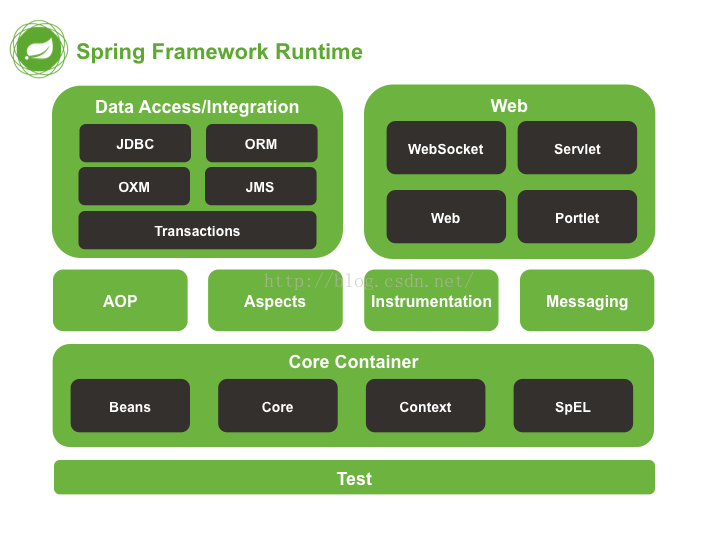
框架面试

###### Spring

**1). 什么是Spring框架？Spring框架有哪些主要模块？**

i)Spring框架是一个为Java应用程序的开发提供了综合、广泛的基础性支持的Java平台;   
ii).Spring帮助开发者解决了开发中基础性的问题，使得开发人员可以专注于应用程序的开发;

iii). Spring框架本身亦是按照[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)精心打造，这使得可以在开发环境中安心的集成Spring框架，不必担心Spring是如何在后台进行工作的;



**2).Spring能带来哪些好处**

i). [Dependency Injection(DI: 依赖注入)](http://howtodoinjava.com/2013/03/19/inversion-of-control-ioc-and-dependency-injection-di-patterns-in-spring-framework-and-related-interview-questions/) 方法使得构造器和JavaBean properties文件中的依赖关系一目了然;

ii). 与EJB容器相比较，IoC容器更加趋向于轻量级。这样一来IoC容器在有限的内存和CPU资源的情况下进行应用程序的开发和发布就变得十分有利;

iii). Spring利用了已有的技术比如ORM框架、logging框架、J2EE、Quartz和JDK Timer，以及其他视图技术;

iv). Spring框架是按照模块的形式来组织的。由包和类的编号就可以看出其所属的模块，开发者仅仅需要选用他们需要的模块即可;

v). Spring提供了一个便捷的事务管理接口，适用于小型的本地事物处理（比如在单DB的环境下）和复杂的共同事物处理（比如利用JTA的复杂DB环境）.

**3) .什么是控制反转, 什么是依赖注入**

**控制反转:** 软件中的对象就像齿轮一样，协同工作，但是互相耦合，一个零件不能正常工作，整个系统就崩溃了。这是一个强耦合的系统。齿轮组中齿轮之间的啮合关系,与软件系统中对象之间的耦合关系非常相似。对象之间的耦合关系是无法避免的，也是必要的，这是协同工作的基础。现在，伴随着工业级应用的规模越来越庞大，对象之间的依赖关系也越来越复杂，经常会出现对象之间的多重依赖性关系; 为了解决对象间耦合度过高的问题，软件专家Michael Mattson提出了IOC理论，用来实现对象之间的“解耦”. **控制反转（Inversion of Control）**是一种是面向对象编程中的一种设计原则，用来减低计算机代码之间的耦合度。其基本思想是：借助于“第三方”实现具有依赖关系的对象之间的解耦.

软件系统在没有引入IOC容器之前，如图1所示，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上:

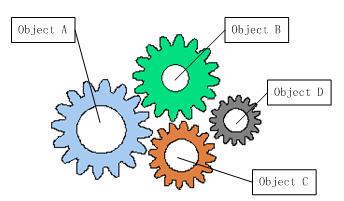
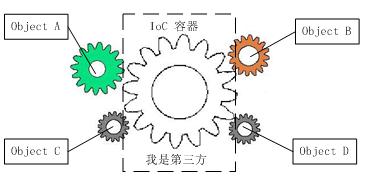
 

图1 图2

系统在引入IOC容器之后，这种情形就完全改变了，如图2所示，由于IOC容器的加入，对象A与对象B之间失去了直接联系，所以，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方; 通过前后的对比，我们不难看出来：对象A获得依赖对象B的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来.

依赖注入: 如果在 Class A 中，有 Class B 的实例，则称 Class A 对 Class B 有一个依赖。例如下面类 Human 中用到一个 Father 对象，我们就说类 Human 对类 Father 有一个依赖。

public class Human {

...

Father father;

...

public Human() {

father = new Father();

}

}

仔细看这段代码我们会发现存在一些问题：如果现在要改变 father 生成方式，如需要用new Father(String name)初始化 father，需要修改 Human 代码；如果想测试不同 Father 对象对 Human 的影响很困难，因为 father 的初始化被写死在了 Human 的构造函数中；如果new Father()过程非常缓慢，单测时我们希望用已经初始化好的 father 对象 Mock 掉这个过程也很困难。

**依赖注入的实现**: 将 father 对象作为构造函数的一个参数传入。在调用 Human 的构造方法之前外部就已经初始化好了 Father 对象。像这种**非自己主动初始化依赖，而通过外部来传入依赖的方式，就称为依赖注入**.

public class Human {

...

Father father;

...

public Human(Father father) {

this.father = father;

}

}

**控制反转和依赖注入的关系**:控制反转是一种思想,依赖注入是一种设计模式的实现.

在Spring中实现依赖注入的方式有三种:构造器注入;Setter方法注入;接口注入.

**Spring中通过IoC(控制反转)容器来实现依赖注入:**

Spring中的 org.springframework.beans 包和 org.springframework.context包构成了Spring框架IoC容器的基础; BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与[Spring的AOP](http://howtodoinjava.com/category/frameworks/java-spring-tutorials/spring-aop/)更容易集成，也提供了处理[message resource的机制](http://howtodoinjava.com/2015/02/10/spring-mvc-internationalization-i18n-and-localization-i10n-example/)（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

IOC:把对象的创建、初始化、销毁交给spring来管理，而不是由开发者控制，实现控制反转.

Spring中实现IoC的方式: 首先需要配置bean.xml文件:

class MovieLister...

private MovieFinder finder;

public void setFinder(MovieFinder finder) {

this.finder = finder;

}

class MovieFinder...

public void setFilename(String filename) {

this.filename = filename;

}

bean.xml

<beans>

<bean id="MovieLister" class="spring.MovieLister">

<property name="finder">

<ref local="MovieFinder"/>

</property>

</bean>

<bean id="MovieFinder" class="spring.MovieFinder">

<property name="filename">

<value>movies1.txt</value>

</property>

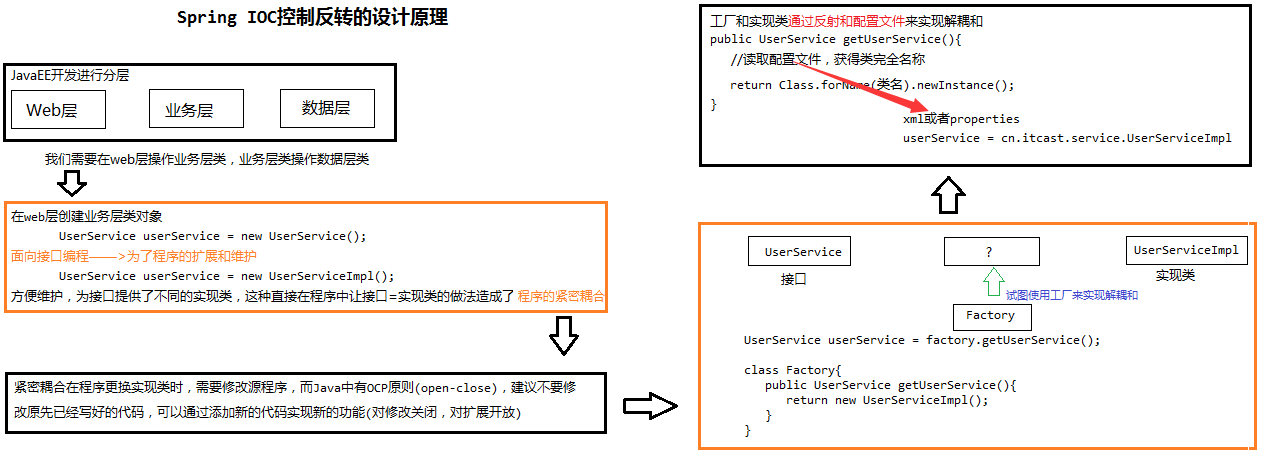
</bean>

</beans>

在Spring中，每个bean代表一个对象的实例，默认是单例模式，即在程序的生命周期内，所有的对象都只有一个实例，进行重复使用。通过配置bean，IoC容器在启动的时候会根据配置生成bean实例。具体的配置语法参考Spring文档。这里只要知道IoC容器会根据配置创建MovieFinder，在运行的时候把MovieFinder赋值给MovieLister的finder属性，完成依赖注入的过程.

