****Spring知识点总结****

1. **Spring 框架**

**Spring框架是一个开源的容器性质的轻量级框架。主要有三大特点：容器、IOC（控制反转）、AOP（面向切面编程）。**

1. **Spring三大优点**

(**1) 容器。Spring框架是一个容器，能够管理项目中的所有对象。**

**(2) IOC（控制反转）。Spring将创建对象的方式反转了，从程序员自己创建反转给了程序。**

**(3) AOP（面向切面）。面向切面编程，简而言之，就是将纵向重复的代码横向抽取出来。Spring框架应用了面向切面的思想，主要体现在为容器中管理的对象生成动态代理对象。**

1. **控制反转（ICO）**

**控制反转，指得是将对象的创建权反转给Spring。作用是实现了程序的解耦合。**

1. **依赖注入（DI）**

**依赖注入，需要有IOC环境，在Spring创建Bean对象时，动态的将依赖对象注入到Bean对象中去。依赖注入最大的好处就是解耦合。**

1. **面向切面编程（AOP）**

**AOP将哪些与业务无关的，但对多个对象产生一个像的公共行为和逻辑，抽取并封装为一个可重用的模块。这个模块被命名为‘切面’（Aspect）,减少系统中的重复代码，降低了模块间的耦合，同时提高了系统的可维护性。可用于权限认证、日志，事务处理。**

**AOP实现的关键在于 代理模式，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；动态代理则以Spring AOP为代表。**

**（1）AspectJ是静态代理的增强，所谓静态代理，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，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。**

**（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。**

**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：**

**①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例,  生成目标类的代理对象。**

**②如果代理类没有实现 InvocationHandler 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指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。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。**

**（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。**

**InvocationHandler 的 invoke(Object  proxy,Method  method,Object[] args)：proxy是最终生成的代理实例;  method 是被代理目标实例的某个具体方法;  args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。**

1. **BeanFactory接口和ApplicationContext接口的理解**

**BeanFactory接口是Spring框架的顶层接口，是最原始的接口，通过new （BeanFactory的实现类）来启动Spring容器时，并不会创建Spring容器里面的对象，只有在每次通过getBean()获得对象时才会创建。**

**ApplicationContext接口是用来替代BeanFactory接口的，通过new （ApplicationContext接口的实现类）ClassPathXmlApplicationContext来启动Spring容器时，就会创建容器中配置的所有对象**

1. **Spring中的工厂容器**

**BeanFactory和ApplicationContext。**

**BeanFactory接口是Spring框架的顶层接口，是最原始的接口，ApplicationContext是对BeanFactory扩展，BeanFactory在第一次getBean时才会初始化Bean, ApplicationContext是会在加载配置文件时初始化Bean。**

1. **Bean标签**

**Bean标签用来描述Spring容器管理的对象。**

**假如有一个User对象，需要交给Spring容器来管理，这样就需要在Spring容器的主配置文件中通过Bean标签来描述该对象。Bean标签常见的属性有以下几个：**

**name属性：给被管理的对象起个名称，获得对象时要根据该名称来获得。**

**class属性：被管理对象的完整类名。**

**scope属性：scope属性常见的有两个属性值，singleton和prototype，这两个属性值用 来指定创建对象时是单例还是多例，默认为singleton(单例)，但在整合整合struts2 时,ActionBean必须配置为多例的.**

1. **Spring通过配置<bean>标签来生成Bean对象有哪三种方式？**

**空参构造方式、静态工厂方式和实例工厂方式。一般都只会用空参构造方式。如下：**

**<bean id="bean1" class="cn.itcast.spring.b\_instance.Bean1"></bean>**

1. **Spring框架中属性注入方式**

**Spring中的输入注入方式包括set方法注入、构造函数注入、p名称空间注入、spel注入，除此之外，还包括复杂方式注入，如数组、List、Map、Properties等属性的注入。**

1. **简述一下bean的生命周期？**

**bean的生命周期包括bean的定义、bean的初始化、bean的调用和bean的销毁。**

**在配置文件里面通过<bean></bean>来完成bean的定义，通过配置init-method属性来完成bean的初始化，通过得到bean的实例对象来完成bean的调用，通过配置destory-method属性来完成bean的销毁。**

1. **简述一下bean的作用域？**

**bean有5种作用域，分别是singleton（单例，默认）、prototype（默认）、request、session、globalSession。**

**singleton**

**当一个bean的作用域为singleton, 那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。**

**prototype**

**Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean() 方法）时都会创建一个新的bean实例。根据经验，对所有有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用 singleton作用域**

