**1，Spring 框架中都用到了哪些设计模式**

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

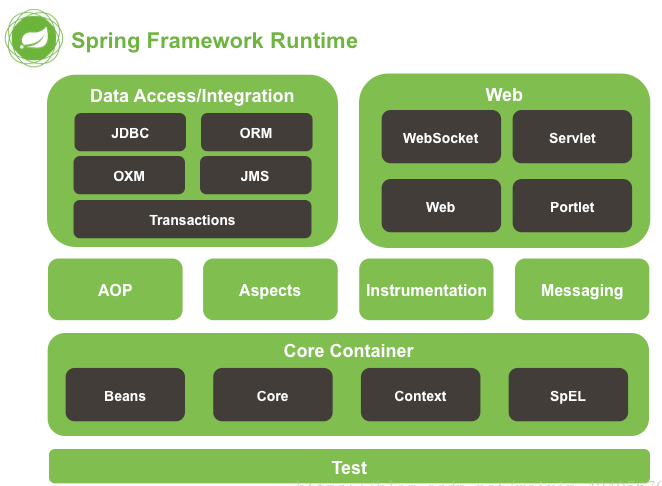
（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate， RedisTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

2，请简单介绍一下spring

Spring是一个轻量级（不依赖或者很少依赖其他接口或者框架，不依赖容器，配置简单，方便通用等等。。。）框架，可以一站式构建你的企业级应用。Spring框架是一个开源的框架，目的是为了简化企业级应用开发，是简单的JavaBean对象也拥有以前只有EJB才有的部分功能，spirng的核心是控制反转（IOC）和面向切面（AOP）

Spring的模块大概分为8个。分别是：



1、Core Container（Spring的核心）【重要】

2、AOP，Aspects（面向切面编程）【重要】

3、Messaging（消息发送的支持）

4、Data Access/Integration（数据访问和集成）

5、Web（主要是SpringWeb内容，包括MVC）【重要】

6、Test（Spring测试支持，包含JUint等测试单元的支持）

 7、Instrumentation（设备支持，比如Tomcat的支持）

3，请问什么是IOC和DI？并且简要说明一下DI是如何实现的

IoC叫控制反转，是Inversion of Control的缩写，DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。IoC体现了好莱坞原则 - "Don’t call me, we will call you"。依赖注入的基本原则是应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由容器负责，查找资源的逻辑应该从应用组件的代码中抽取出来，交给容器来完成。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

或者：

IoC叫控制反转，是Inversion of Control的缩写，DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。传统的java开发模式中，当需要一个对象时，我们会自己使用new或者getInstance等直接或者间接调用构造方法创建一个对象，而在Spring开发模式中，Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。

4，请谈一谈Spring中自动装配的方式有哪些

- no：不进行自动装配，手动设置Bean的依赖关系。

- byName：根据Bean的名字进行自动装配。

- byType：根据Bean的类型进行自动装配。

- default：可以根据名称或者是类型进行自动装配。

- constructor：类似于byType，不过是应用于构造器的参数，如果正好有一个Bean与构造器的参数类型相同则可以自动装配，否则会导致错误。

- autodetect：自动选择，如果有默认的构造器，则通过constructor的方式进行自动装配，否则使用byType的方式进行自动装配。（用于spring2.5 ，spring3.0之后测试不通过，估计是废弃了）

5，请问Spring中Bean的作用域有哪些

在Spring的早期版本中，仅有两个作用域：singleton和prototype，前者表示Bean以单例的方式存在；后者表示每次从容器中调用Bean时，都会返回一个新的实例，prototype通常翻译为原型。

设计模式中的创建型模式中也有一个原型模式，原型模式也是一个常用的模式，例如做一个室内设计软件，所有的素材都在工具箱中，而每次从工具箱中取出的都是素材对象的一个原型，可以通过对象克隆来实现原型模式。Spring 2.x中针对WebApplicationContext新增了3个作用域，分别是：request（每次HTTP请求都会创建一个新的Bean）、session（同一个HttpSession共享同一个Bean，不同的HttpSession使用不同的Bean）和globalSession（同一个全局Session共享一个Bean）。

