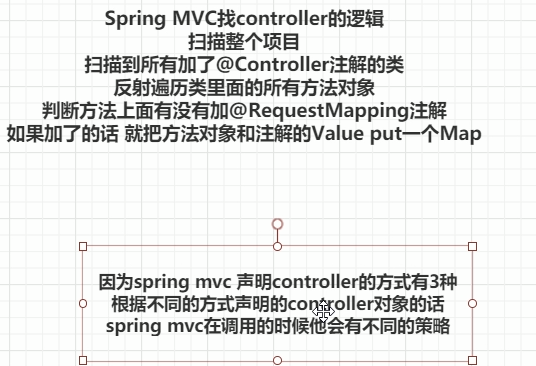
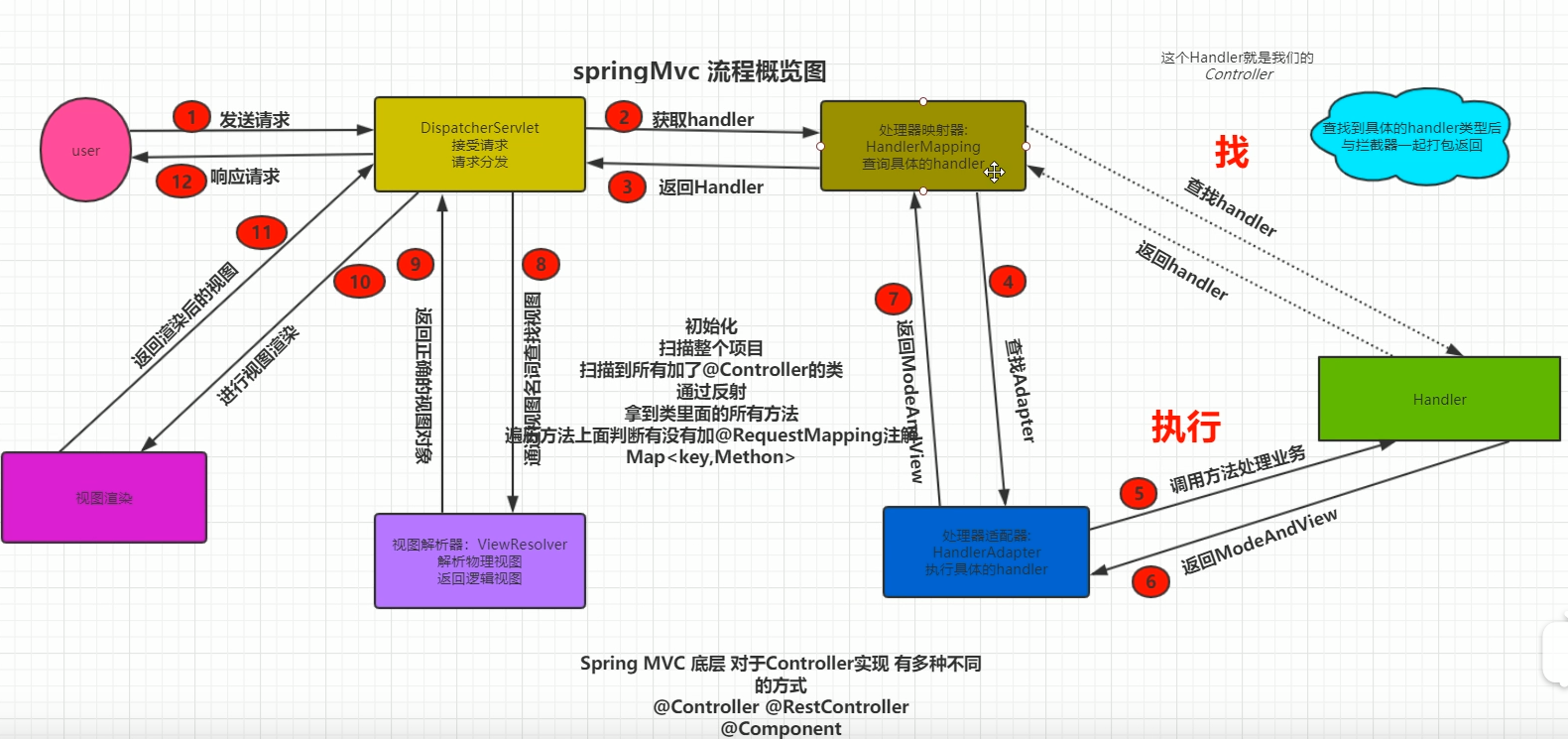