**IoC(控制反转)与DI(依赖注入)的区别:**

IoC Inverse of Control 反转控制的概念，就是将原本在程序中手动创建UserService对象的控制权，交由Spring框架管理，简单说，就是创建UserService对象控制权被反转到了Spring框架, DI：Dependency Injection 依赖注入，在Spring框架负责创建Bean对象时，动态的将依赖对象注入到Bean组件:



IoC 控制反转，指将对象的创建权，反转到Spring容器 ， DI 依赖注入，指Spring创建对象的过程中，将对象依赖属性通过配置进行注入.

**4).BeanFactory与ApplicationContext的区别**

BeanFactory 可以理解为含有bean集合的工厂类。BeanFactory 包含了种bean的定义，以便在接收到客户端请求时将对应的bean实例化。BeanFactory还能在实例化对象的时生成协作类之间的关系。此举将bean自身与bean客户端的配置中解放出来。BeanFactory还包含了bean生命周期的控制，调用客户端的初始化方法（initialization methods）和销毁方法（destruction methods）。从表面上看，application context如同bean factory一样具有bean定义、bean关联关系的设置，根据请求分发bean的功能。但applicationcontext在此基础上还提供了其他的功能:

i). 提供了支持国际化的文本消息;

ii). 统一的资源文件读取方式;

iii). 已在监听器中注册的bean的时间

三种ApplicationContext的获取方式:

i). ClassPathXmlApplicationContext：从classpath的XML配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得: ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”);

ii). FileSystemXmlApplicationContext ：由文件系统中的XML配置文件读取上下文: ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”);

iii). XmlWebApplicationContext：由Web应用的XML文件读取上下文.

**5).Spring的配置方式**

Spring配置主要有三种方式:

i).基于XML的配置: SpringXML配置的主要目的时候是使所有的Spring组件都可以用xml文件的形式来进行配置, Spring的XML配置方式是使用被Spring命名空间的所支持的一系列的XML标签来实现的。Spring有以下主要的命名空间：context、beans、jdbc、tx、aop、mvc和aso;

ii).基于注解的配置: 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; 注解装配在Spring中是默认关闭的。所以需要在Spring文件中配置一下才能使用基于注解的装配模式。如果你想要在你的应用程序中使用关于注解的方法的话,需要加入下面语句:

<beans>

   <context:annotation-config/>

   <!-- bean definitions go here -->

</beans>

常用的注解包括: @Required：该注解应用于设值方法, @Autowired：该注解应用于有值设值方法、非设值方法、构造方法和变量, @Qualifier：该注解和@Autowired注解搭配使用，用于消除特定bean自动装配的歧义,JSR-250 Annotations：Spring支持基于JSR-250 注解的以下注解，@Resource、@PostConstruct 和 @PreDestroy.

iii).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<bean/> 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.

**6).Bean的生命周期**

在一个bean实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个bean不在被调用时需要进行相关的析构操作，并从bean容器中移除,Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成: 初始化之后调用的回调方法; 销毁之前调用的回调方法.



**i)实例化Bean:** 对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化, 对于ApplicationContext容器，当容器启动结束后，便实例化所有的bean, 容器通过获取BeanDefinition对象中的信息进行实例化。并且这一步仅仅是简单的实例化，并未进行依赖注入, 实例化对象被包装在BeanWrapper对象中，BeanWrapper提供了设置对象属性的接口，从而避免了使用反射机制设置属性.

**ii).注入Bean属性(依赖注入):** 实例化后的对象被封装在BeanWrapper对象中，并且此时对象仍然是一个原生的状态，并没有进行依赖注入, 紧接着，Spring根据BeanDefinition中的信息进行依赖注入, 并且通过BeanWrapper提供的设置属性的接口完成依赖注入.

**iii).注入Aware接口**: Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给bean;

**iv).BeanPostProcessor:** 当经过上述几个步骤后，bean对象已经被正确构造, 但如果想要对象被使用前再进行一些自定义的处理，就可以通过BeanPostProcessor接口实现, 该接口提供了两个函数:

postProcessBeforeInitialzation( Object bean, String beanName ) ,当前正在初始化的bean对象会被传递进来，我们就可以对这个bean作任何处理。 这个函数会先于InitialzationBean执行，因此称为前置处理。所有Aware接口的注入就是在这一步完成的

postProcessAfterInitialzation( Object bean, String beanName ) ,当前正在初始化的bean对象会被传递进来，我们就可以对这个bean作任何处理。 这个函数会在InitialzationBean完成后执行，因此称为后置处理.

**v). InitializingBean与init-method:** 当BeanPostProcessor的前置处理完成后就会进入本阶段, InitializingBean接口只有一个函数afterPropertiesSet(),这一阶段也可以在bean正式构造完成前增加我们自定义的逻辑，但它与前置处理不同，由于该函数并不会把当前bean对象传进来，因此在这一步没办法处理对象本身，只能增加一些额外的逻辑, 若要使用它，我们需要让bean实现该接口，并把要增加的逻辑写在该函数中。然后Spring会在前置处理完成后检测当前bean是否实现了该接口，并执行afterPropertiesSet函数, 当然，Spring为了降低对客户代码的侵入性，给bean的配置提供了init-method属性，该属性指定了在这一阶段需要执行的函数名。Spring便会在初始化阶段执行我们设置的函数。init-method本质上仍然使用了InitializingBean接口

**vi). DisposableBean和destroy-method:** 和init-method一样，通过给destroy-method指定函数，就可以在bean销毁前执行指定的逻辑.

**7). Spring中Bean作用于的区别**

Spring容器中的bean可以分为5个范围:

**i). 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;

**ii).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;

**iii). 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;

**iv). 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;

**v). 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, 全局作用域与Servlet中的session作用域效果相同.

**8).Spring中的Beans是否是线程安全的**

Spring框架并没有对[单例](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/)bean进行任何多线程的封装处理。关于单例bean的[线程安全](http://howtodoinjava.com/2014/06/02/what-is-thread-safety/)和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全, 最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”.

**9).Spring中注入Collection**

Spring提供了以下四种集合类的配置元素

i). <list> :   该标签用来装配可重复的list值;

ii). <set> :    该标签用来装配没有重复的set值;

iii). <map>:   该标签可用来注入键和值可以为任何类型的键值对;

iv). <props> : 该标签支持注入键和值都是字符串类型的键值对,如下:

**10).Spring Bean的自动装配**

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring容器还可以自动装配合作关系bean之间的关联关系。这意味着Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上, 下面的XML配置文件表明了如何根据名称将一个bean设置为自动装配:

<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

除了bean配置文件中提供的自动装配模式，还可以使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在按照如下的配置方式在Spring配置文件进行配置才可以使用:

<context:annotation-config />

配置好以后就可以使用@Autowired来标注了.

Spring中开启自动装配的两种方式:

i). 引入配置文件中的<bean>下引入 <context:annotation-config>;

ii). 在bean配置文件中直接引入AutowiredAnnotationBeanPostProcessor

<beans>

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

</beans>

**11).Spring中的自动装配模式**

Spring中共有五种自动装配:

i). 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;

ii). 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;

iii). 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;

iv). constructor：构造器的自动装配和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;

v). 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.

**12)**[**.@Autorwired**](mailto:.@Autorwired)**, @Qualifier, @Required, @Resource的使用  
@Required:**

**@Required** 注释应用于 bean 属性的 setter 方法，它表明受影响的 bean 属性在配置时必须放在 XML 配置文件中，否则容器就会抛出一个 BeanInitializationException 异常。在应用程序的生命周期中，你可能不大愿意花时间在验证所有bean的属性是否按照上下文文件正确配置。或者你宁可验证某个bean的特定属性是否被正确的设置。即使是用“dependency-check”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解.

1. public class EmployeeFactoryBean extends AbstractFactoryBean<Object>{

     private String designation;

1. public String getDesignation() {
2. return designation;
3. }
4. @Required
5. public void setDesignation(String designation) {
6. this.designation = designation;
7. }
8. //more code here
9. }

RequiredAnnotationBeanPostProcessor是Spring中的后置处理用来验证被@Required 注解的bean属性是否被正确的设置了。在使用RequiredAnnotationBeanPostProcesso来验证bean属性之前，首先要在IoC容器中对其进行注册:

**<bean** class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" **/>**

但是如果没有属性被用 @Required 注解过的话，后置处理器会抛出一个BeanInitializationException 异常

**@Autorwired:**

@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。

|  |  |
| --- | --- |
| 10 | public class TestServiceImpl {      // 下面两种@Autowired只要使用一种即可      @Autowired      private UserDao userDao; // 用于字段上        @Autowired      public void setUserDao(UserDao userDao) { // 用于属性的方法上          this.userDao = userDao;  }  } |

@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：@Autowired(required=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 。如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class TestServiceImpl {      @Autowired      @Qualifier("userDao")      private UserDao userDao;  } |

**@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。

|  |  |
| --- | --- |
| 10 | public class TestServiceImpl {      // 下面两种@Resource只要使用一种即可      @Resource(name="userDao")      private UserDao userDao; // 用于字段上        @Resource(name="userDao")      public void setUserDao(UserDao userDao) { // 用于属性的setter方法上          this.userDao = userDao;      }  } |

注：最好是将@Resource放在setter方法上，因为这样更符合面向对象的思想，通过set、get去操作属性，而不是直接去操作属性。

@Resource装配顺序：

①如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。

②如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。

③如果指定了type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。

④如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配。

@Resource的作用相当于@Autowired，只不过@Autowired按照byType自动注入

**@Qualifier:**

@Qualifier注解意味着可以在被标注bean的字段上可以自动装配。Qualifier注解可以用来取消Spring不能取消的bean应用, 下面的示例将会在Customer的person属性中自动装配person的值:

1. **public** **class** Customer{
2. @Autowired
3. **private** Person person;
4. }

Bean.xml配置文件:

1. **<bean** id="customer" class="com.somnus.common.Customer" **/>**
3. **<bean** id="personA" class="com.somnus.common.Person" **>**
4. **<property** name="name" value="lokesh" **/>**
5. **</bean>**
7. **<bean** id="personB" class="com.somnus.common.Person" **>**
8. **<property** name="name" value="alex" **/>**
9. **</bean>**

Spring会知道要自动装配哪个person bean么？不会的，但是运行上面的示例时，会抛出下面的异常:

NoSuchBeanDefinitionException,expected single matching bean but found 2: [personA, personB].