单例模式和原型模式都是重要的设计模式。一般情况下，无状态或状态不可变的类适合使用单例模式。在传统开发中，由于DAO持有Connection这个非线程安全对象因而没有使用单例模式；但在Spring环境下，所有DAO类对可以采用单例模式，因为Spring利用AOP和Java API中的ThreadLocal对非线程安全的对象进行了特殊处理。

或者：

singleton：这种 bean 范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个 bean 的实例，单例的模式由 bean factory 自身来维护。

prototype：原形范围与单例范围相反，为每一个 bean 请求提供一个实例。

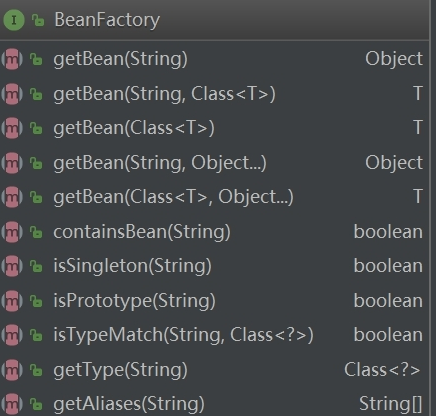
request：在请求 bean 范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean 会失 效并被垃圾回收器回收。

Session：与请求范围类似，确保每个 session 中有一个 bean 的实例，在 session 过期后，bean 会随 之失效。

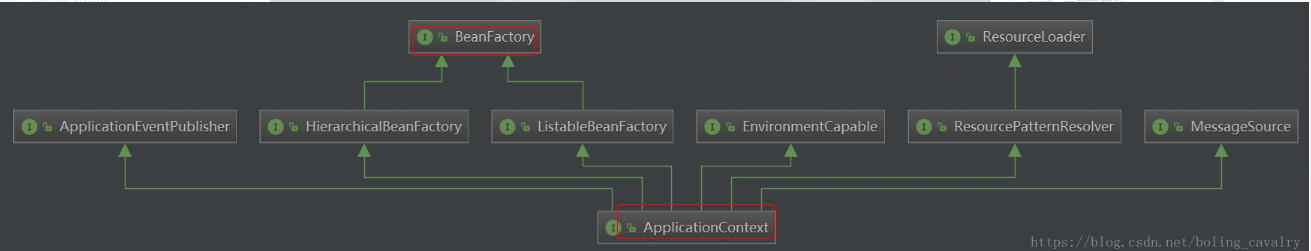
global-session：global-session 和 Portlet 应用相关。当你的应用部署在 Portlet 容器中工作时， 它包含很多 portlet。如果你想要声明让所有的 portlet 共用全局的存储变量的话，那么这全局变量需要存储在 global-session 中

6，请说明一下Spring中BeanFactory和ApplicationContext的区别是什么

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。



ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：



①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（2）两者装载bean的区别

BeanFactory：

BeanFactory在启动的时候不会去实例化Bean，只有从容器中拿Bean的时候才会去实例化；

ApplicationContext：

ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化；

没有特殊要求的情况下，应该使用ApplicationContext完成。因为BeanFactory能完成的事情，ApplicationContext都能完成，并且提供了更多接近现在开发的功能。

7，请问在以前的学习中有使用过Spring里面的注解吗？如果有请谈一下autowired 和resource区别是什么？

@ResponseBody：表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，用于构建RESTful的api。在使用@RequestMapping后，返回值通常解析为跳转路径，加上 @Responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。比如异步获取json数据，加上@Responsebody后，会直接返回json数据。

@Controller：用于定义控制器类，在spring项目中由控制器负责将用户发来的URL请求转发到对应的服务接口（service层），一般这个注解在类中，通常方法需要配合注解

@Service：一般用于修饰service层的组件

@Repository：使用@Repository注解可以确保DAO或者repositories提供异常转译，这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置，同时也不需要为它们提供XML配置项。

@RequestMapping 提供路由信息，负责URL到Controller中的具体函数的映射。

@RestController：

用于标注控制层组件(如struts中的action)，@ResponseBody和@Controller的合集。它继承自@Controller注解。spring4.0之前的版本，Spring MVC的组件都使用@Controller来标识当前类是一个控制器servlet。使用这个特性，我们可以开发REST服务的时候不需要使用@Controller而专门的@RestController。当你实现一个RESTful web services的时候，response将一直通过response body发送。为了简化开发，Spring 4.0提供了一个专门版本的controller。