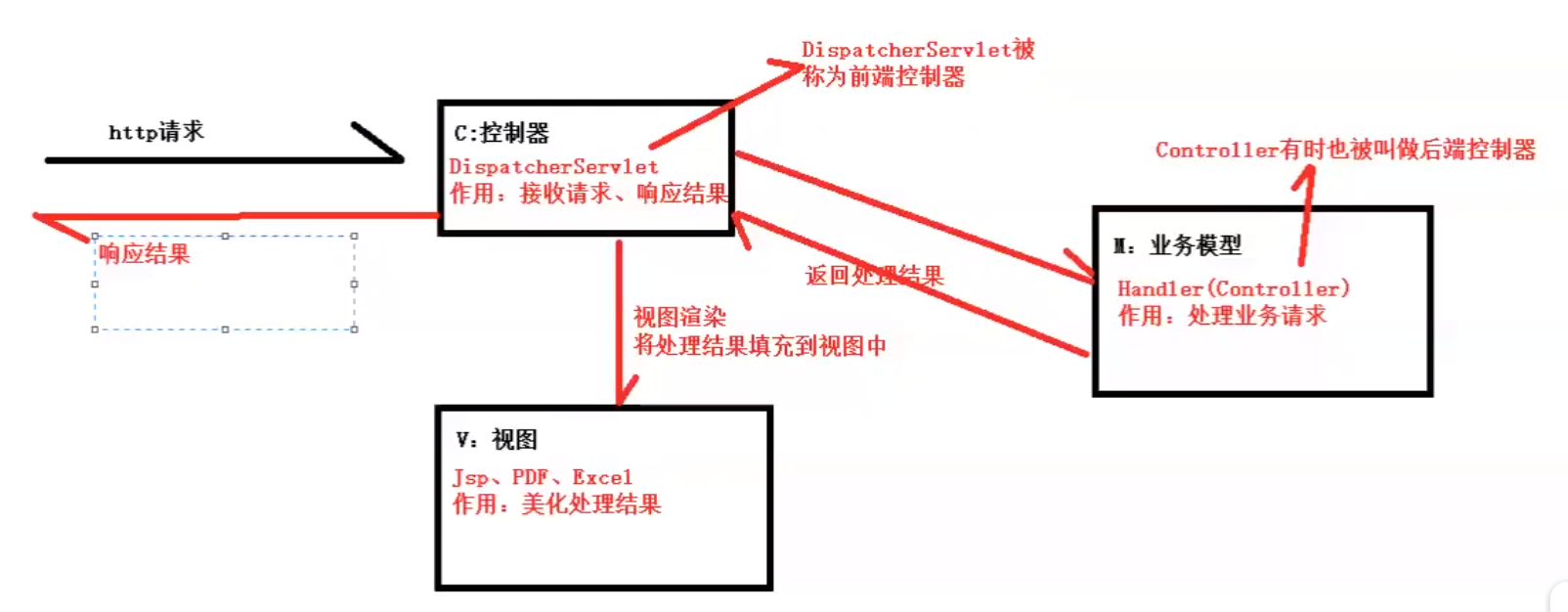
SpringMVC依赖包：