要解决上面的问题，需要使用 @Quanlifier注解来告诉Spring容器要装配哪个bean:

1. **public** **class** Customer{
2. @Autowired
3. @Qualifier("personA")
4. **private** Person person;
5. }

**13).构造方法注入和设值注入有什么区别**

i). 在设值注入方法支持大部分的依赖注入，如果我们仅需要注入int、string和long型的变量，我们不要用设值的方法注入。对于基本类型，如果我们没有注入的话，可以为基本类型设置默认值。在构造方法注入不支持大部分的依赖注入，因为在调用构造方法中必须传入正确的构造参数，否则的话为报错;

ii). 设值注入不会重写构造方法的值。如果我们对同一个变量同时使用了构造方法注入又使用了设置方法注入的话，那么构造方法将不能覆盖由设值方法注入的值。很明显，因为构造方法尽在对象被创建时调用;

iii). 在使用设值注入时有可能还不能保证某种依赖是否已经被注入，也就是说这时对象的依赖关系有可能是不完整的。而在另一种情况下，构造器注入则不允许生成依赖关系不完整的对象;

iv). 在设值注入时如果对象A和对象B互相依赖，在创建对象A时Spring会抛出sObjectCurrentlyInCreationException异常，因为在B对象被创建之前A对象是不能被创建的，反之亦然。所以Spring用设值注入的方法解决了循环依赖的问题，因对象的设值方法是在对象被创建之前被调用的.

**14).Spring中的事件**

i). 上下文更新事件（ContextRefreshedEvent）：该事件会在ApplicationContext被初始化或者更新时发布。也可以在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发;

ii). 上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件;

iii). 上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件;

iv). 上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁;

v). 请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件.

**15).FileSystemResource和ClassPathResource的区别**

在FileSystemResource 中需要给出spring-config.xml文件在你项目中的相对路径或者绝对路径。在ClassPathResource中spring会在ClassPath中自动搜寻配置文件，所以要把ClassPathResource 文件放在ClassPath下, 如果将spring-config.xml保存在了src文件夹下的话，只需给出配置文件的名称即可，因为src文件夹是默认, 简而言之，ClassPathResource在环境变量中读取配置文件，FileSystemResource在配置文件中读取配置文件.

**16).Spring中的设计模式**

i). 代理模式—在AOP和remoting中被用的比较多;

ii). 单例模式—在spring配置文件中定义的bean默认为单例模式;

iii). 模板方法—用来解决代码重复的问题。比如. [RestTemplate](http://howtodoinjava.com/2015/02/20/spring-restful-client-resttemplate-example/), JmsTemplate, JpaTemplate;

iv). 前端控制器—Spring提供了DispatcherServlet来对请求进行分发;

v). 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;;

vi). 依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念;

vii). 工厂模式—BeanFactory用来创建对象的实例.

**17).AOP**

面向切面编程（AOP）提供另外一种角度来思考程序结构，通过这种方式弥补了面向对象编程（OOP）的不足，除了类（classes）以外，AOP提供了切面。切面对关注点进行模块化，例如横切多个类型和对象的事务管理;Spring的一个关键的组件就是AOP框架，可以自由选择是否使用AOP 提供声明式企业服务，特别是为了替代EJB声明式服务。最重要的服务是声明性事务管理，这个服务建立在Spring的抽象事物管理之上。允许用户实现自定义切面，用AOP来完善OOP的使用,可以把Spring AOP看作是对Spring的一种增强.

常见的应用有：权限管理、日志、事务管理等.

**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实现用的是动态代理的方式。

**AOP相关术语:**

i). **切面/方面（Aspect）**：AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在SpringAOP中，切面通过带有@Aspect注解的类实现;

ii). **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;

iii). **通知/增强（Advice）**：在切入点上，可以应用的增强包括：around、before和throws。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中定义了四个advice:BeforeAdvice, AfterAdvice, ThrowAdvice和DynamicIntroductionAdvice;

iv). **切入点（Pointcut）**：将被增强（Advice）应用的连接点的集合（通常是Method集合）。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;

v). **目标对象（TargetObject）**：被通知（Advice）或被代理对象;

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

**Spring AOP的增强/通知(Advice)类型:**

i). **Before Advice：**在方法执行前执行;

ii). **AfterAdvice：**在方法执行之后调用的通知，无论方法执行是否成功;

iii). **After ReturningAdvice：**在方法执行后返回一个结果后执行;

iv). **After ThrowingAdvice：**在方法执行过程中抛出异常的时候执行;

v). **Around Advice：**在方法执行前后和抛出异常时执行，相当于综合了以上三种通知;

vi). **IntroductionAdvice**（引入增强）：引入通知是一种特殊的通知，它能将新的成员变量、成员方法引入到目标类中。它不能作用于任何切入点，因为它只作用于类层次，而不是方法层次。实现引入通知需要实现IntroductionAdvisor和IntroductionInterceptor接口.

**Spring AOP的织入:**

**织入(Weaving)**：把切面（Aspect）应用到目标对象来创建新的代理对象的过程，织入一般发生在如下几个时机:

i). **编译时**：当一个类文件被编译时进行织入，这需要特殊的编译器才可以做的到，例如AspectJ的织入编译器;

ii). **类加载时**：使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码;

iii). **运行时**：切面在运行的某个时刻被织入，SpringAOP就是以这种方式织入切面的，原理应该是使用了动态代理技术.

**Spring AOP的实现:**

与 AspectJ 相同的是，Spring AOP 同样需要对目标类进行增强，也就是生成新的 AOP 代理类；与 AspectJ 不同的是，Spring AOP 无需使用任何特殊命令对 Java 源代码进行编译，它采用运行时动态地、在内存中临时生成“代理类”的方式来生成 AOP 代理, Spring 允许使用 AspectJ Annotation 用于定义方面（Aspect）、切入点（Pointcut）和增强处理（Advice），Spring 框架则可识别并根据这些 Annotation 来生成 AOP 代理。Spring 只是使用了和 AspectJ 5 一样的注解，但并没有使用 AspectJ 的编译器或者织入器（Weaver），底层依然使用的是 Spring AOP，依然是在运行时动态生成 AOP 代理，并不依赖于 AspectJ 的编译器或者织入器, Spring 依然采用运行时生成动态代理的方式来增强目标对象，所以它不需要增加额外的编译，也不需要 AspectJ 的织入器支持；而 AspectJ 在采用编译时增强，所以 AspectJ 需要使用自己的编译器来编译 Java 文件，还需要织入器.

为了启用 Spring 对 @AspectJ 方面配置的支持，并保证 Spring 容器中的目标 Bean 被一个或多个方面自动增强，必须在 Spring 配置文件中配置如下片段:

<beans>

<!-- 启动 @AspectJ 支持 -->

<aop:aspectj-autoproxy/>

</beans>

当启动了 @AspectJ 支持后，只要我们在 Spring 容器中配置一个带 @Aspect 注释的 Bean，Spring 将会自动识别该 Bean，并将该 Bean 作为方面 Bean 处理。

在 Spring 容器中配置方面 Bean（即带 @Aspect 注释的 Bean），与配置普通 Bean 没有任何区别，一样使用 <bean.../> 元素进行配置，一样支持使用依赖注入来配置属性值

使用 @Aspect 标注一个 Java 类，该 Java 类将会作为方面 Bean，如下面代码片段所示：

// 使用 @Aspect 定义一个方面类.

@Aspect

public class LogAspect

{

// 定义该类的其他内容

...

}

方面类（用 @Aspect 修饰的类）和其他类一样可以有方法、属性定义，还可能包括切入点、增强处理定义, 当我们使用 @Aspect 来修饰一个 Java 类之后，Spring 将不会把该 Bean 当成组件 Bean 处理，因此负责自动增强的后处理 Bean 将会略过该 Bean，不会对该 Bean 进行任何增强处理, 开发时无须担心使用 @Aspect 定义的方面类被增强处理，当 Spring 容器检测到某个 Bean 类使用了 @Aspect 标注之后，Spring 容器不会对该 Bean 类进行增强.