@Bean：用@Bean标注方法等价于XML中配置的bean。

@Value：注入Spring boot application.properties配置的属性的值。

（1）@Autowired

@Autowired为Spring提供的注解，需要导入包org.springframework.beans.factory.annotation.Autowired;只按照byType注入。

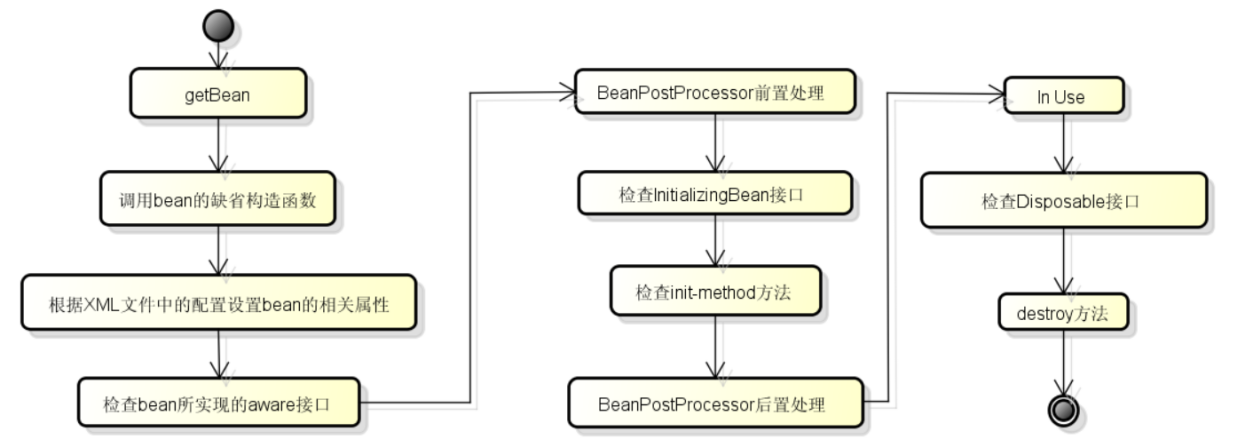
@Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果我们想使用按照名称（byName）来装配，可以结合@Qualifier注解一起使用。

（2）@Resource

@Resource默认按照ByName自动注入，由J2EE提供，需要导入包javax.annotation.Resource。@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用byName自动注入策略。

8，请介绍一下bean的生命周期

参考项目（spring\_interview\_question下com.aaa.spring.lifecycle包）



1.Spring容器 从XML 文件中读取Bean的定义，并实例化Bean。

2.Spring根据Bean的定义填充所有的属性。

3.如果Bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。

4.如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。

5.如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。

6.如果bean实现IntializingBean了，调用它的afterPropertiesSet方法，如果bean声明了初始化方法，调用此初始化方法。

7.如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。

8.如果bean实现了 DisposableBean，它将调用destroy()方法。

9，请简要说明一下AOP是什么？

AOP它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事务。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

你如何理解AOP中的连接点（Joinpoint）、切点（Pointcut）、增强（通知，Advice）、引介（Introduction）、织入（Weaving）、切面（Aspect）这些概念？

**切面**（Aspect）：切面编程的完整定义模块，比如日志切面，包含了何时、对谁、如何等等所有的内容。

**连接点**（Join point）：能够植入切面的部分，比如方法前后，抛出异常时都可以是连接点，spring只支持方法连接点

**增强（通知**Advice）：要对切面添加的功能代码，比如 安全，事物，日志等功能代码。类型：before/after/around/after-returning/after-throwing

**切入点**（Pointcut）：针对哪些连接点植入通知，也就是指定具体的拦截地点。可以通过spEL指定连接点

**引介**（引入Introduction）：对目标类添加新方法及属性

**目标对象**（Target object）：被切面切的对象。真正的业务逻辑，完全不知道存在切面

**代理**（proxy）：实现AOP的一种原理

**织入**（Weaving）：把切面应用到目标对象来创建新的代理对象的过程。有3种方式，spring采用的是运行时。

10，请问AOP的原理是什么

AOP（Aspect Oriented  Programming），指面向方面（切面）编程，作为面向对象的一种补充，用于处理系统中分布于各个模块的横切关注点，比如事务管理、日志、缓存等等。AOP实现的关键在于AOP框架自动创建的AOP代理，AOP代理主要分为静态代理和动态代理，静态代理的代表为AspectJ；而动态代理则以Spring AOP为代表。通常使用AspectJ的编译时增强实现AOP，AspectJ是静态代理的增强，所谓的静态代理就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强。