##### 2、源码

##### 3、执行流程

SpringMVC处理流程：



#### 1、springIOC的理解

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的类的对象交给容器控制，而不是传统的在你的对象内部直接控制

深入理解：

●谁控制谁，控制什么：传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建；谁控制谁？当然是**IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。**

●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而**反转则是由容器来帮忙创建及注入依赖对象**；**为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转**；哪些方面反转了？依赖对象的获取被反转了。

DI：

DI—Dependency Injection，即“依赖注入

是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

●谁依赖于谁：当然是某个容器管理对象依赖于IoC容器；“被注入对象的对象”依赖于“依赖对象”；

●为什么需要依赖：容器管理对象需要IoC容器来提供对象需要的外部资源；

●谁注入谁：很明显是IoC容器注入某个对象，也就是注入“依赖对象”；

●注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

#### 2、springIOC的java底层原理；

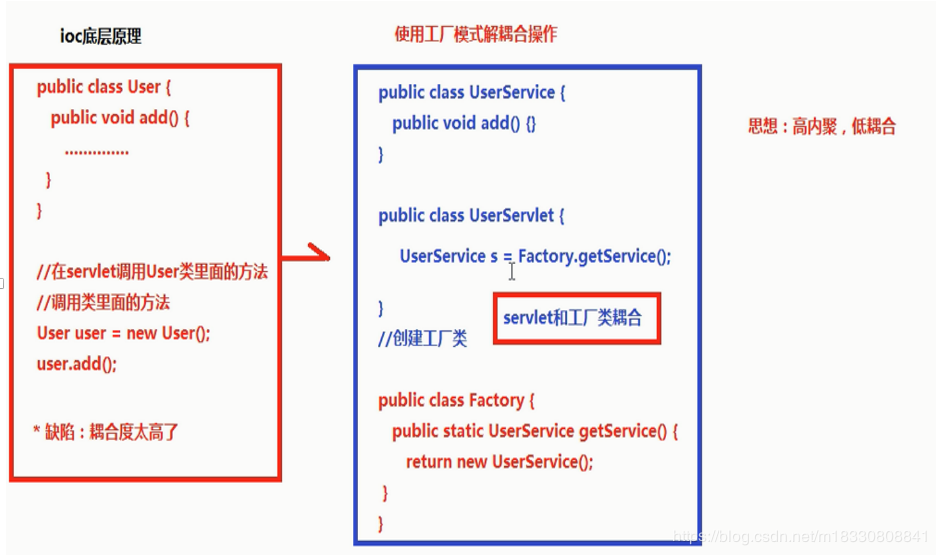
ioc底层原理使用技术

（1）xml配置文件

（2）dom4j解决xml

（3）工厂设计模式

（4）反射





上述的意思就是，一个类，在类里面有方法（不是静态的方法），调用类里面的方法，创建类的对象，使用对象的调用方法，创建类对象的过程，需要new出来对象。

spring的ioc就是把对象的创建不是通过new方式实现，而是交给spring配置创建类对象。

spring的ioc操作，就是通过spring配置创建类对象。

有两种方式：ioc的配置文件方式，和ioc的注解方式。配置个人认为ioc配置的xml文件相对于注解比较麻烦。

#### 3、如何控制事务；

Spring 的事务，可以说是 Spring AOP 的一种实现。

AOP面向切面编程，即在不修改源代码的情况下，对原有功能进行扩展，通过代理类来对具体类进行操作。

spring是一个容器，通过spring这个容器来对对象进行管理，根据配置文件来实现spring对对象的管理。

在开发中需要操作数据库，进行增、删、改操作的过程中属于一次操作，如果在一个业务中需要更新多张表，那么任意一张表的更新失败，整个业务的更新就是失败，这时那些更新成功的表必须回滚，否则业务会出错，这时就要用到事务，即这个业务的操作属于一个事务，事务具有原子性、隔离性、一致性、持续性。这时便用到了事务，事务控制的目的是保证一组操作要么全部成功，要么全部失败。spring提供了对事务的支持，在spring中主要有两种方式使用事务，**一、编程式事务控制；二、声明式事务控制**。

spring的事务声明有两种方式，**编程式和声明式**。spring主要是通过“声明式事务”的方式对事务进行管理，即在配置文件中进行声明，**通过AOP将事务切面切入程序，最大的好处是大大减少了代码量**

#### 4、事务的隔离级别和传播机制

事务四大特性:

　　\* 原子性(Atomicity)  :强调的事务的不可分割.

