1 解释IoC

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。如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●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。

2 IoC的优势

IoC不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

我们还是从USB的例子说起，使用USB外部设备比使用内置硬盘，到底带来什么好处？

第一、USB设备作为电脑主机的外部设备，在插入主机之前，与电脑主机没有任何的关系，只有被我们连接在一起之后，两者才发生联系，具有相关性。所以，无论两者中的任何一方出现什么的问题，都不会影响另一方的运行。这种特性体现在软件工程中，就是可维护性比较好，非常便于进行单元测试，便于调试程序和诊断故障。代码中的每一个Class都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。

第二、USB设备和电脑主机的之间无关性，还带来了另外一个好处，生产USB设备的厂商和生产电脑主机的厂商完全可以是互不相干的人，各干各事，他们之间唯一需要遵守的就是USB接口标准。这种特性体现在软件开发过程中，好处可是太大了。每个开发团队的成员都只需要关心实现自身的业务逻辑，完全不用去关心其它的人工作进展，因为你的任务跟别人没有任何关系，你的任务可以单独测试，你的任务也不用依赖于别人的组件，再也不用扯不清责任了。所以，在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。

第三、同一个USB外部设备可以插接到任何支持USB的设备，可以插接到电脑主机，也可以插接到DV机，USB外部设备可以被反复利用。在软件工程中，这种特性就是可复用性好，我们可以把具有普遍性的常用组件独立出来，反复利用到项目中的其它部分，或者是其它项目，当然这也是面向对象的基本特征。显然，IOC不仅更好地贯彻了这个原则，提高了模块的可复用性。符合接口标准的实现，都可以插接到支持此标准的模块中。

第四、同USB外部设备一样，模块具有热插拔特性。IOC生成对象的方式转为外置方式，也就是把对象生成放在配置文件里进行定义，这样，当我们更换一个实现子类将会变得很简单，只要修改配置文件就可以了，完全具有热插拨的特性。

3 Spring IOC核心源码

1. 初始化

初始化的过程主要就是读取XML资源，并解析，最终注册到Bean Factory中：

https://cloud.githubusercontent.com/assets/1736354/7897341/032179be-070b-11e5-9ecf-d7befc804e9d.png

**准备：**

**保存配置位置，并刷新**

在调用ClassPathXmlApplicationContext后，先会将配置位置信息保存到configLocations，供后面解析使用，之后，会调用AbstractApplicationContext的refresh方法进行刷新

**创建载入BeanFactory**

**创建XMLBeanDefinitionReader**

**读取：**

**创建处理每一个resource**

**处理XML每个元素**

**解析和注册bean**

**解析：**

**处理每个Bean的元素**

**处理属性的值**

**注册：**

注册过程中，最核心的一句就是：this.beanDefinitionMap.put(beanName, beanDefinition)，也就是说注册的实质就是以beanName为key，以beanDefinition为value，将其put到HashMap中。

1. 注入依赖

当完成初始化IOC容器后，如果bean没有设置lazy-init(延迟加载)属性，那么bean的实例就会在初始化IOC完成之后，及时地进行初始化。初始化时会先建立实例，然后根据配置利用反射对实例进行进一步操作，具体流程如下所示：

https://cloud.githubusercontent.com/assets/1736354/7929429/615570ea-0930-11e5-8097-ae982ef7709d.png

**创建bean的实例**

**注入bean的属性**

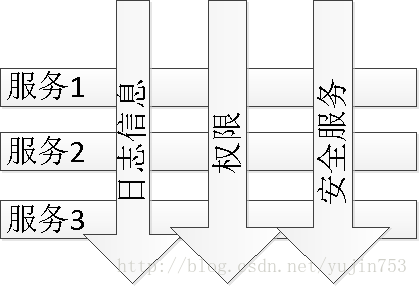
**4 Spring AOP**

AOP思想的实现一般都是基于 代理模式 ，在JAVA中一般采用JDK动态代理模式，但是我们都知道，JDK动态代理模式只能代理接口而不能代理类。因此，Spring AOP 会这样子来进行切换，因为Spring AOP 同时支持 CGLIB、ASPECTJ、JDK动态代理。

如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；

如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类——不过这个选择过程对开发者完全透明、开发者也无需关心。

|  |
| --- |
| 面向切面——Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。 |



5 [JDK动态代理、CGLIB动态代理讲解](http://www.cnblogs.com/puyangsky/p/6218925.html)

1）JDK动态代理

java.lang.reflect包里的InvocationHandler接口：

|  |
| --- |
| public interface InvocationHandler {  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable;  } |

我们对于被代理的类的操作都会由该接口中的invoke方法实现，其中的参数的含义分别是：

* + proxy：被代理的类的实例
  + method：调用被代理的类的方法
  + args：该方法需要的参数

java.lang.reflect包中的Proxy类的newProxyInstance方法：

|  |
| --- |
| public static Object newProxyInstance(ClassLoader loader,  Class<?>[] interfaces,  InvocationHandler h)  throws IllegalArgumentException |

其中的参数含义如下：

* + loader：被代理的类的类加载器
  + interfaces：被代理类的接口数组
  + invocationHandler：就是刚刚介绍的调用处理器类的对象实例

例子：

接口

|  |
| --- |
| public interface Fruit {  public void show();  } |

接口的实现类

|  |
| --- |
| public class Apple implements Fruit{  @Override  public void show() {  System.out.println("<<<<show method is invoked");  }  } |