静态代理和动态代理：

静态代理

Java中的静态代理要求代理类(ProxySubject)和委托类(RealSubject)都实现同一个接口(Subject)。静态代理中代理类在编译期就已经确定，而动态代理则是JVM运行时动态生成，静态代理的效率相对动态代理来说相对高一些，但是静态代理代码冗余大，一旦需要修改接口，代理类和委托类都要发生变化

动态代理

Java中的动态代理依靠反射来实现，代理类和委托类不需要实现同一个接口。委托类需要实现接口，否则无法创建动态代理。代理类在JVM运行时动态生成，而不是编译期就能确定。

Java动态代理主要涉及到两个类：java.lang.reflect.Proxy和java.lang.reflect.InvocationHandler。代理类需要实现InvocationHandler接口或者创建匿名内部类，而Proxy用于创建动态代理。动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法（java.lang.reflect.InvocationHandler.invoke()）中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。而且动态代理的应用使我们的类职责更加单一，复用性更强

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理（参考项目：spring\_interview\_question的包com.aaa.proxy）。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

请问aop的应用场景有哪些

Authentication 权限 ，Caching 缓存 ，Context passing 内容传递 ，Error handling 错误处理 ，Lazy loading 懒加载 ，Debugging 调试 ，logging, tracing, profiling and monitoring 记录跟踪　优化　校准，Performance optimization 性能优化 ，Persistence 持久化 ，Resource pooling 资源池 ，Synchronization 同步，Transactions 事务。

11，事务：什么是事务？

[事务ACID特性](http://www.cnblogs.com/wangchuanqi/p/5554708.html)

 所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。例如，银行转帐工作：从一个帐号扣款并使另一个帐号增款，这两个操作要么都执行，要么都不执行。

　 数据库事务必须具备ACID特性，ACID是Atomic（原子性）、Consistency（一致性）、Isolation（隔离性）和Durability（持久性）的英文缩写。

　 原子性：指整个数据库事务是不可分割的工作单位。只有使据库中所有的操作执行成功，才算整个事务成功；事务中任何一个SQL语句执行失败，那么已经执行成功的SQL语句也必须撤销，数据库状态应该退回到执行事务前的状态。

　 一致性：指数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。例如对银行转帐事务，不管事务成功还是失败，应该保证事务结束后ACCOUNTS表中Tom和Jack的存款总额为2000元。

　 隔离性：指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看到中间状态的数据。

　 持久性：指的是只要事务成功结束，它对数据库所做的更新就必须永久保存下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。

　　事务的（ACID）特性是由关系数据库管理系统（RDBMS，数据库系统）来实现的。数据库管理系统采用日志来保证事务的原子性、一致性和持久性。日志记录了事务对数据库所做的更新，如果某个事务在执行过程中发生错误，就可以根据日志，撤销事务对数据库已做的更新，使数据库退回到执行事务前的初始状态。

　　数据库管理系统采用锁机制来实现事务的隔离性。当多个事务同时更新数据库中相同的数据时，只允许持有锁的事务能更新该数据，其他事务必须等待，直到前一个事务释放了锁，其他事务才有机会更新该数据。

12，Spring事务管理

支持两种类型的事务管理：

**1.编程式事务管理：**这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

实际应用使用很少，通过TransactionTemplate手动管理事务

<https://blog.csdn.net/aimashi620/article/details/81663835>

**2.声明式事务管理：**这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**基于AOP方式的声明式事务管理**

<!-- 定义事务管理器（声明式的事务） -->

    <bean id="transactionManager"