**AOP的例子:**

使用一个简单的 Chinese 类来模拟业务逻辑组件:

@Component

public class Chinese

{

// 实现 Person 接口的 sayHello() 方法

    public String sayHello(String name)

    {

   System.out.println("-- 正在执行 sayHello 方法 --");

// 返回简单的字符串

        return name + " Hello , Spring AOP";

    }

// 定义一个 eat() 方法

    public void eat(String food)

    {

   System.out.println("我正在吃 :"+ food);

    }

}

接下来假设同样需要为上面 Chinese 类的每个方法增加事务控制、日志记录，此时可以考虑使用 Around、AfterReturning 两种增强处理:

// 定义一个方面

@Aspect

public class AfterReturningAdviceTest

{

// 匹配 org.crazyit.app.service.impl 包下所有类的、

// 所有方法的执行作为切入点

@AfterReturning(returning="rvt",

pointcut="execution(\* org.crazyit.app.service.impl.\*.\*(..))")

public void log(Object rvt)

{

System.out.println("获取目标方法返回值 :" + rvt);

System.out.println("模拟记录日志功能 ...");

}

}

 Aspect 类使用了 @Aspect 修饰，这样 Spring 会将它当成一个方面 Bean 进行处理。其中将会在调用 org.crazyit.app.service.impl 包下的所有类的所有方法之后织入 log(Object rvt) 方法;

/ 定义一个方面

@Aspect

public class AroundAdviceTest

{

// 匹配 org.crazyit.app.service.impl 包下所有类的、

// 所有方法的执行作为切入点

@Around("execution(\* org.crazyit.app.service.impl.\*.\*(..))")

public Object processTx(ProceedingJoinPoint jp)

throws java.lang.Throwable

{

System.out.println("执行目标方法之前，模拟开始事务 ...");

// 执行目标方法，并保存目标方法执行后的返回值

Object rvt = jp.proceed(new String[]{"被改变的参数"});

System.out.println("执行目标方法之后，模拟结束事务 ...");

return rvt + " 新增的内容";

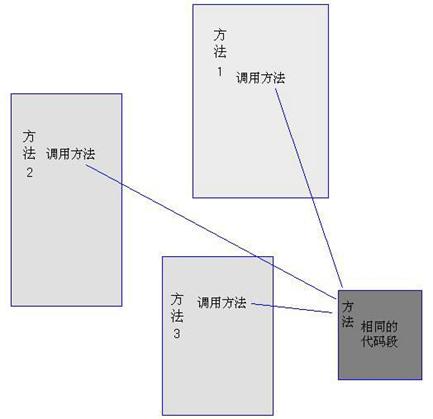
}

}

此处同样使用了 @Aspect 来修饰前面 Bean，其中指定在调用 org.crazyit.app.service.impl 包下的所有类的所有方法的“前后（Around）” 织入 processTx(ProceedingJoinPoint jp) 方法, 需要指出的是，虽然此处只介绍了 Spring AOP 的 AfterReturning、Around 两种增强处理，但实际上 Spring 还支持 Before、After、AfterThrowing 等增强处理.

**Spring AOP的原理剖析:**

AOP 代理其实是由 AOP 框架动态生成的一个对象，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 的 AOP 代理由 Spring 的 IoC 容器负责生成、管理，其依赖关系也由 IoC 容器负责管理。因此，AOP 代理可以直接使用容器中的其他 Bean 实例作为目标，这种关系可由 IoC 容器的依赖注入提供;  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.



上图所示的软件调用结构：当方法 1、方法 2、方法 3 ……都需要去调用某个具有“横切”性质的方法时，传统的做法是程序员去手动修改方法 1、方法 2、方法 3 ……、通过代码来调用这个具有“横切”性质的方法，但这种做法的可扩展性不好，因为每次都要改代码; OP 框架则可以“动态的”生成一个新的代理类，而这个代理类所包含的方法 1、方法 2、方法 3 ……也增加了调用这个具有“横切”性质的方法——但这种调用由 AOP 框架自动生成的代理类来负责，因此具有了极好的扩展性。程序员无需手动修改方法 1、方法 2、方法 3 的代码，程序员只要定义切入点即可—— AOP 框架所生成的 AOP 代理类中包含了新的方法 1、访法 2、方法 3，而 AOP 框架会根据切入点来决定是否要在方法 1、方法 2、方法 3 中回调具有“横切”性质的方法.

**18).Spring如何配置数据库驱动**

使用”org.springframework.jdbc.datasource.DriverManagerDataSource”数据源来配置数据库驱动。示例如下：

<bean id=”dataSource”>

    <property name=”driverClassName”>

        <value>org.mysql.jdbcDriver</value>

    </property>

    <property name=”url”>

        <value>jdbc: mysql:db/appfuse</value>

    </property>

    <property name=”username”><value>abc</value></property>

    <property name=”password”><value>abc</value></property>

</bean>

**19).Spring中的applicationContext能不能改文件名**

ContextLoaderListener是一个ServletContextListener, 它在你的web应用启动的时候初始化。缺省情况下， 它会在WEB-INF/applicationContext.xml文件找Spring的配置。 你可以通过定义一个<context-param>元素名字为”contextConfigLocation”来改变Spring配置文件的 位置。示例如下：

<listener>

    <listener-class>org.springframework.web.context.ContextLoaderListener

        <context-param>

         <param-name>contextConfigLocation</param-name>

         <param-value>/WEB-INF/xyz.xml</param-value>

        </context-param>

    </listener-class>

</listener>

**20).Spring中定义hibernate mapping文件**

添加hibernate mapping 文件到web/WEB-INF目录下的applicationContext.xml文件里面。示例如下：

<property name=”mappingResources”>

    <list>

        <value>org/appfuse/model/User.hbm.xml</value>

    </list>

</property>

**21).Spring如何处理并发问题**

i).在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean(如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等)中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了;

ii). ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题, 在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大, 而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal;

iii). 对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响.

**22).Spring中的事物管理**

事务就是对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。这样可以防止出现脏数据，防止数据库数据出现问题, 开发中为了避免这种情况一般都会进行事务管理。Spring中也有自己的事务管理机制，一般是使用TransactionMananger进行管 理，可以通过Spring的注入来完成此功能。spring提供了几个关于事务处理的类:

TransactionDefinition //事务属性定义;

TranscationStatus //代表了当前的事务，可以提交，回滚;

PlatformTransactionManager这个是spring提供的用于管理事务的基础接口，其下有一个实现的抽象类 AbstractPlatformTransactionManager,我们使用的事务管理类例如 DataSourceTransactionManager等都是这个类的子类.

一般事务定义步骤:

TransactionDefinition td =newTransactionDefinition();

TransactionStatus ts = transactionManager.getTransaction(td);

try{

    //do sth

    transactionManager.commit(ts);

}catch(Exception e){

    transactionManager.rollback(ts);

}

spring提供的事务管理可以分为两类：编程式的和声明式的。编程式的，比较灵活，但是代码量大，存在重复的代码比较多；声明式的比编程式的更灵活.

编程式主要使用transactionTemplate。省略了部分的提交，回滚，一系列的事务对象定义，需注入事务管理对象:

void add(){

    transactionTemplate.execute(newTransactionCallback(){

        pulic Object doInTransaction(TransactionStatus ts){

         //do sth

        }

    }

}

声明式：使用TransactionProxyFactoryBean:PROPAGATION\_REQUIRED PROPAGATION\_REQUIRED PROPAGATION\_REQUIRED,readOnly

围绕Poxy的动态代理 能够自动的提交和回滚事务

围绕Poxy的动态代理 能够自动的提交和回滚事务

org.springframework.transaction.interceptor.TransactionProxyFactoryBean

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。

###### Mybatis

**1).JDBC有哪些不足之处,Mybatis是如何解决这些问题的**

① 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接

② Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离

③ 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决： Mybatis自动将java对象映射至sql语句

④ 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象.

**2).Mybatis编程步骤**

① 创建SqlSessionFactory   
② 通过SqlSessionFactory创建SqlSession   
③ 通过sqlsession执行数据库操作   
④ 调用session.commit()提交事务   
⑤ 调用session.close()关闭会话

**3).Mybatis与Hibernate的区别**

i).Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句，不过mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象;

ii).Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大;

iii).Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用hibernate开发可以节省很多代码，提高效率。但是Hibernate的缺点是学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行.

**4).使用Mybatis的mapper接口调用有哪些要求**

①  Mapper接口方法名和mapper.xml中定义的每个sql的id相同   
②  Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同   
③  Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同   
④  Mapper.xml文件中的namespace即是mapper接口的类路径。

**5).SqlMapConfig.xml中配置有哪些内容**

SqlMapConfig.xml中配置的内容和顺序如下：

properties（属性）

settings（配置）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

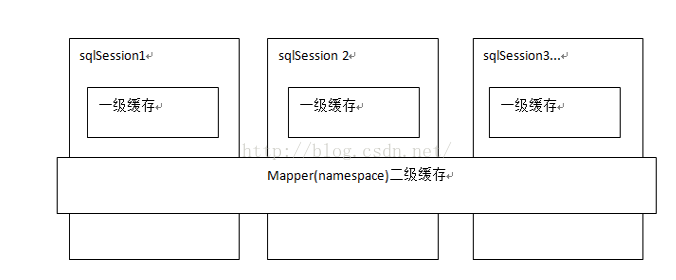
dataSource（数据源）

mappers（映射器）

**6).Mybatis的一级缓存与二级缓存**

mybatis提供查询缓存，用于减轻数据压力，提高数据库性能。如果缓存中有数据就不用从数据库中获取，大大提高系统性能.

mybaits提供一级缓存，和二级缓存.



**一级缓存**是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个(内存区域)数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的; 一级缓存的作用域是同一个SqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis默认开启一级缓存.

**二级缓存**是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession去操作数据库得到数据会存在二级缓存区域，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的,  二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存.