代理类

|  |
| --- |
| public class DynamicAgent {  //实现InvocationHandler接口，并且可以初始化被代理类的对象  static class MyHandler implements InvocationHandler {  private Object proxy;  public MyHandler(Object proxy) {  this.proxy = proxy;  }    //自定义invoke方法  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.out.println(">>>>before invoking");  //真正调用方法的地方  Object ret = method.invoke(this.proxy, args);  System.out.println(">>>>after invoking");  return ret;  }  }  //返回一个被修改过的对象  public static Object agent(Class interfaceClazz, Object proxy) {  return Proxy.newProxyInstance(interfaceClazz.getClassLoader(), new Class[]{interfaceClazz},  new MyHandler(proxy));  }  } |

测试

|  |
| --- |
| public class ReflectTest {  public static void main(String[] args) throws InvocationTargetException, IllegalAccessException {  //注意一定要返回接口，不能返回实现类否则会报错  Fruit fruit = (Fruit) DynamicAgent.agent(Fruit.class, new Apple());  fruit.show();  }  } |

2）CGLIB动态代理

|  |
| --- |
| import net.sf.cglib.proxy.Enhancer;  import net.sf.cglib.proxy.MethodInterceptor;  import net.sf.cglib.proxy.MethodProxy;  import java.lang.reflect.Method;  public class CGlibAgent implements MethodInterceptor {  private Object proxy;  public Object getInstance(Object proxy) {  this.proxy = proxy;  Enhancer enhancer = new Enhancer();  enhancer.setSuperclass(this.proxy.getClass());  // 回调方法  enhancer.setCallback(this);  // 创建代理对象  return enhancer.create();  }  //回调方法  @Override  public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  System.out.println(">>>>before invoking");  //真正调用  Object ret = methodProxy.invokeSuper(o, objects);  System.out.println(">>>>after invoking");  return ret;  }  public static void main(String[] args) {  CGlibAgent cGlibAgent = new CGlibAgent();  Apple apple = (Apple) cGlibAgent.getInstance(new Apple());  apple.show();  }  } |

6 Spring事务管理

事务是逻辑上的一组操作，要么都执行，要么都不执行。事务的特性：

原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

* 一致性： 执行事务前后，数据保持一致；
* 隔离性： 并发访问数据库时，一个用户的事物不被其他事物所干扰，各并发事务之间数据库是独立的；
* 持久性: 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

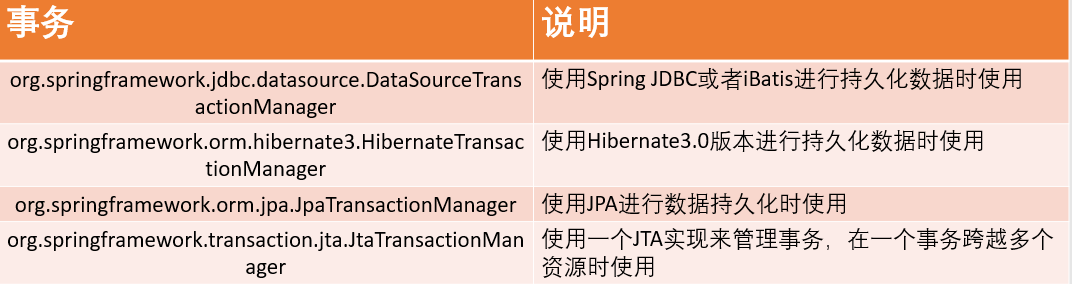
Spring事务管理接口：

* + PlatformTransactionManager： （平台）事务管理器
  + TransactionDefinition： 事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则)
  + TransactionStatus： 事务运行状态

Spring并不直接管理事务，而是提供了多种事务管理器 ，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。 Spring事务管理器的接口是： org.springframework.transaction.PlatformTransactionManager ，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

PlatformTransactionManager接口中定义了三个方法：

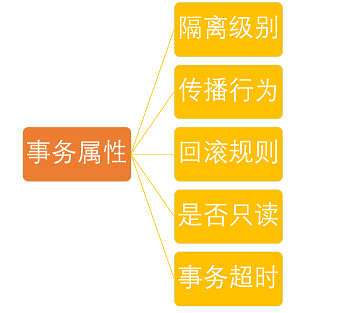
|  |
| --- |
| Public interface PlatformTransactionManager()...{  // Return a currently active transaction or create a new one, according to the specified propagation behavior（根据指定的传播行为，返回当前活动的事务或创建一个新事务。）  TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  // Commit the given transaction, with regard to its status（使用事务目前的状态提交事务）  Void commit(TransactionStatus status) throws TransactionException;  // Perform a rollback of the given transaction（对执行的事务进行回滚）  Void rollback(TransactionStatus status) throws TransactionException;  } |



比如我们在使用JDBC或者iBatis（就是Mybatis）进行数据持久化操作时,我们的xml配置通常如下：

|  |
| --- |
| <!-- 事务管理器 -->  <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- 数据源 -->  <property name="dataSource" ref="dataSource" />  </bean> |

TransactionDefinition接口介绍



TransactionDefinition接口中定义了5个方法以及一些表示事务属性的常量比如隔离级别、传播行为等等的常量。

|  |
| --- |
| public interface TransactionDefinition {  // 返回事务的传播行为  int getPropagationBehavior();  // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据  int getIsolationLevel();  // 返回事务必须在多少秒内完成  //返回事务的名字  String getName()；  int getTimeout();  // 返回是否优化为只读事务。  boolean isReadOnly();  } |