        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

        <property name="dataSource" ref="dataSource" />

    </bean>

<!-- 配置Advice通知 -->

    <tx:advice id="txAdvice" transaction-manager="transactionManager">

        <tx:attributes>

            <tx:method name="\*" propagation="REQUIRED" />

        </tx:attributes>

    </tx:advice>

    <!-- 配置切点切面 -->

    <aop:config>

        <aop:pointcut id="interceptorPointCuts"

            expression="execution(\* com.aaa.xxx.service.\*.\*(..))" />

        <aop:advisor advice-ref="txAdvice"

            pointcut-ref="interceptorPointCuts" />

    </aop:config>

**基于注解方式的声明式事务管理**

<!-- 定义事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- 注解驱动，启动注解 -->

<tx:annotation-driven transaction-manager="taransactionManager"/>

在业务层类上或者业务方法添加注解@Transactional

.....

13 Spring事务的隔离级别和传播机制

**传播机制：**

指的就是当**一个事务B方法被另一个事务A方法调用时，这个事务B方法应该如何进行。**

在不同的业务类里面

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

操作。。。

b();

c();

操作。。。。

}

@Transactional(propagation=Propagation.REQUIRED)

public void b(){

。。。操作

}

@Transactional(propagation=Propagation.REQUIRED)

public void c(){

}

在@Transactional注解中，可以propagation属性用来配置事务传播

注解配置时如：@Transactional(propagation=Propagation.REQUIRED)

支持7种不同的传播机制：

**REQUIRED**（默认）：如果当前没在事务之中，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。

如果当前没在事务之中，就新建一个事务：

public void a(){

b();

c();

}

@Transactional(propagation=Propagation.REQUIRED)

public void b(){

异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void c(){

无异常

}

只有新闻表添加上去，说明b和c方法的异常对a方法，没有任何影响

public void a(){

b();

c();

}

@Transactional(propagation=Propagation.REQUIRED)

public void b(){

无异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void c(){

异常

}

新闻表和新闻附件1表都添加上了数据 b和c的事务相互不影响

public void a(){

b();

c();

出现异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void b(){

无异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void c(){

无异常

}

三个表都添加了数据，a的异常对b和c都没有影响

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

出现异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void b(){

无异常

}

@Transactional(propagation=Propagation.REQUIRED)

public void c(){

无异常

}

三个表都没有添加数据 b和c已经存在于a的事务中，所有a如果出错b,c也会回滚

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

public void b(){

无异常

}

public void c(){

异常

}

三个表都没有添加数据，c出现了异常,全部回滚了。

**SUPPORTS**：支持使用当前事务（a方法有没有事务），如果当前事务不存在，则不使用事务。

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.SUPPORTS)

public void b(){

异常

}

@Transactional(propagation=Propagation.**SUPPORTS**)

public void c(){

异常

}

新闻表添加成功，附件1添加一条，附件2没有添加（附件1出错，对附件2影响）

b和c为Propagation.SUPPORTS时，如果不在事务中，不使用事务

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.SUPPORTS)

public void b(){

异常

}

@Transactional(propagation=Propagation.**SUPPORTS**)

public void c(){

异常

}

三个表都没有数据，a存在事务，b,c支持事务

**MANDATORY**：中文翻译为强制，支持使用当前事务，如果当前事务不存在，则抛出Exception。

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.MANDATORY)

public void b(){

无异常

}

@Transactional(propagation=Propagation.MANDATORY)

public void c(){

异常

}

三个表都没有数据，a存在事务，b,c支持事务

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.MANDATORY)

public void b(){

无异常

}

@Transactional(propagation=Propagation.MANDATORY)

public void c(){

无异常

}

三个表都没有数据，a不存在事务，直接出异常 IllegalTransactionStateException: No existing transaction found for transaction marked with propagation 'mandatory'