**request**

**在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用 域仅在基于web的Spring ApplicationContext情形下有效。**

**session**

**在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。**

**global session**

**在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于 web的Spring ApplicationContext情形下有效。**

**以上讲了Spring容器的常见配置，比如向Spring容器中注册一个对象，需要在Spring的主配置文件中用Bean标签来描述该对象，在实际开发中，一个项目中会有特别多的对象，如果都用Bean标签来配置，未免太过麻烦了。基于此，Spring容器为我们提供了注解的概念，用注解来代替配置。**

1. **如何使用Spring中的注解？**

**在使用注解之前，要先在spring的主配置文件中通过Context:component-scan标签来开启使用注解的开关。**

1. **用注解将对象注册到Spring容器当中的注解方式**

**4种**

**分别是：@Component()、@Service()、@Controller()、@Respository()。**

**Spring框架最早出现的只有@Component()注解，但如果所有的对象都使用同一个注解，很难区分对象究竟属于哪一层架构。基于此，Spring又推出了@Service()、@Controller()、@Respository()三种注解，用于区分对象属于哪一层架构。但4种注解方式从功能上来说没有任何区别。**

1. **Spring框架中，什么注解可以用来指定对象的作用范围？**

**@Scope(scopeName=”singleton”)。**

**14. 如何用注解的方式来完成属性注入？**

**按类型分可以分为值类型注入和引用类型注入。**

**值类型注入可以通过@Value()注解来完成，该注解既可以声明在属性上，也可以声明在方法上，建议声明在方法上，但是更多的人会声明在属性上，因为更方便。**

**引用类型注入可以通过三种注解方式来完成，分别为：@Autowired、@Autowired和@Qualifier()二者结合、@Resource()。建议使用@Resource()，但是一般我都会用@Autowired。**

****Spring中的AOP****

**1. 谈谈你对AOP的理解。**

**AOP，面向切面编程。简单来讲就是将纵向重复的代码，横向抽取出来。**

**很明显的一个体现就是在Filter过滤器中。在没有Filter之前，解决servlet的乱码问题是很复杂的，每次在接收请求之前，都要写句代码来解决乱码问题,即：request.setCharacterEncoding(“UTF-8”)，只要写一个servlet，就是写这句代码来解决乱码问题。直到有一天，Filter出现了。我们把解决乱码的那句代码放到Filter中去，从此在servlet中，就再也不用重复写那句代码了。从架构上来说，Filter解决乱码的事架在了所有的servlet上，这样一来，切面就形成了。面向切面编程的思想，还有一个直接的体现就是在拦截器中。**

**2. Spring中的AOP思想靠什么来体现的呢？**

**Spring中的AOP思想体现在能够为容器中管理的对象生成动态代理对象。**

1. **aop名词学习**

**Joinpoint（连接点）：目标对象中，所有可以增强的方法**

**Pointcut（切入点）：目标对象，已经增强的方法。**

**Advice（通知/增强）：增强的代码**

**Target（目标对象）：被代理对象**

**Weaving（织入）：将通知应用到切入点的过程**

**Proxy（代理）：将通知织入目标对象之后，形成代理对象**

1. **为什么叫springAOP呢？**

**Spring可以为所有service层的类生成动态代理对象，告诉spring，为每一个service层的类里面的方法添加管理事务的代码，spring一听，是！这样我们叫可以只写一遍管理事务的代码，就为所有的service就都加上了管理事务的代码。这样一来，aop思想就体现出来了。**

**SpringAOP的本质就是帮我们生成动态代理对象。**

1. **Spring实现AOP的原理？**

**JDK动态代理和cglib代理。**

**JDK动态代理有缺陷，就是被代理对象必须实现接口才能产生代理对象，如果没有接口，就不能使用动态代理技术。我们用spring容器来实现动态代理，假如要管理的对象没有实现接口，那么就不能产生代理对象了。为了让所有的对象都能产生动态代理对象，Spring又融入了第三方代理技术cglib代理。Cglib可以对任何类生成代理对象，它的原理是对目标对象进行继承代理，如果目标对象被final修饰，那么该类无法被cglib代理。**

1. **那么Spring到底使用的是JDK代理，还是cglib代理呢？**

**混合使用。如果被代理对象实现了接口，就优先使用JDK代理，如果没有实现接口，就用用cglib代理。**

1. **Spring切面可以应用5种类型的通知，哪5种？**

**前置通知（Before）、后置通知（After，在方法完成之后调用通知，无论方法执行是否成功）、后置通知（After-returning，在方法成功执行之后调用通知）、异常通知（After-throwing，在方法抛出异常后调用通知）、环绕通知(Around，在目标方法之前之后都调用)。**