1. 事务隔离级别（定义了一个事务可能受其他并发事务影响的程度）：

* 脏读（Dirty read）: 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
* 丢失修改（Lost to modify）: 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。
* 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
* 不可重复读（Unrepeatableread）: 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
* 幻读（Phantom read）: 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

* TransactionDefinition.ISOLATION\_DEFAULT: 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别.
* TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
* TransactionDefinition.ISOLATION\_READ\_COMMITTED: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
* TransactionDefinition.ISOLATION\_REPEATABLE\_READ: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
* TransactionDefinition.ISOLATION\_SERIALIZABLE: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

1. 事务传播行为（为了解决业务层方法之间互相调用的事务问题）：

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

支持当前事务的情况：

* TransactionDefinition.PROPAGATION\_REQUIRED： 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
* TransactionDefinition.PROPAGATION\_SUPPORTS： 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
* TransactionDefinition.PROPAGATION\_MANDATORY： 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

不支持当前事务的情况：

* TransactionDefinition.PROPAGATION\_REQUIRES\_NEW： 创建一个新的事务，如果当前存在事务，则把当前事务挂起。
* TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED： 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
* TransactionDefinition.PROPAGATION\_NEVER： 以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

* TransactionDefinition.PROPAGATION\_NESTED： 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

(3) 事务超时属性(一个事务允许执行的最长时间)

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

(4) 事务只读属性（对事物资源是否执行只读操作）

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。所谓事务性资源就是指那些被事务管理的资源，比如数据源、 JMS 资源，以及自定义的事务性资源等等。如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。在 TransactionDefinition 中以 boolean 类型来表示该事务是否只读。

(5) 回滚规则（定义事务回滚规则）

这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。

但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

TransactionStatus接口介绍

TransactionStatus接口用来记录事务的状态 该接口定义了一组方法,用来获取或判断事务的相应状态信息.

7 使用Spring框架的好处是什么？

* 轻量：Spring 是轻量的，基本的版本大约2MB。
* 控制反转：Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
* 面向切面的编程(AOP)：Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
* 容器：Spring 包含并管理应用中对象的生命周期和配置。
* MVC框架：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
* 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
* 异常处理：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

8 Spring由哪些模块组成

以下是Spring 框架的基本模块：

Core module

Bean module

Context module

Expression Language module

JDBC module

ORM module

OXM module

Java Messaging Service(JMS) module

Transaction module

Web module

Web-Servlet module

Web-Struts module

Web-Portlet module

来自网上的资料：

**什么是Spring**

Spring是一个开源的Java EE开发框架。Spring框架的核心功能可以应用在任何Java应用程序中，但对Java EE平台上的Web应用程序有更好的扩展性。Spring框架的目标是使得Java EE应用程序的开发更加简捷，通过使用POJO为基础的编程模型促进良好的编程风格。

**Spring有哪些优点**

**轻量级**：Spring在大小和透明性方面绝对属于轻量级的，基础版本的Spring框架大约只有2MB。

**控制反转(IOC)**：Spring使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。

**面向切面编程(AOP)**： Spring支持面向切面编程，同时把应用的**业务逻辑与系统的服务分离开来**。

**容器**：Spring包含并管理应用程序对象的配置，依赖关系和生命周期。

**MVC框架**：Spring的web框架是一个设计优良的web MVC框架，很好的取代了一些web框架。

**事务管理**：Spring对下至本地业务上至全局业务(JAT)提供了统一的事务管理接口。

**异常处理**：Spring提供一个方便的API将特定技术的异常(由JDBC, Hibernate, 或JDO抛出)转化为一致的、Unchecked异常。

**Spring框架有哪些模块**

Spring框架至今已集成了20多个模块。这些模块主要被分如下图所示的核心容器、数据访问/集成、Web、AOP（面向切面编程）、工具、消息和测试模块。

**核心容器模块**：是spring中最核心的模块。负责Bean的创建，配置和管理。主要包括：beans,core,context,expression等模块。

**Spring的AOP模块**：主要负责对面向切面编程的支持，帮助应用对象解耦。

**数据访问和集成模块**：包括JDBC，ORM，OXM，JMS和事务处理模块，其细节如下： JDBC模块提供了不再需要冗长的JDBC编码相关了JDBC的抽象层。 ORM模块提供的集成层。流行的对象关系映射API，包括JPA，JDO，Hibernate和iBatis。 OXM模块提供了一个支持对象/ XML映射实现对JAXB，Castor，使用XMLBeans，JiBX和XStream 的抽象层。 Java消息服务JMS模块包含的功能为生产和消费的信息。 事务模块支持编程和声明式事务管理实现特殊接口类，并为所有的POJO。

**Web和远程调用**：包括web,servlet,struts,portlet模块。

**测试模块**：test

**工具模块消息模块**

**什么是控制反转(IOC)？什么是依赖注入？**

传统模式中对象的调用者需要创建被调用对象，两个对象过于耦合，不利于变化和拓展．在spring中，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，从而实现对象之间的**松耦合**。所谓的“控制反转”概念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。

**依赖注入**：对象无需自行创建或管理它们的依赖关系，IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。依赖注入能让相互协作的软件组件保持松散耦合。

**BeanFactory和ApplicationContext有什么区别？**

Bean工厂(BeanFactory)是Spring框架最核心的接口，提供了高级Ioc的配置机制．

应用上下文(ApplicationContext)建立在BeanFacotry基础之上，提供了更多面向应用的功能，如果国际化，属性编辑器，事件等等．

beanFactory是spring框架的基础设施，是面向spring本身，ApplicationContext是面向使用Spring框架的开发者，几乎所有场合都会用到ApplicationContext.