**REQUIRES\_NEW**：创建一个新事务，如果当前事务存在，把当前事务挂起。

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void b(){

无异常

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void c(){

异常

}

发现新闻表没有添加数据，附件1添加成功，附件2添加失败 b自己新创建事务，和a,c

方法没有关系，c也自己新创建事务出错回滚，但是a调用了c,因为a上有事务所以也会回滚

**NOT\_SUPPORTED**：无事务执行，如果当前事务存在，把当前事务挂起。

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

public void b(){

无异常

}

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

public void c(){

异常

}

新闻表无数据，附件1有数据，附件2有一条数据。a方法执行失败，因为c有异常，b方法执行成功，c方法有一条执行成功，所有b和c都没有使用a的事务，把a事务挂起。

**NEVER**：无事务执行，如果当前有事务则抛出Exception。

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.NEVER)

public void b(){

无异常

}

@Transactional(propagation=Propagation.NEVER)

public void c(){

异常

}

三个表都没有数据 抛出 IllegalTransactionStateException: Existing transaction found for transaction marked with propagation 'never'

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.NEVER)

public void b(){

无异常

}

@Transactional(propagation=Propagation.NEVER)

public void c(){

异常

}

新闻表添加成功，附件1添加成功，附件2成功一条，无事务执行

**NESTED**：嵌套事务，如果当前事务存在，那么在嵌套的事务中执行。如果当前事务不存在，则表现跟REQUIRED一样。

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.**NESTED**)

public void b(){

无异常

}

@Transactional(propagation=Propagation.**NESTED**)

public void c(){

异常

}

新闻表成功，附件1成功，附件2有一条，和REQUIRED

@Transactional(propagation=Propagation.REQUIRED)

public void a(){

b();

c();

无异常

}

@Transactional(propagation=Propagation.**NESTED**)

public void b(){

无异常

}

@Transactional(propagation=Propagation.**NESTED**)

public void c(){

异常

}

三个表都没有数据，数据一起回滚, 和REQUIRES\_NEW的区别在于，事务回滚

**事务的并发问题：**

1、脏读：脏读表示一个事务能够读取另一个事务中还未提交的数据。事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

　　2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

　　3、幻读：就是指A事务对所有记录进行修改，尚未提交，此时B事务创建了一条新记录，A、B都提交。A查看所有数据，发现有一条数据没有被修改，因为这是B事务新增的，就想看到了幻象一样。如：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

　　小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

**隔离级别：**

注解配置时如：@Transactional(isolation = Isolation.READ\_UNCOMMITTED)



READ\_UNCOMMITTED：读未提交，即能够读取到没有被提交的数据，这是隔离性最低的一种隔离级别，无法解决脏读、不可重复读、幻读问题。

READ\_COMMITED：读已提交，即能够读到那些已经提交的数据，能够防止脏读，避免不了幻读和不可重复读。

REPEATABLE\_READ：可重复读，解决脏读和不可重复读的问题，但是其无法解决幻读。

SERLALIZABLE：串行化，提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行，故此性能很低

DEFAULT：默认隔离级别，每种数据库支持的事务隔离级别不一样，Oracle默认（读已提交） Mysql默认（可重复读）。

14，请说明一下Spring框架为企业级开发带来的好处有哪些？

- 非侵入式：支持基于POJO的编程模式，不强制性的要求实现Spring框架中的接口或继承Spring框架中的类。

- IoC容器：IoC容器帮助应用程序管理对象以及对象之间的依赖关系，对象之间的依赖关系如果发生了改变只需要修改配置文件而不是修改代码，因为代码的修改可能意味着项目的重新构建和完整的回归测试。有了IoC容器，程序员再也不需要自己编写工厂、单例，这一点特别符合Spring的精神"不要重复的发明轮子"。

- AOP（面向切面编程）：将所有的横切关注功能封装到切面（aspect）中，通过配置的方式将横切关注功能动态添加到目标代码上，进一步实现了业务逻辑和系统服务之间的分离。另一方面，有了AOP程序员可以省去很多自己写代理类的工作。

- MVC：Spring的MVC框架为Web表示层提供了更好的解决方案。

- 事务管理：Spring以宽广的胸怀接纳多种持久层技术，并且为其提供了声明式的事务管理，在不需要任何一行代码的情况下就能够完成事务管理。

- 其他：选择Spring框架的原因还远不止于此，Spring为Java企业级开发提供了一站式选择，你可以在需要的时候使用它的部分和全部，更重要的是，甚至可以在感觉不到Spring存在的情况下，在你的项目中使用Spring提供的各种优秀的功能。