(在核心配置文件SqlMapConfig.xml中加入<setting name="cacheEnabled"value="true"/>)

**7). #{}和${}的区别是什么**

#{}是预编译处理，${}是字符串替换。

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止SQL注入，提高系统安全性。

**8).mapper的编写方式以及工作原理**

**①接口实现类继承SqlSessionDaoSupport:** 使用此种方法需要编写mapper接口，mapper接口实现类、mapper.xml文件.主要包括以下步骤:

a).在sqlMapConfig.xml中配置mapper.xml的位置

<mappers>

    <mapper resource="mapper.xml文件的地址" />

    <mapper resource="mapper.xml文件的地址" />

</mappers>

b). 定义mapper接口;

c). 实现类继承SqlSessionDaoSupport, mapper方法中可以this.getSqlSession()进行数据增删改查

d). spring 配置

<bean id=" " class="mapper接口的实现">

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

</bean>

**②使用org.mybatis.spring.mapper.MapperFactoryBean**:主要步骤如下

**a).** 在sqlMapConfig.xml中配置mapper.xml的位置:

<mappers>

    <mapper resource="mapper.xml文件的地址" />

    <mapper resource="mapper.xml文件的地址" />

</mappers>

**b).** 定义mapper接口

**③使用mapper扫描器:**

a). mapper.xml文件编写: mapper.xml中的namespace为mapper接口的地址; mapper接口中的方法名和mapper.xml中的定义的statement的id保持一致;

b). 定义mapper接口: mapper.xml的文件名和mapper的接口名称保持一致，且放在同一个目录;

**c).** 配置mapper扫描器:

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

    <property name="basePackage" value="mapper接口包地址"></property>

    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>

</bean>

d). 使用扫描器后从spring容器中获取mapper的实现对象, 扫描器将接口通过代理方法生成实现对象，要spring容器中自动注册，名称为mapper 接口的名称.

**Mapper的工作原理:** mybatis 在运行时使用 JDK 动态代理生成代理对象，在调用接口方法时，代理对象会拦截接口方法，转而运行 MappedStatement 的 sql 语句，并生成结果返回.

**9). MyBatis里面的动态Sql是怎么设定的?用什么语法?**

i).MyBatis里面的动态Sql一般是通过if节点来实现,通过OGNL语法来实现,但是如果要写的完整,必须配合where,trim节点,where节点是判断包含节点有内容就插入where,否则不插入,trim节点是用来判断如果动态语句是以and 或or开始,那么会自动把这个and或者or去掉.

ii).Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态拼接sql的功能。

Mybatis提供了9种动态sql标签：trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能

**10). IBatis和MyBatis在核心处理类分别叫什么以及Ibatis和MyBatis的区别**

IBatis里面的核心处理类交SqlMapClient,

MyBatis里面的核心处理类叫做SqlSession

区别:

i). 在sql里面变量命名有原来的#变量# 变成了#{变量};

ii). 原来的$变量$变成了${变量};

iii). 原来在sql节点里面的class都换名字交type;

iv). 原来的queryForObject queryForList 变成了selectOne selectList;

v). 原来的别名设置在映射文件里面放在了核心配置文件里

**11).Mybatis标签**

常用标签:select、insert、update、delete;

动态 sql 标签：trim、where、set、foreach、if、choose、when、otherwise、bind

**12).Mybatis如何确定唯一的操作**

mybatis 通过 namespace 和 MappedStatement 的 id 值确定唯一的 MappedStatement.

**13).mybatis 的 mapper 接口中的接口方法可以通过不同参数的方式对方法进行重载吗**

不可以。在 mybatis 在执行方法前是通过 namespace+id 进行 MappedStatement 的确定，跟参数没有关系，所以不能实现重载;

**14).Mybatis是如何进行分页的**

mybatis 是通过 RowBounds 对象进行分页的，他针对返回结果集 ResultSet 进行内存分页，而非物理分页, 分页插件通过 mybatis 提供的分页插件接口进行实现，通过拦截器拦截需要执行的 sql 并重写 sql，以此来添加对应的参数，最终实现分页效果.

**15).Mybatis延迟加载以及原理**

mybatis 支持延迟加载，但有限制。他支持一对一、一对多的延迟加载; 要使 mybatis 支持延迟加载，需要在配置文件中配置 lazyLoadingEnabled 值为 true。mybatis 通过 CGLIB 创建目标对象的代理对象，在调用目标对象的 get 方法时，进行拦截并检查关联对象是否为空，如果为空则调用 sql 进行相应对象的查询并通过 set 方法进行数据填充，最后在返回对应关联对象

**16).Mybatis中的执行器**

在 mybatis 中有三种基本的执行器，他们分别是：SimpleExecutor、ReuseExecutor、BatchExecutor。他们区别如下:

i).SimpleExecutor：每执行一次 update 或 select 都开启一个 Statement 对象，执行完后关闭该对象

ii). ReuseExecutor: 执行 update 或 select 时，以 sql 作为 key 进行缓存查找 Statement，如果找到直接执行，如果没找到则创建并缓存，执行完操作后不关闭该对象，以备下次执行时使用;

iii). BatchExecutor: 在执行 update 时，所有的 sql 都将添加到批处理器中进行统一处理。他缓存了多个 Statement 对象，所有的对象都等待着屁处理器统一执行.

**17). 当实体类中的属性名和表中的字段名不一样 ，怎么处理**

i). 通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致:

<select id=”selectorder” parametertype=”int” resultetype=”me.gacl.domain.order”>

select order\_id id, order\_no orderno ,order\_price price form orders where order\_id=#{id};

</select>

ii). 通过<resultMap>来映射字段名和实体类属性名的一一对应的关系:

<select id="getOrder" parameterType="int" resultMap="orderresultmap">

select \* from orders where order\_id=#{id}

</select>

<resultMap type=”me.gacl.domain.order” id=”orderresultmap”>

<!–用id属性来映射主键字段–>

<id property=”id” column=”order\_id”>

<!–用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性–>

<result property = “orderno” column =”order\_no”/>

<result property=”price” column=”order\_price” />

</reslutMap>

**18). Like的使用方法**

i). 在Java代码中添加sql通配符:

string wildcardname = “%smi%”;

list<name> names = mapper.selectlike(wildcardname);

<select id=”selectlike”>

select \* from foo where bar like #{value}

</select>

ii). 在sql语句中拼接通配符，会引起sql注入:

string wildcardname = “smi”;

list<name> names = mapper.selectlike(wildcardname);

<select id=”selectlike”>

select \* from foo where bar like "%"#{value}"%"

</select>

**19).mapper传递多个参数**

i).通过参数顺序:

//DAO层的函数

Public UserselectUser(String name,String area);

//对应的xml,#{0}代表接收的是dao层中的第一个参数，#{1}代表dao层中第二参数，更多参数一致往后加即可。

<select id="selectUser"resultMap="BaseResultMap">

select \* fromuser\_user\_t whereuser\_name = #{0} anduser\_area=#{1}

</select>

ii). 使用 @param 注解:

public interface usermapper {

user selectuser(@param(“username”) string username,

@param(“hashedpassword”) string hashedpassword);

}

然后,就可以在xml像下面这样使用:

<select id=”selectuser” resulttype=”user”>

select id, username, hashedpassword

from some\_table

where username = #{username}

and hashedpassword = #{hashedpassword}

</select>

###### SpringMVC

**1).SpringMVC的工作流程**

i).用户发送请求至前端控制器DispatcherServlet

ii). 处理器映射器找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet;

iii). DispatcherServlet调用HandlerAdapter处理器适配器;

iv). HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器);

v). Controller执行完成返回ModelAndView, HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet;

vi). DispatcherServlet将ModelAndView传给ViewReslover视图解析器, ViewReslover解析后返回具体View;

vii). DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）, DispatcherServlet响应用户;

**2).如何解决POST和GET请求的中文乱码问题**

对于POST请求:

在web.xml中加入：

<filter>

    <filter-name>CharacterEncodingFilter</filter-name>

    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

    <init-param>

        <param-name>encoding</param-name>

        <param-value>utf-8</param-value>

    </init-param>

</filter>

<filter-mapping>

    <filter-name>CharacterEncodingFilter</filter-name>

    <url-pattern>/\*</url-pattern>

</filter-mapping>

对于GET请求:

修改tomcat配置文件添加编码与工程编码一致，如下：

<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

另外一种方法对参数进行重新编码：

String userName = newString(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

**3).SpringMVC与Struts2的区别**

i). springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过虑器;

ii). springmvc是基于方法开发，传递参数是通过方法形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例;

iii). Struts采用值栈存储请求和响应的数据，通过OGNL存取数据， springmvc通过参数解析器是将request对象内容进行解析成方法形参，将响应数据和页面封装成ModelAndView对象，最后又将模型数据通过request对象传输到页面。 Jsp视图解析器默认使用jstl.

**4).SpringMVC和Struts2的核心入口是什么**

SpringMvc的是DispatchServlet, Struts2的是StrutsPrepareAndExecuteFilter.

**5).SpringMVC的控制器是否是单例模式**

是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段.

**6).@RequestMapping注解用在类上面有什么作用**

用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径.

**7).如果在拦截请求中,我想拦截get方式提交的方法,怎么配置**

可以在@RequestMapping注解里面加上method=RequestMethod.GET.