**Spring有几种配置方式？**

将Spring配置到应用开发中有以下三种方式：

**基于XML的配置**:**基于注解的配置**： Spring在2.5版本以后开始支持用注解的方式来配置依赖注入。可以用注解的方式来替代XML方式的bean描述，可以将bean描述转移到组件类的内部，只需要在相关类上、方法上或者字段声明上使用注解即可。注解注入将会被容器在XML注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结果

**基于Java的配置**： Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。

**Spring Bean的生命周期**

Bean在Spring中的生命周期如下：

**实例化**。Spring通过new关键字将一个Bean进行实例化，JavaBean都有默认的构造函数，因此不需要提供构造参数。

**填入属性**。Spring根据xml文件中的配置通过调用Bean中的setXXX方法填入对应的属性。 事件通知。Spring依次检查Bean是否实现了BeanNameAware、BeanFactoryAware、ApplicationContextAware、BeanPostProcessor、InitializingBean接口，如果有的话，依次调用这些接口。

**使用**。应用程序可以正常使用这个Bean了。

**销毁**。如果Bean实现了DisposableBean接口，就调用其destroy方法。

**---加载过程---**

1.容器寻找Bean的定义信息并且将其实例化。

 2.如果允许提前暴露工厂，则提前暴露这个bean的工厂，这个工厂主要是返回该未完全处理的bean．主要是用于避免单例属性循环依赖问题．

 3.受用**依赖注入**，Spring按照Bean定义信息配置Bean的所有属性。

4.如果Bean实现了**BeanNameAware**接口，工厂调用Bean的**setBeanName()**方法传递Bean的ID。

5.如果Bean实现了**BeanFactoryAware**接口，工厂调用**setBeanFactory()**方法传入工厂自身。

6.如果**BeanPostProcessor**和Bean关联，那么它们的**postProcessBeforeInitialzation()**方法将被调用。

 7.如果Bean指定了init-method方法，它将被调用。

8.如果有**BeanPostProcessor**和Bean关联，那么它们的postProcessAfterInitialization()方法将被调用

9.最后如果配置了destroy-method方法则注册**DisposableBean**.

到这个时候，Bean已经可以被应用系统使用了，并且将被保留在Bean Factory中知道它不再需要。 有两种方法可以把它从Bean Factory中删除掉。

1.如果Bean实现了DisposableBean接口，destory()方法被调用。

2.如果指定了订制的销毁方法，就调用这个方法。

**Spring Bean的作用域之间有什么区别**

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。

**请解释自动装配模式的区别**

**no**：这是Spring框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在bean定义中用标签明确的设置依赖关系。

**byName\*\***：该选项可以根据bean名称设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

**byType**：该选项可以根据bean类型设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的类型自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

**constructor**：造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。

**autodetect**：该模式自动探测**使用构造器自动装配或者byType自动装配**。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的构造器或者是无参构造器，容器就会自动选择byTpe的自动装配方式。

**Spring 框架中都用到了哪些设计模式**

**代理模式**—在AOP和remoting中被用的比较多。

**单例模式**—在spring配置文件中定义的bean默认为单例模式。

**模板方法**—用来解决代码重复的问题 比如. RestTemplate, JmsTemplate, JpaTemplate。 前端控制器—Srping提供了DispatcherServlet来对请求进行分发。 视图帮助(View Helper )—Spring提供了一系列的JSP标签，高效宏来辅助将分散的代码整合在视图里。 依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念。

**工厂模式**—BeanFactory用来创建对象的实例。

**Builder模式**- 自定义配置文件的解析bean是时采用builder模式，一步一步地构建一个beanDefinition

**策略模式**：Spring 中策略模式使用有多个地方，如 Bean 定义对象的创建以及代理对象的创建等。这里主要看一下代理对象创建的策略模式的实现。 前面已经了解 Spring 的代理方式有两个 Jdk 动态代理和 CGLIB 代理。这两个代理方式的使用正是使用了策略模式。

**AOP是怎么实现的**

实现AOP的技术，主要分为两大类：

一是采用**动态代理技术**，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成,并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异，AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。

Spring AOP使用动态代理技术在运行期织入增强代码。使用两种代理机制：

基于**JDK的动态代理**（JDK本身只提供接口的代理）；

**基于CGlib的动态代理**。

1)JDK的动态代理主要涉及java.lang.reflect包中的两个类：Proxy和InvocationHandler。其中InvocationHandler只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。 其代理对象**必须是某个接口的实现**,它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理,而**不能针对类**

2)CGLib采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.所以无法通知final的方法,因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类,覆盖其中方法.