1. **Spring中应用aop，需要哪些步骤？**

**（1）导包**

**（2）准备目标对象**

**（3）准备通知**

**（4）将通知织入目标对象中**

**Spring中配置aop实际上主要配的是第四步——将通知织入目标对象中。有两种方式，一种是xml配置方式，一种是注解配置方式。**

**先看xml配置方式：**

**再看注解配置方式：**

1. **Spring AOP的实现方式有哪些？**

**1、经典的基于代理的AOP：使用Java代码实现，编写Advice、PointCut，然后提供给Advisor使用。开启自动代理后，即可在applicationContext中获得增强后的bean。**

**2、@AspectJ注解驱动的切面：基于注解的开发（推荐使用），在项目中需要开启AOP自动代理<aop:aspectj-autoproxy/>。**

**3、XML Schema方式：需要实现相应的增强接口，如BeforeAdvice、AfterAdvice等。然后利用一下配置如：**

**read-only="true" />**

****Spring中的事务管理****

**1. 简单介绍一下Spring中的事务管理。**

**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。现实世界中最常见的事务例子可能就是转账了。**

**2. 事务的4个特性？**

**ACID。**

**原子性（Atomic）：事务是由一个或多个活动所组成的一个工作单元。原子确保事务中的所有操作全部发生或全部不发生。如果所有的活动都成功了，事务也就成功了。如果任意一个活动失败了，整个事务也失败并回滚。**

**一致性（Consistent）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态。现实的数据不应该被损坏。**

**隔离性（Isolated）:事务允许多个用户对相同的数据进行操作，每个用户的操作不会与其他用户纠缠在一起。因此，事务应该被彼此隔离，避免发生同步读写相同数据的事情（注意的是，隔离性往往涉及到锁定数据库中的行或表）。**

**持久性（Durable）:一旦事务完成，事务的结果应该持久化，这样就能从任何的系统崩溃中恢复过来。这一般会涉及将结果存储到数据库或其他形式的持久化存储中。**

**通俗的说事务，指一组操作，要么都成功执行，要么都不执行（原子性）**

**在所有的操作没有执行完毕之前，其它会话不能够看到中间改变的过程（隔离性）**

**事务发生前和发生后，数据的总额保持不变（一致性）**

**事务产生的影响不能被撤销（持久性）**

**3. 在Spring的核心配置文件applicationContext.xml中配置事务，主要配置三大方面：事务管理器、事务通知和定义事务性切面。**

**代码如下：**

**<beans xmlns="http://www.springframework.org/schema/beans"**

**xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p"**

**xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd**

**http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd**

**http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd**

**http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.2.xsd">**

**<!-- 事务管理器 -->**

**<bean id="transactionManager"**

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

**<!-- 数据源 -->**

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

**</bean>**

**<!-- 事务通知 -->**

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

**<tx:attributes>**

**<!-- 传播行为 -->**

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

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

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

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

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

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

**<tx:method name="find\*" propagation="SUPPORTS" read-only="true" />**

**<tx:method name="select\*" propagation="SUPPORTS" read-only="true" />**

**<tx:method name="get\*" propagation="SUPPORTS" read-only="true" />**

**</tx:attributes>**

**</tx:advice>**

**<!-- 定义事务性切面 -->**

**<aop:config>**

**<aop:advisor advice-ref="txAdvice"**

**pointcut="execution(\* cn.e3mall.service..\*.\*(..))" />**

**</aop:config>**

**</beans>**

**4. 事务管理器**

**Spring管理事务有多种方式，但无论选择将事务编码到Bean中还是将选择其定义为切面，你都需要使用spring事务管理器与平台相关的事务进行交互。让我们看一下Spring的事务管理器能如何帮你避免直接与平台相关的实现打交道？Spring并不直接管理事务，而是提供了多种事务管理器，每个事务管理器都会充当某一特定平台的事务实现的门面，这使得门户在Spring中使用事务时，几乎不用关注实际的事务实现是什么。**

**操作事务的方法一般就三种：开启事务、提交事务、回滚事务。针对这三个方法，JDBC、Hibernate、Mybatis等都有不同的实现方式。因为在不同平台、操作事务的代码各不相同。为此，Spring提供了一个接口PlatformTransactionManager，针对不同的操作平台提供不同的实现类。比方说，如果是JDBC和iBatis，实现类是DataSourceTransactionManager，如果是Hibernate，实现类是HibernateTransactionManager。不同的实现类就是不同的事务管理器，为了使用事务管理器，你需要将其声明在应用程序上下文中。**