**8). 在拦截的方法里面得到从前台传入的参数,怎么得到**

直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样

**9).如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象**

直接在方法中声明这个对象,SpringMvc就自动会把属性赋值到这个对象里面

**10).SpringMVC的返回值**

返回值可以有很多类型,有String, ModelAndView,一般用String比较好;

处理返回值: SpringMvc根据配置文件中InternalResourceViewResolver的前缀和后缀,用前缀+返回值+后缀组成完整的返回值;

设置重定向和转发:在返回值前加上” forward:”进行转发; 在返回值前加上"redirect:"就可以让返回值重定向,譬如"redirect:http://www.baidu.com".

**11).SpringMVC数据传输**

i). 通过ModelMap对象把数据从后台向前台传递,可以在这个对象里面用put方法,把对象加到里面,前台就可以通过el表达式拿到;

ii).ModelAndView把视图和数据合并到一起;

iii). 把ModelMap里面的数据放入Session, 可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key.

**12).SpringMVC与Ajax相互调用**

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象

  具体步骤如下

  1.加入Jackson.jar

  2.在配置文件中配置json的映射

  3.在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解

**13).SpringMVC实现拦截器**

有两种写法,一种是实现接口,另外一种是继承适配器类,然后在SpringMvc的配置文件中配置拦截器即可:

  <!-- 配置SpringMvc的拦截器 -->

<mvc:interceptors>

    <!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->

    <bean id="myInterceptor" class="com.et.action.MyHandlerInterceptor"></bean>

    <!-- 只针对部分请求拦截 -->

    <mvc:interceptor>

       <mvc:mapping path="/modelMap.do" />

       <bean class="com.et.action.MyHandlerInterceptorAdapter" />

    </mvc:interceptor>

</mvc:interceptors>

**14).Model,ModelMap,ModelAndView的区别**

**Model** 是一个接口， 其实现类为ExtendedModelMap，继承了ModelMap类;

ModelMap对象主要用于传递控制方法处理数据到结果页面，把结果页面上需要的数据放到ModelMap对象中即可，他的作用类似于request对象的setAttribute方法的作用，用来在一个请求过程中传递处理的数据。通过以下方法向页面传递参数：  
addAttribute(String key,Object value);  
在页面上可以通过el变量方式$key或者通过其他数据展示标签获取并展示ModelMap中的数据。Mapdelmap本身不能设置页面跳转的url地址别名或者物理跳转地址，那么我们可以通过控制器方法的返回值来设置跳转url地址别名或者物理跳转地址.

ModelAndView对象有两个作用：  
i).设置转向地址,如下所示（这也是ModelAndView和ModelMap的主要区别）  
ModelAndView view = new ModelAndView("path:ok");

ii).用于传递控制方法处理结果数据到结果页面，也就是说我们把需要在结果页面上需要的数据放到ModelAndView对象中即可，他的作用类似于request对象的setAttribute方法的作用，用来在一个请求过程中传递处理的数据。通过以下方法向页面传递参数：  
addObject(String key,Object value);

在页面上可以通过el变量方式$key或者其他数据展示标签获取并展示ModelAndView中的数据。

###### **4.Struts2**

**1).Struts2的工作流程**

i). 请求发送给 StrutsPrepareAndExecuteFilter;

ii). StrutsPrepareAndExecuteFilter 判定该请求是否是一个 Struts2 请  
求;

iii). 若该请求是一个 Struts2 请求，则 StrutsPrepareAndExecuteFilter  
把请求的处理交给 ActionProxy;

iv). ActionProxy 创建一个 ActionInvocation 的实例，并进行初始化;

v). ActionInvocation 实例在调用 Action 的过程前后，涉及到相关拦截  
器（Intercepter）的调用;

vi). Action 执行完毕，ActionInvocation 负责根据 struts.xml 中的配置  
找到对应的返回结果。调用结果的 execute 方法，渲染结果；

vii).执行各个拦截器 invocation.invoke() 之后的代码；

viii). 把结果发送到客户端.

**2).Struts2中拦截器和过滤器的区别**

i). 过滤器依赖于 Servlet 容器，而拦截器不依赖于 Servlet 容器;

ii). Struts2 拦截器只能对 Action 请求起作用，而过滤器则可以对几乎所  
有请求起作用;

iii). 拦截器可以访问 Action 上下文(ActionContext)、值栈里的对象  
(ValueStack)，而过滤器不能;  
iv). 在 Action 的生命周期中，拦截器可以多次调用，而过滤器只能在容器  
初始化时被调用一次

**3).Struts2的优点**

i)基于 MVC 架构，框架结构清晰;

ii).使用 OGNL: OGNL 可以快捷的访问值栈中的数据、调用值栈中对象的方  
法;

iii). 拦截器: Struts2 的拦截器是一个 Action 级别的 AOP, Struts2 中的  
许多特性都是通过拦截器来实现的, 例如异常处理，文件上传，验证等。拦截器  
是可配置与重用的;

iv). 多种表现层技术. 如：JSP、FreeMarker、Velocity 等.

**4).Struts2如何访问HttpServletRequest,HttpSession,ServletContext三个对象域**

与 Servlet API 解耦的访问方式:

i). 通过 ActionContext 访问域对象对应的 Map 对象;

ii). 通过实现 Aware 接口使 Struts2 注入对应的 Map 对象;

与 Servlet API 耦合的访问方式:

i). 通过 ServletActionContext 直接获取 Servlet API 对象;

ii). 通过实现 ServletXxxAware 接口的方式使 Struts2 注入对应的对象.

**5). Struts2 中的默认包 struts-default 有什么作用**

①. struts-default 包是 struts2 内置的，它定义了 struts2 内部的众  
多拦截器和 Result 类型，而 Struts2 很多核心的功能都是通过这些内置的拦  
截器实现，如：从请求中把请求参数封装到 action、文件上传和数据验证等等  
都是通过拦截器实现的。当包继承了 struts-default 包才能使用 struts2 为我  
们提供的这些功能。  
② .struts-default 包 是 在 struts-default.xml 中 定 义 ，  
struts-default.xml 也是 Struts2 默认配置文件。 Struts2 每次都会自动加  
载 struts-default.xml 文件。  
③. 通常每个包都应该继承 struts-default 包

**6).Struts2的拦截器**

Exception:异常拦截器;

fileUpload: 提供文件上传功能;

i18n: 记录用户选择的locale;

cookies :使用配置的name,value来是指cookies   
chain :让前一个Action的属性可以被后一个Action访问，现在和chain类型的result（）结合使用

**7).Struts2的值栈(ValueStack)**

①. ValueStack 贯穿整个 Action 的生命周期，保存在 request 域中，所  
以 ValueStack 和 request 的生命周期一样. 当 Struts2 接受一个请求时，会  
迅 速 创 建 ActionContext ， ValueStack ， Action. 然 后 把 Action 存 放 进  
ValueStack，所以 Action 的实例变量可以被 OGNL 访问。 请求来的时候，  
Action、ValueStack 的生命开始；请求结束，Action、ValueStack 的生命结束  
②. 值栈是多实例的，因为 Action 是多例的(和 Servlet 不一样，Servelt  
是单例的)，而每个 Action 都有一个对应的值栈，Action 对象默认保存在栈顶；  
③. ValueStack 本质上就是一个 ArrayList(查看源代码得到)；  
④. 使用 OGNL 访问值栈的内容时，不需要#号，而访问 request、session、  
application、attr 时，需要加#号；  
⑤. Struts2 重写了 request 的 getAttribute 方法，所以可以使用 EL  
直接访问值栈中的内容.

**8) . ActionContext、ServletContext、pageContext 的区别**

①. ActionContext Struts2 的 API：是当前的 Action 的上下文环境  
②. ServletContext 和 PageContext 是 Servlet 的 API

**9). 拦截器的生命周期与工作过程**

每个拦截器都是需要实现 Interceptor 接口  
> init()：在拦截器被创建后立即被调用, 它在拦截器的生命周期内只  
被调用一次. 可以在该方法中对相关资源进行必要的初始化；  
> intercept(ActionInvocation invocation)：每拦截一个动作请求，  
该方法就会被调用一次；  
> destroy：该方法将在拦截器被销毁之前被调用, 它在拦截器的生命周  
期内也只被调用一次;