在spring中默认情况下使用JDK动态代理实现AOP,如果proxy-target-class设置为true或者使用了优化策略那么会使用CGLIB来创建动态代理.Spring　AOP在这两种方式的实现上基本一样．以JDK代理为例，会使用JdkDynamicAopProxy来创建代理，在invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入．最终返回代理对象．

[http://zhengjianglong.cn/2015/12/12/Spring/spring-source-aop/](https://link.jianshu.com?t=http:/zhengjianglong.cn/2015/12/12/Spring/spring-source-aop/)

介绍spring的IOC实现

Spring　IOC主要负责创建和管理bean及bean之间的依赖关系．

Spring　IOC的可分为:**IOC容器的初始化和bean的加载**．

在IOC容器阶段主要是完成资源的加载(如定义bean的xml文件)，bean的解析及对解析后得到的beanDefinition的进行注册．以xmlBeanFactory为例，XmlBeanFactory继承了DefaultListableBeanFactory，XmlBeanFactory将读取xml配置文件，解析bean和注册解析后的beanDefinition工作交给XmlBeanDefinitionReader(是BeanDefinitionReader接口的一个个性化实现)来执行.spring中定义了一套资源类，将文件，class等都看做资源．

1)所以首先是将xml文件转化为资源然后用EncodeResouce来封装，该功能主要时考虑Resource可能存在编码要求的情况，如UTF-8等．

 2)然后根据xml文件判断xml的约束模式，是DTD还是Schema,以及寻找模式文档(验证文件)的方法(EntityResolver，这部分采用了代理模式和策略模式)． 完成了前面所有的准备工作以后就可以正式的加载配置文件，获取Docoment和解析注册BeanDefinition．Docoment的获取以及BeanDefinition的解析注册并不是由XmlBeanDefinitionReader完成，XmlBeanDefinitionReader只是将前面的工作完成以后文档加载交给DefaultDocumentLoader类来完成．而解析交给了DefaultBeanDefinitionDocumentReader来处理.bean标签可以分为两种，一种是spring自带的默认标签，另一种就是用户自定义的标签．所以spring针对这两种情况，提供了不同的解析方式. 每种bean的解析完成后都会先注册到容器中然后最后发出响应事件，通知相关的监听器这个bean已经注册完成了．

**bean的加载**[http://zhengjianglong.cn/2015/12/06/Spring/spring-source-ioc-bean-parse/](https://link.jianshu.com?t=http:/zhengjianglong.cn/2015/12/06/Spring/spring-source-ioc-bean-parse/)

**springMVC流程具体叙述下**

当应用启动时,容器会加载servlet类并调用init方法. 在这个阶段，DispatcherServlet在init()完成初始化参数init-param的解析和封装,相关配置,spring的WebApplicationContext的初始化即完成xml文件的加载,bean的解析和注册等工作,另外为servlet功能所用的变量进行初始化,如:handlerMapping,viewResolvers等.

当用户发送一个请求时，首先根据请求的类型调用DispatcherServlet不同的方法，这些方法都会转发到doService()中执行．在该方法内部完成以下工作：

1)spring首先考虑multipart的处理,如果是MultipartContent类型的request,则将该请求转换成MultipartHttpServletRequest类型的request.

2)根据request信息获取对应的Handler. 首先根据request获取访问路径,然后根据该路径可以选择直接匹配或通用匹配的方式寻找Handler,即用户定义的controller. Handler在init()方法时已经完成加载且保存到Map中了,只要根据路径就可以得到对应的Handler. 如果不存在则尝试使用默认的Handler. 如果还是没有找到那么就通过response向用户返回错误信息.找到handler后会将其包装在一个执行链中,然后将所有的拦截器也加入到该链中.

4)如果存在handler则根据当前的handler寻找对应的HandlerAdapter. 通过遍历所有适配器来选择合适的适配器.

5)SpringMVC允许你通过处理拦截器Web请求,进行前置处理和后置处理.所以在正式调用 Handler的逻辑方法时,先执行所有拦截器的preHandle()方法.

6)正式执行handle的业务逻辑方法handle(),返回ModelAndView.逻辑处理是通过适配器调用handle并返回视图.这过程其实是调用用户controller的业务逻辑.

8)调用拦截器的postHandle()方法,完成后置处理.

9)根据视图进行页面跳转.该过程首先会根据视图名字解析得到视图,该过程支持缓存,如果缓存中存在则直接获取,否则创建新的视图并在支持缓存的情况下保存到缓冲中.

10)过程完成了像添加前缀后缀,设置必须的属性等工作.最后就是进行页面跳转处理.

11)调用拦截器的afterComplection()

AOP相关概念

**方面（Aspect）**：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用Spring的 Advisor或拦截器实现。

**连接点（Joinpoint）**: 程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

**通知（Advice）:**在特定的连接点，AOP框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多AOP框架包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring中定义了4个advice.Interception Around(MethodInterceptor)、Before(MethodBeforeAdvice)、After Returning(AfterReturningAdvice)、After(AfterAdvice)。

**切入点（Pointcut）**: 一系列连接点的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。 Spring定义了Pointcut接口，用来组合MethodMatcher和ClassFilter，可以通过名字很清楚的理解， MethodMatcher是用来检查目标类的方法是否可以被应用此通知，而ClassFilter是用来检查Pointcut是否应该应用到目标类上

**引入（Introduction）**: 添加方法或字段到被通知的类。 Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。Spring中要使用Introduction, 可有通过DelegatingIntroductionInterceptor来实现通知，通过DefaultIntroductionAdvisor来配置Advice和代理类要实现的接口

**目标对象（Target Object）**: 包含连接点的对象。也被称作被通知或被代理对象。

POJO**AOP代理（AOP Proxy）**: AOP框架创建的对象，包含通知。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。