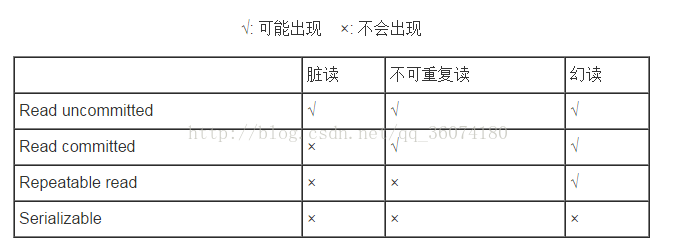
　　\* 一致性(Consistency)    :强调的事务的执行前后,数据库的的完整性保持一致.

　　\* 隔离性(Isolation)  :强调的事务的并发的访问,一个事务的执行,不应该受到另一个事务的打扰.

　　\* 持久性(Durability) :强调的事务结束之后,数据就永久的保存的数据库中.

\* 安全性:read uncommitted   <  read committed  < repeatable read  < serializable

\* 效率:read uncommitted  > read committed  > repeatable read > serializable



数据库事务的隔离级别有4种，由低到高分别为Read uncommitted 、Read committed 、Repeatable read 、Serializable 。而且，在事务的并发操作中可能会出现脏读，不可重复读，幻读。

**１. ISOLATION\_READ\_UNCOMMITTED：这是事务最低的隔离级别，A事务可以看到B事务未提交的数据**

      这种隔离级别会产生脏读，不可重复读和幻像读。

事例：代静给郭华发工资，一个月一块，但是代静按错了，发成了两块，该钱已经打到了郭华的账户，但是事务还没有提交，郭华发现了自己的账户多了一块，但是代静及时发现了，修改了又提交了事务（郭华可以看到代静未提交的事务，这就出现了脏读问题）

**２. ISOLATION\_READ\_COMMITTED：A事务可以读取B事务修改后提交的数据，但是不能读取B事务未提交的数据**

事例：郭华去吃饭，买单时（事务已开启），代静把郭华的钱转走了，这时郭华发现卡里没钱了（饭店老板的事务要等到代静的转钱事务操作提交后，他才能读取数据）

一个事务范围内两个相同的查询却返回了不同数据，这就是不可重复读

**３. ISOLATION\_REPEATABLE\_READ：这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。**

      它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

事例：郭华去吃饭，买单时（事务已开启），代静不能进行转钱操作，避免了不可重复读的问题

（但是可能出现幻读问题，幻读问题对应的是插入INSERT操作，而不是UPDATE操作）

**４. ISOLATION\_SERIALIZABLE：这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。**

      除了防止脏读，不可重复读外，还避免了幻像读。

大多数数据库默认的事务隔离级别是Read committed，比如Sql Server , Oracle。MySQL的默认隔离级别是Repeatable read。

**脏读：A事务处理过程中读取了B事务未提交的数据，这就是脏读**

**不可重复读：不可重复读取是指同一个事务在整个事务过程中对同一笔数据进行读取，每次读取结果都不同**

**幻读：是指同样一笔查询在整个事务过程中多次执行后，查询所得的结果集是不一样的。幻读针对的是多笔记录**

#### 5、Spring BeanFactory与FactoryBean的区别；

#### 6、springAOP的应用场景；

**场景一： 记录日志**

**场景二： 监控方法运行时间 （监控性能）**

**场景三： 权限控制**

**场景四： 缓存优化** （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

**场景五： 事务管理** （调用方法前开启事务， 调用方法后提交关闭事务 ）