**10).Struts2如何防止表单重复提交**

第一步：在表单中加入<s:token />

第二步，使用token栏截器，定义invalid.token结果集

**11).Struts2与SpringMVC的区别**

i).  机制：spring mvc的入口是servlet，而struts2是filter二者的机制不同(Servlet与Filter的区别);

ii). 性能：spring会稍微比struts2快。spring mvc是基于方法的设计，而sturts是基于类，每次发一次请求都会实例一个action，每个action都会被注入属性，而spring基于方法，粒度更细，但要小心把握像在servlet控制数据一样。spring mvc是方法级别的拦截，拦截到方法后根据参数上的注解，把request数据注入进去，在spring3 mvc中，一个方法对应一个request上下文。而struts2框架是类级别的拦截，每次来了请求就创建一个Action，然后调用setter getter方法把request中的数据注入；struts2实际上是通过setter getter方法与request打交道的；struts2中，一个Action对象对应一个request上下文;

iii). 参数传递：struts2是在接受参数的时候，可以用属性来接受参数，这就说明参数是让多个方法共享的;

iv). 设计思想上：struts2更加符合oop的编程思想， spring就比较谨慎，在servlet上扩展;

v). intercepter的实现机制：struts2有以自己的interceptor机制，spring mvc用的是独立的AOP方式。这样导致struts的配置文件量还是比spring mvc大，虽然struts2的配置能继承，所以使用上来讲，springmvc使用更加简洁，开发效率SpringMVC确实比struts2高。springmvc是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应，所以说从架构本身上springmvc就容易实现restful url。struts2是类级别的拦截，一个类对应一个request上下文；实现restful url要费劲，因为struts2 action的一个方法可以对应一个url；而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。springmvc的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架方法之间不共享变量，而struts2搞的就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码，读程序时带来麻烦

vi). springmvc的验证支持JSR303(JSR303是JAVA EE6中的子规范。用于对Java Bean的字段值进行校验，确保输入进来的数据在语义上是正确的，使验证逻辑从业务代码中脱离出来。JSR303是运行时数据验证框架，验证之后验证的错误信息会马上返回。有两个版本JSR303(BeanValidation1.0)和JSR349(BeanValidation1.1))，处理ajax的请求更是方便，只需一个注解@ResponseBody ，然后直接返回响应文本即可

vii). Spring MVC强大的注解，可以认为已经近100%零配置

###### **5.Hibernate**

**1).简述 Hibernate 和 JDBC 的区别、优缺点**

JDBC与Hibernate在性能上相比，JDBC灵活性有优势。而Hibernate在易学性，易用性上有些优势。当用到很多复杂的多表联查和复杂的数据库操作时，JDBC有优势。

**相同点：**◆两者都是JAVA的数据库操作中间件。

◆两者对于数据库进行直接操作的对象都不是线程安全的，都需要及时关闭。

◆两者都可以对数据库的更新操作进行显式的事务处理。

**不同点：**◆使用的SQL语言不同：JDBC使用的是基于关系型数据库的标准SQL语言，Hibernate使用的是HQL(Hibernate query language)语言.

◆操作的对象不同：JDBC操作的是数据，将数据通过SQL语句直接传送到数据库中执行，Hibernate操作的是持久化对象，由底层持久化对象的数据更新到数据库中。

◆数据状态不同：JDBC操作的数据是“瞬时”的，变量的值无法与数据库中的值保持一致，而Hibernate操作的数据是可持久的，即持久化对象的数据属性的值是可以跟数据库中的值保持一致的。

**JDBC与Hibernate读取性能**

i). JDBC仍然是最快的访问方式，不论是Create还是Read操作，都是JDBC快;

ii). Hibernate使用uuid.hex构造主键，性能稍微有点损失，但是不大;

iii). Create操作，JDBC在使用批处理的方式下速度比Hibernate快，使用批处理方式耗用JVM内存比不使用批处理方式要多得多;

iv). 读取数据，Hibernate的Iterator速度非常缓慢，因为他是每次next的时候才去数据库取数据，这一点从观察任务管理器的java进程占用内存的变化也可以看得很清楚，内存是几十K几十K的增加;

v). 读取数据，Hibernate的List速度很快，因为他是一次性把数据取完，这一点从观察任务管理器的java进程占用内存的变化也可以看得很清楚，内存几乎是10M的10M的增加;

vi). JDBC读取数据的方式和Hibernate的List方式是一样的（这跟JDBC驱动有很大关系，不同的JDBC驱动，结果会很不一样），这 从观察java进程内存变化可以判断出来，由于JDBC不需要像Hibernate那样构造一堆对象实例，所以占用JVM内存要比 Hibernate的List方式大概少一半左右;

vii). Hibernate的Iterator方式并非一无是处，它适合于从大的结果集中选取少量的数据，即不需要占用很多内存，又可以迅速得到结果。另外Iterator适合于使用JCS缓冲;

实际上，不管CMP，Hibernate，JDO等等，所有的ORM都是对JDBC的封装，CMP则是一个重量级封装，JDO中度封 装，Hibernate是轻量级的封装。从理论上来说，ORM永远也不可能比JDBC性能好。就像任何高级语言的运行性能永远也不会好过汇编语言一个道 理.

**2).Hibernate三种状态的区分**

对于Hibernate，它的对象有三种状态，transient、persistent、detached

◆transient：瞬态或者自由态(new Data("没关联到Session前")，该po的实例和session没有关联，该po的实例处于transient)  
◆persistent：持久化状态(和数据库中记录相射的Po实例，即数据库表中有条记录是对应于这个对象的。它的状态是persistent, 通过get和load等得到的对象都是persistent)  
◆detached：脱管状态或者游离态  
i)当通过get或load方法得到的po对象它们都处于persistent,但如果执行delete(po)时(但还没执行事务),该 po状态就处于detached, (表示和session脱离关联),因delete而变成游离态可以通过save或saveOrUpdate()变成持久态  
ii)当一个session执行close()或clear()、evict()之后，session缓存中的persistent的po对象也变成detached，此时持久对象会变成脱管对象，此时该对象虽然具有数据库识别值，但它已不在HIbernate持久层的管理之下。

**3). Hibernate中的update()和saveOrUpdate()的区别**

i)update() 更新，没有主键会报错，saveOrUpdate() 保存或更新, 没有主键就执行插入。  
ii)Update:是对暂态(transient )或是只是脱管(detached)的更新操作，对于暂态对象的更新操作通常不产生效果，对于脱管对象是做了同步的操作，即数据库的数据发生变化并且对象状态也成为托管对象;  
 SaveOrUpdate : 也是对暂态(transient )或是只是脱管(detached)的进行操作，至于是插入还是更新，则要根据(identifier)id 中指定的一些具体条件来分析，如果对象没有持久化标识(identifier)属性，对其调用save() ，否则update() 这个对象 。  
iii)如果该po对象已经在本session中持久化了，在本session中执行saveOrUpdate（po）不做任何事;  
如果savaOrUpdate(给定id的新po)与另一个与本session关联的po对象拥有相同的持久化标识(identifier)，抛出一个NonUniqueObjectException异常 ：a different object with the same identifier value was already associated with the session。

**4).Hibernate update和flush区别**

Hibernate update操作的是在自由态或脱管状态(因session的关闭而处于脱管状态)的对象，而flush是操作的在持久状态的对象;

默认情况下，一个持久状态的对象的改动(包含set容器)是不需要update的，只要你更改了对象的值，等待Hibernate flush就自动更新或保存到数据库了。  
i) 调用某些查询的和手动flush(),session的关闭、SessionFactory关闭结合。get()一个对象，把对象的属性进行改变,把资源关闭。  
ii)transaction commit的时候(包含了flush)

**5).Hibernate session的load()和get()的区别**

i). 如果使用load方法，hibernate认为该id对应的对象（数据库记录）在数据库中是一定存在的，所以它可以放心的使用，它可以放心的使用代理来延迟加载该对象。在用到对象中的其他属性数据时才查询数据库，但是万一数据库中不存在该记录，那没办法，只能抛异常，所说的load方法抛异常是指在使用该对象的数据时，数据库中不存在该数据时抛异常，而不是在创建这个对象时。由于session中的缓存对于hibernate来说是个相当廉价的资源，所以在load时会先查一下session缓存看看该id对应的对象是否存在，不存在则创建代理。所以如果你知道该id在数据库中一定有对应记录存在就可以使用load方法来实现延迟加载。 对于get方法，hibernate会确认一下该id对应的数据是否存在，首先在session缓存中查找，然后在二级缓存中查找，还没有就查数据库，数据库中没有就返回null;

ii) . get方法如果在session缓存中找到了该id对应的对象，如果刚好该对象前面是被代理过的，如被load方法使用过，或者被其他关联对象延迟加载过，那么返回的还是原先的代理对象，而不是实体类对象，如果该代理对象还没有加载实体数据（就是id以外的其他属性数据），那么它会查询二级缓存或者数据库来加载数据，但是返回的还是代理对象，只不过已经加载了实体数据;

iii) . get方法首先查询session缓存，没有的话查询二级缓存，最后查询数据库；反而load方法创建时首先查询session缓存，没有就创建代理，实际使用数据时才查询二级缓存和数据库;

hibernate对于load方法认为该数据在数据库中一定存在，可以放心的使用代理来延迟加载，如果在使用过程中发现了问题，只能抛异常；而对于get方法，hibernate一定要获取到真实的数据，否则返回null.