**织入（Weaving）**: 组装方面来创建一个被通知对象。这可以在编译时完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

**过滤器与监听器的区别**

Filter可认为是Servlet的一种“变种”，它主要用于对用户请求进行预处理，也可以对HttpServletResponse进行后处理，是个典型的处理链。它与Servlet的区别在于：它不能直接向用户生成响应。完整的流程是：Filter对用户请求进行预处理，接着将请求交给 Servlet进行处理并生成响应，最后Filter再对服务器响应进行后处理。 Java中的Filter 并不是一个标准的Servlet ，它不能处理用户请求，也不能对客户端生成响应。 主要用于对HttpServletRequest 进行预处理，也可以对HttpServletResponse 进行后处理，是个典型的处理链。优点：过滤链的好处是，执行过程中任何时候都可以打断，只要不执行chain.doFilter()就不会再执行后面的过滤器和请求的内容。而在实际使用时，就要特别注意过滤链的执行顺序问题[http://blog.csdn.net/sd0902/article/details/8395641](https://link.jianshu.com?t=http:/blog.csdn.net/sd0902/article/details/8395641)

Servlet,Filter都是针对url之类的，而Listener是针对对象的操作的，如session的创建，session.setAttribute的发生，或者在启动服务器的时候将你需要的数据加载到缓存等，在这样的事件发生时做一些事情。[http://www.tuicool.com/articles/bmqMjm](https://link.jianshu.com?t=http:/www.tuicool.com/articles/bmqMjm)

**请描述一下java事件监听机制。**

(1) Java的事件监听机制涉及到三个组件：事件源、事件监听器、事件对象

(2) 当事件源上发生操作时，它将会调用事件监听器的一个方法，并在调用这个方法时，会传递事件对象过来

(3) 事件监听器由开发人员编写，开发人员在事件监听器中，通过事件对象可以拿到事件源，从而对事件源上的操作进行处理。

**解释核心容器(应用上下文)模块**

这是Spring的基本模块，它提供了Spring框架的基本功能。BeanFactory 是所有Spring应用的核心。Spring框架是建立在这个模块之上的，这也使得Spring成为一个容器。

**BeanFactory – BeanFactory 实例**

BeanFactory是工厂模式的一种实现，它使用控制反转将应用的配置和依赖与实际的应用代码分离开来。最常用的BeanFactory实现是XmlBeanFactory类。

XmlBeanFactory

最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory，它根据XML文件中定义的内容加载beans。该容器从XML文件中读取配置元数据，并用它来创建一个完备的系统或应用。

**解释AOP模块**

AOP模块用来开发Spring应用程序中具有切面性质的部分。该模块的大部分服务由AOP Aliance提供，这就保证了Spring框架和其他AOP框架之间的互操作性。另外，该模块将元数据编程引入到了Spring。

**解释抽象JDBC和DAO模块**

通过使用抽象JDBC和DAO模块保证了与数据库连接代码的整洁与简单，同时避免了由于未能关闭数据库资源引起的问题。它在多种数据库服务器的错误信息之上提供了一个很重要的异常层。它还利用Spring的AOP模块为Spring应用程序中的对象提供事务管理服务。

**解释对象/关系映射集成模块**

Spring通过提供ORM模块在JDBC的基础上支持对象关系映射工具。这样的支持使得Spring可以集成主流的ORM框架，包括Hibernate, JDO, 及iBATIS SQL Maps。Spring的事务管理可以同时支持以上某种框架和JDBC。

**解释web模块**

Spring的web模块建立在应用上下文(application context)模块之上，提供了一个适合基于web应用程序的上下文环境。该模块还支持了几个面向web的任务，如透明的处理多文件上传请求及将请求参数同业务对象绑定起来。

**解释Spring MVC模块**

Spring提供MVC框架构建web应用程序。Spring可以很轻松的同其他MVC框架结合，但Spring的MVC是个更好的选择，因为它通过控制反转将控制逻辑和业务对象完全分离开来。

**ContextLoaderListener是监听什么事件**

ContextLoaderListener的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法。

**Spring IoC容器**

Spring IOC负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

**IOC有什么优点？**

IOC或依赖注入减少了应用程序的代码量。它使得应用程序的**测试很简单**，因为在单元测试中不再需要单例或JNDI查找机制。简单的实现以及较少的干扰机制使得**松耦合**得以实现。IOC容器支持勤性单例及延迟加载服务。

**应用上下文是如何实现的？**

ClassPathXmlApplicationContext 容器加载XML文件中beans的定义。XML Bean配置文件的完整路径必须传递给构造器。 FileSystemXmlApplicationContext 容器也加载XML文件中beans的定义。注意，你需要正确的设置CLASSPATH，因为该容器会在CLASSPATH中查看bean的XML配置文件。 WebXmlApplicationContext：该容器加载xml文件，这些文件定义了web应用中所有的beans。

**有哪些不同类型的IOC(依赖注入)**

**接口注入**:接口注入的意思是通过接口来实现信息的注入，而其它的类要实现该接口时，就可以实现了注入 构造器依赖注入：构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。 Setter方法依赖注入：首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用bean中的setter方法完成Setter方法依赖注入。

你推荐哪种依赖注入？构造器依赖注入还是Setter方法依赖注入？

你可以同时使用两种方式的依赖注入，最好的选择是使用构造器参数实现强制依赖注入，使用setter方法实现可选的依赖关系。

**什么是Spring Beans**

Spring Beans是构成Spring应用核心的Java对象。这些对象由Spring IOC容器实例化、组装、管理。这些对象通过容器中配置的元数据创建，例如，使用XML文件中定义的创建。 在Spring中创建的beans都是单例的beans。在bean标签中有一个属性为”singleton”,如果设为true，该bean是单例的，如果设为false，该bean是原型bean。Singleton属性默认设置为true。因此，spring框架中所有的bean都默认为单例bean。

**Spring Bean中定义了什么内容？**

Spring Bean中定义了所有的配置元数据，这些配置信息告知容器如何创建它，它的生命周期是什么以及它的依赖关系。

**如何向Spring 容器提供配置元数据**

有三种方式向Spring 容器提供元数据: XML配置文件 基于注解配置 基于Java的配置

**你如何定义bean的作用域**

在Spring中创建一个bean的时候，我们可以声明它的作用域。只需要在bean定义的时候通过’scope’属性定义即可。例如，当Spring需要产生每次一个新的bean实例时，应该声明bean的scope属性为prototype。如果每次你希望Spring返回一个实例，应该声明bean的scope属性为singleton。

**Spring框架中单例beans是线程安全的吗？**

不是，Spring框架中的单例beans不是线程安全的。

**哪些是最重要的bean生命周期方法？能重写它们吗？**

有两个重要的bean生命周期方法。第一个是setup方法，该方法在容器加载bean的时候被调用。第二个是teardown方法，该方法在bean从容器中移除的时候调用。 bean标签有两个重要的属性(init-method 和 destroy-method)，你可以通过这两个属性定义自己的初始化方法和析构方法。Spring也有相应的注解：@PostConstruct 和 @PreDestroy。

**什么是Spring的内部bean**

当一个bean被用作另一个bean的属性时，这个bean可以被声明为内部bean。在基于XML的配置元数据中，可以通过把元素定义在 或元素内部实现定义内部bean。内部bean总是匿名的并且它们的scope总是prototype。

**如何在Spring中注入Java集合类**

Spring提供如下几种类型的集合配置元素： list元素用来注入一系列的值，允许有相同的值。 set元素用来注入一些列的值，不允许有相同的值。 map用来注入一组”键-值”对，键、值可以是任何类型的。 props也可以用来注入一组”键-值”对，这里的键、值都字符串类型。

**什么是bean wiring？**

Wiring，或者说bean Wiring是指beans在Spring容器中结合在一起的情况。当装配bean的时候，Spring容器需要知道需要哪些beans以及如何使用依赖注入将它们结合起来。

**什么是bean自动装配？**

Spring容器可以自动配置相互协作beans之间的关联关系。这意味着Spring可以自动配置一个bean和其他协作bean之间的关系，通过检查BeanFactory 的内容里没有使用和< property>元素。

**解释自动装配的各种模式**

自动装配提供五种不同的模式供Spring容器用来自动装配beans之间的依赖注入: no：默认的方式是不进行自动装配，通过手工设置ref 属性来进行装配bean。 byName：通过参数名自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byName。之后容器试图匹配、装配和该bean的属性具有相同名字的bean。 byType：通过参数的数据类型自动自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byType。之后容器试图匹配和装配和该bean的属性类型一样的bean。如果有多个bean符合条件，则抛出错误。 constructor：这个同byType类似，不过是应用于构造函数的参数。如果在BeanFactory中不是恰好有一个bean与构造函数参数相同类型，则抛出一个严重的错误。 autodetect：如果有默认的构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

**自动装配有哪些局限性？**

自动装配有如下局限性： 重写：你仍然需要使用 和< property>设置指明依赖，这意味着总要重写自动装配。 原生数据类型:你不能自动装配简单的属性，如原生类型、字符串和类。 模糊特性：自动装配总是没有自定义装配精确，因此，如果可能尽量使用自定义装配。

**你可以在Spring中注入null或空字符串吗**

完全可以。

**什么是Spring基于Java的配置？给出一些注解的例子**

基于Java的配置允许你使用Java的注解进行Spring的大部分配置而非通过传统的XML文件配置。以注解@Configuration为例，它用来标记类，说明作为beans的定义，可以被Spring IOC容器使用。另一个例子是@Bean注解，它表示该方法定义的Bean要被注册进Spring应用上下文中。

**什么是基于注解的容器配置**

另外一种替代XML配置的方式为基于注解的配置，这种方式通过字节元数据装配组件而非使用尖括号声明。开发人员将直接在类中进行配置，通过注解标记相关的类、方法或字段声明，而不再使用XML描述bean之间的连线关系。

**如何开启注解装配？**

注解装配默认情况下在Spring容器中是不开启的。如果想要开启基于注解的装配只需在Spring配置文件中配置元素即可。

**@Required 注解**

@Required表明bean的属性必须在配置时设置，可以在bean的定义中明确指定也可通过自动装配设置。如果bean的属性未设置，则抛出BeanInitializationException异常。

**@Autowired 注解**

@Autowired 注解提供更加精细的控制，包括自动装配在何处完成以及如何完成。它可以像@Required一样自动装配setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

**@Qualifier 注解**

当有多个相同类型的bean而只有其中的一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用消除这种混淆，指明需要装配的bean。 Spring数据访问

**在Spring框架中如何更有效的使用JDBC？**

使用Spring JDBC框架，资源管理以及错误处理的代价都会减轻。开发人员只需通过statements和queries语句从数据库中存取数据。Spring框架中通过使用模板类能更有效的使用JDBC，也就是所谓的JdbcTemplate。

**JdbcTemplate**

JdbcTemplate类提供了许多方法，为我们与数据库的交互提供了便利。例如，它可以将数据库的数据转化为原生类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据库错误处理功能。