**6).Hibernate List和Iterator方式的比较**

i)、List方式是1次性把所有的数据全部取到内存中，构造一个超大的结果集，主要的时间开销是这一步，这一步的时间开销要远远超过JDBC和 Iterator方式下构造结果集的时间开销，并且内存开销也很惊人；而对结果集的遍历操作，速度则是非常的惊人（经过测试，30万记录的内 存遍历不到100ms，由于这一步不受JDBC影响，因此结果可信）。因此，List方式适合于对结果集进行反复多次操作的情况，例如分页显示，往后往前 遍历，跳到第一行，跳到最后一行等等。

ii)、Iterator方式只取记录id到内存中，并没有把所有数据取到内存中，因此构造结果集的时间开销很小，比JDBC和List方式都要少，并且内 存开销也小很多。而对结果集的遍历的操作的时候，Iterator仍然要访问数据库，所有主要的时间开销都花在这里。因此，Iterator方式适合于只 对结果集进行1次遍历操作的情况，并且Iterator方式特别适合于从超大结果集中取少量数据，这种情况Iterator性能非常好。

**7). Hibernate的inverse属性的作用**

inverse是反转的意思，在关联关系中，inverse="false"为主控方，由主控方负责维护对象的关联关系。  
inverse 决定是否把对对象中集合的改动反映到数据库中，所以inverse只对集合起作用，也就是只对one-to-many或many-to-many有效（因 为只有这两种关联关系包含集合，而one-to-one和many-to-one只含有关系对方的一个引用，注意一般只在双向关联时才有需要设置inverse）。  
i）、一对多：  
通常会在的one一方放弃对多的关系的维护，这样效率会高起来（如老师记住每位学生是件困难的事情，效率是很低的，所以干脆就不记了，这关系由学生来维护，学生记住一位老师是很容易）  
 所以应该在一方的设置 inverse=true ，多的一方设置 inverse=false（多的一方也可以不设置inverse属性，因为默认值是false），这说明关联关系由多的一方来维护。  
如果要一方维护关系，就会使在插入或是删除"一"方时去update"多"方的每一个与这个"一"的对象有关系的对象。  
而如果让"多"方面维护关系时就不会有update操作，因为关系就是在多方的对象中的，直指插入或是删除多方对象就行了。  
显然这样做的话，会减少很多操作，提高了效率。  
注：单向one-to-many关联关系中，不可以设置inverse="true",因为被控方的映射文件中没有主控方的信息。  
ii）、多对多： 属性在独立表中。inverse属性的默认值为false。在多对多关联关系中，关系的两端 inverse不能都设为false,即默认的情况是不对的，如果都设为false,在做插入操作时会导致在关系表中插入两次关系。也不能都设为 true，如果都设为true,任何操作都不会触发对关系表的操作。因此在任意一方设置inverse=true，另一方inverse=false。

**8). Hibernate的cascade属性的作用**

cascade属性的作用是描述关联对象进行操作时的级联特性。因此，只有涉及到关系的元素才有cascade属性。  
具 有cascade属性的标记包括<many-to-one /> <one-to-one /> <any /> <set /><bag /> <idbag /> <list /> <array />  
注意：<one-to-many />和 <many-to-many />是用在集合标记内部的，所以是不需要cascade属性的。  
级联操作：指当主控方执行某项操作时，是否要对被关联方也执行相同的操作

**9).各种关联关系下的lazy懒加载区别**

i)、one-to-one懒加载  
 一对一的懒加载并不常用，因为懒加载的目的是为了减少与数据库的交互，从而提高执行效率，而在一对一关系中，主表中的每一条数据只对应从表的一条数据库，就算都查询也不会增加多少交互的成本，而且主表不能有contrained=true，所以主表是不能懒加载的。但是从表可以有。  
    实现此种懒加载必须在从对象这边同时满足三个条件：  
    lazy!=false（lazy的属性有三个选项分别为：no-proxy、false和proxy）  
    Constrained = true ；  
    fetch=select。  
 注：当fetch设置为join时，懒加载就会失效。因为fetch的作用是抓取方式，他有两个值分别为select和join，默认值为select。即在设为join时，他会直接将从表信息以join方式查询到而不是再次使用select查询，这样导致了懒加载的失效。  
ii)、one-to-many懒加载  
 与one-to-one关联不同，对one-to-many而言，主表的每一条属性都会对应从表的多条数据，这个时候懒加载就显得非常有效了。比如一个部门里面有多个员工，如果没有懒加载，每查询这个部门的时候都会查询出多个员工，这会大大增加与数据库交互的成本。所以Hbernate默认的是加入懒加载的。这就是查询集合属性的时候返回的是一个PersistentIndexed\*类型对象的原因。该对象其实就是一个代理对象。当然，可以在映射文件中通过将lazy属性设为假来禁用。  
 Hibernate默认对one-to-many就是使用的懒加载，但用户也可以取消懒加载操作：  
     设置lazy=”false”;  
     设置fetch=”join”.  
实现此种懒加载必须在从对象这边同时满足两个条件：  
   lazy!=false（lazy的属性有三个选项分别为：no-proxy、false和proxy）  
   fetch=select。  
iii)、many-to-one懒加载  
此关联关系的懒加载和one-to-one的懒加载一样都是可要可不要的，因为对执行效率的提高都不是非常明显。虽然多对一与一对一关系方式相同，但是在Hibernate中多对一时，默认是进行懒加载的。另外有一点需要注意的是懒加载并不会区分集合属性里面是否有值，即使是没有值，他依然会使用懒加载。  
实现此种懒加载必须在从对象这边同时满足两个条件  
   lazy!=false（lazy的属性有三个选项分别为：no-proxy、false和proxy）  
   fetch=select  
iv)、many-to-many懒加载  
此关联关系的懒加载和one-to-many的懒加载一样对程序的执行效率的提高都是非常明显的。  
   实现此种懒加载必须在从对象这边同时满足两个条件：  
    lazy!=false（lazy的属性有三个选项分别为：no-proxy、false和proxy）  
     fetch=select  
能够懒加载的对象都是被改过的代理对象，当相应的对象没有关闭时，访问这些懒加载对象的属性（getId和getClass除外）Hibernate会初始化这些代理，或用hibernate.initalize(proxy)来初始化代理对象；当关闭session后在访问懒加载的对象就会出现异常.

**10).MyBatis与Hibernate有什么不同**

Hibernate：Hibernate是当前最流行的ORM框架之一，对JDBC提供了较为完整的封装。Hibernate的O/R Mapping实现了POJO 和数据库表之间的映射，以及SQL的自动生成和执行。

Mybatis：Mybatis同样也是非常流行的ORM框架，主要着力点在于 POJO 与 SQL 之间的映射关系。然后通过映射配置文件，将SQL所需的参数，以及返回的结果字段映射到指定 POJO 。相对Hibernate“O/R”而言，Mybatis 是一种“Sql Mapping”的ORM实现

i). hibernate是全自动，而mybatis是半自动, hibernate完全可以通过对象关系模型实现对数据库的操作，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。而mybatis仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写sql来实现和管理;

ii) . hibernate数据库移植性远大于mybatis, hibernate通过它强大的映射结构和hql语言，大大降低了对象与数据库（oracle、mysql等）的耦合性，而mybatis由于需要手写sql，因此与数据库的耦合性直接取决于程序员写sql的方法，如果sql不具通用性而用了很多某数据库特性的sql语句的话，移植性也会随之降低很多，成本很高;

iii).hibernate拥有完整的日志系统, hibernate日志系统非常健全，涉及广泛，包括：sql记录、关系异常、优化警告、缓存提示、脏数据警告等；而mybatis则除了基本记录功能外，功能薄弱很多;

iv). mybatis相比hibernate需要关心很多细节, hibernate配置要比mybatis复杂的多，学习成本也比mybatis高。但也正因为mybatis使用简单，才导致它要比hibernate关心很多技术细节。mybatis由于不用考虑很多细节，开发模式上与传统jdbc区别很小，因此很容易上手并开发项目，但忽略细节会导致项目前期bug较多，因而开发出相对稳定的软件很慢，而开发出软件却很快。hibernate则正好与之相反。但是如果使用hibernate很熟练的话，实际上开发效率丝毫不差于甚至超越mybatis;

v). sql直接优化上，mybatis要比hibernate方便很多, 由于mybatis的sql都是写在xml里，因此优化sql比hibernate方便很多。而hibernate的sql很多都是自动生成的，无法直接维护sql；虽有hql，但功能还是不及sql强大，见到报表等变态需求时，hql也歇菜，也就是说hql是有局限的；hibernate虽然也支持原生sql，但开发模式上却与orm不同，需要转换思维，因此使用上不是非常方便。总之写sql的灵活度上hibernate不及mybatis

**11).Hibernate的缓存**

Hibernate 缓存包括两大类：Hibernate 一级缓存和 Hibernate 二级缓存：  
i）. Hibernate 一级缓存又称为“Session 的缓存”，它是内置的，不能被  
卸载。由于 Session 对象的生命周期通常对应一个数据库事务或者一个应用事务，因此它的缓存是事务范围的缓存。在第一级缓存中，持久化类的每个实例都具有  
唯一的 OID。  
ii）. Hibernate 二 级 缓 存 又 称 为 “ SessionFactory 的 缓 存 ”， 由 于  
SessionFactory 对象的生命周期和应用程序的整个过程对应，因此 Hibernate  
二级缓存是进程范围或者集群范围的缓存，有可能出现并发问题，因此需要采用  
适当的并发访问策略，该策略为被缓存的数据提供了事务隔离级别。第二级缓存  
是可选的，是一个可配置的插件，在默认情况下，SessionFactory 不会启用这  
个插件。  
iii).当 Hibernate 根据 ID 访问数据对象的时候，首先从 Session 一级缓存中查；  
查不到，如果配置了二级缓存，那么从二级缓存中查；如果都查不到，再查询数  
据库，把结果按照 ID 放入到缓存删除、更新、增加数据的时候，同时更新缓存。

[(Mybatis缓存](#_6).Mybatis的一级缓存与二级缓存))

###### **Web**

**1).Servlet与Filter的区别**

i).servlet 流程是短的，url传来之后，就对其进行处理，之后返回或转向到某一自己指定的页面。它主要用来在 业务处理之前进行控制.

ii).filter 流程是线性的， url传来之后，检查之后，可保持原来的流程继续向下执行，被下一个filter, servlet接收等，而servlet 处理之后，不会继续向下传递。filter功能可用来保持流程继续按照原来的方式进行下去，或者主导流程，而servlet的功能主要用来主导流程。

iii).filter可用来进行字符编码的过滤，检测用户是否登陆的过滤，禁止页面缓存等