**Spring对DAO的支持**

Spring对数据访问对象(DAO)的支持旨在使它可以与数据访问技术(如 JDBC, Hibernate 及JDO)方便的结合起来工作。这使得我们可以很容易在的不同的持久层技术间切换，编码时也无需担心会抛出特定技术的异常。

**使用Spring可以通过什么方式访问Hibernate？**

使用Spring有两种方式访问Hibernate： 使用Hibernate Template的反转控制以及回调方法 继承HibernateDAOSupport，并申请一个AOP拦截器节点

**Spring支持的ORM**

Spring支持一下ORM： Hibernate iBatis JPA (Java -Persistence API) TopLink JDO (Java Data Objects) OJB

**如何通过HibernateDaoSupport将Spring和Hibernate结合起来？**

使用Spring的SessionFactory 调用LocalSessionFactory。结合过程分为以下三步： 配置Hibernate SessionFactory 继承HibernateDaoSupport实现一个DAO 使用AOP装载事务支持

**Spring支持的事务管理类型**

Spring支持如下两种方式的事务管理： 编码式事务管理：sping对编码式事务的支持与EJB有很大区别，不像EJB与java事务API耦合在一起．spring通过回调机制将实际的事务实现从事务性代码中抽象出来．你能够精确控制事务的边界，它们的开始和结束完全取决于你． 声明式事务管理：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。通过传播行为，隔离级别，回滚规则，事务超时，只读提示来定义．

**Spring框架的事务管理有哪些优点**

它为不同的事务API(如JTA, JDBC, Hibernate, JPA, 和JDO)提供了统一的编程模型。 它为编程式事务管理提供了一个简单的API而非一系列复杂的事务API(如JTA). 它支持声明式事务管理。 它可以和Spring 的多种数据访问技术很好的融合。

ACID

**原子性(Atomic)**:一个操作要么成功，要么全部不执行.

**一致性(Consistent)**: 一旦事务完成，系统必须确保它所建模业务处于一致的状态

**隔离性(Isolated)**: 事务允许多个用户对相同的数据进行操作，每个用户用户的操作相互隔离互补影响．

**持久性(Durable)**: 一旦事务完成，事务的结果应该持久化．

**spring事务定义的传播规则**

PROPAGATION\_REQUIRED–支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

PROPAGATION\_SUPPORTS–支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION\_MANDATORY–支持当前事务，如果当前没有事务，就抛出异常。

PROPAGATION\_REQUIRES\_NEW–新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION\_NOT\_SUPPORTED–以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

PROPAGATION\_NEVER–以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION\_NESTED–如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。

**spring事务支持的隔离级别**

并发会导致以下问题：

藏读：发生在一个事务读取了另一个事务改写但尚未提交的数据．

不可重复读：在一个事务执行相同的查询两次或两次以上，每次得到的数据不同．

幻读：与不可重复读类似，发生在一个事务读取多行数据，接着另一个并发事务插入一些数据，随后查询中，第一个事务发现多了一些原本不存在的数据．

spring 事务上提供以下的隔离级别:

ISOLATION\_DEFAULT: 使用后端数据库默认的隔离级别

ISOLATION\_READ\_UNCOMMITTED　: 允许读取未提交的数据变更，可能会导致脏读，幻读或不可重复读

 ISOLATION\_READ\_COMMITTD : 允许读取为提交数据,可以阻止脏读，当时幻读或不可重复读仍可能发生

 ISOLATION\_REPEATABLE\_READ: 对统一字段多次读取结果是一致的，除非数据是被本事务自己修改．可以阻止脏读，不可重复读，但幻读可能发生

ISOLATION\_SERIALIZABLE :　完全服从ACID

**你更推荐那种类型的事务管理？**

许多Spring框架的用户选择声明式事务管理，因为这种方式和应用程序的关联较少，因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理，尽管在灵活性方面它弱于编程式事务管理(这种方式允许你通过代码控制业务)。

**有几种不同类型的自动代理？**

BeanNameAutoProxyCreator：bean名称自动代理创建器 DefaultAdvisorAutoProxyCreator：默认通知者自动代理创建器 Metadata autoproxying：元数据自动代理

**什么是织入？什么是织入应用的不同点？**

织入是将切面和其他应用类型或对象连接起来创建一个通知对象的过程。织入可以在编译、加载或运行时完成。

**什么是Spring的MVC框架？**

Spring提供了一个功能齐全的MVC框架用于构建Web应用程序。Spring框架可以很容易的和其他的MVC框架融合(如Struts)，该框架使用控制反转(IOC)将控制器逻辑和业务对象分离开来。它也允许以声明的方式绑定请求参数到业务对象上。

**DispatcherServlet**

Spring的MVC框架围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

**WebApplicationContext**

WebApplicationContext继承了ApplicationContext，并添加了一些web应用程序需要的功能。和普通的ApplicationContext 不同，WebApplicationContext可以用来处理主题样式，它也知道如何找到相应的servlet。

**什么是Spring MVC框架的控制器？**

控制器提供对应用程序行为的访问，通常通过服务接口实现。控制器解析用户的输入，并将其转换为一个由视图呈现给用户的模型。Spring 通过一种极其抽象的方式实现控制器，它允许用户创建多种类型的控制器。

@Controller annotation

@Controller注解表示该类扮演控制器的角色。Spring不需要继承任何控制器基类或应用Servlet API。

@RequestMapping annotation

@RequestMapping注解用于将URL映射到任何一个类或者一个特定的处理方法上。