**<!-- 事务管理器 -->**

**<bean id="transactionManager"**

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

**<!-- 数据源 -->**

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

**</bean>**

**如果在应用程序中你直接使用JDBC/iBatis来进行持久化，DataSourceTransactionManager会为你处理事务边界。为了使用DataSourceTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：**

**如果应用程序的持久化是通过Hibernate实现的，那么你需要使用HibernateTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：**

**<!-- 事务管理器 -->**

**<bean id="transactionManager"**

**class="org.springframework.orm.hibernate3.HibernateTransactionManager">**

**<!-- 数据源 -->**

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

**</bean>**

**当配置好事务管理器之后，在spring中具体配置事务时，又有两种选择：一种是编码事务（了解），一种是声明式事务。声明式事务是通过事务属性来定义的，具体的说，是通过传播行为、隔离级别、只读提示、事务超时及回滚规则来进行定义的。**

**5. 事务的传播行为指什么？**

**在业务（service）方法之间平行调用时，如何来处理事务的问题。**

**6. Spring中定义事务传播行为的属性有几种，分别是什么？**

**7种。**

**分别是REQUIRED、REQUIRES\_NEW、SUPPORTS、NOT\_SUPPORTED、MANDATORY、NESTED、NEVER。我们常用的是propagation=”REQUIRED”，默认的就是REQUIRED，指得是支持当前事务，如果不存在，就新建一个（默认），所以这个属性不用配置。其余6个属性几乎不用。**

**7. 事务中可能会出现的并发问题有哪些？**

**脏读、不可重复读、幻读。**

**解决并发问题的方法就是设置隔离级别。**

**在理想情况下，事务之间是完全隔离的，从而可以防止这些问题发生，但是完全隔离会导致性能问题，因为它通常会涉及锁定数据库中的记录，侵占性的锁定会阻碍并发性，要求事务互相等待以完成各自的工作。考虑到完全的隔离会导致性能问题，而且并不是所有的应用程序都需要完全的隔离，所以有时应用程序需要在事务隔离上有一定的灵活性。因此就会有各种隔离级别。**

**8. 事务中有几种隔离级别呢？**

**4种。**

**读未提交（uncommitted）、读已提交（committed）、可重复读（repeatable\_read）、串行化（serializable）。**

**读未提交（uncommitted）：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。**

**读已提交（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。**

**可重复读（repeatable\_read）：可以阻止脏读和不可重复读，但可能会导致幻读。**

**串行化（serializable）：完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读。这是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的。**

**9. 声明式事务的第三个属性是只读。read-only=”true” 或者 read-only=”false”，在定义查询方法时，通常设置为只读。**

**10. 事务超时和回滚规则 这两个属性几乎不用，用到的时候再查资料。**

**11. 基于在5大事务属性，就可以在XML中定义事务了。代码如下：**

**<!-- 事务通知 -->**

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

**<tx:attributes>**

**<!-- 传播行为 -->**

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

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

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

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

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

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

**<tx:method name="find\*" propagation="SUPPORTS" read-only="true" />**

**<tx:method name="select\*" propagation="SUPPORTS" read-only="true" />**

**<tx:method name="get\*" propagation="SUPPORTS" read-only="true" />**

**</tx:attributes>**

**</tx:advice>**

**<tx:advice>只是定义了AOP通知，用于把事务边界通知给方法，但是这只是事务通知，而不是完整的事务性切面。我们在<tx:advice>中没有声明哪些Bean应该被通知——我们需要一个切点来做这件事，为了完整定义事务切面，我们必须定义一个通知器(advisor)。这就涉及aop命名空间了。以下的XML定义了一个通知器，它使用txAdvice通知所有实现SpitterService接口的Bean，代码如下：**

**<!-- 定义事务性切面 -->**

**<aop:config>**

**<aop:advisor advice-ref="txAdvice"**

**pointcut="execution(\* cn.e3mall.service..\*.\*(..))" />**

**</aop:config>**

**<tx:advice>配置元素极大地简化了Spring声明式事务所需要的XML，如果我告诉你它甚至可以进一步简化，你会感受如何呢？如果我再告诉你，你只需要在Spring上下文中添加一行XML，即可声明事务，你的感受如何呢？这就是用注解的方式来配置事务。**

****Spring整合SSM****

**使用Spring整合SSM框架，需要配置哪些东西？简单说一下。**

**Spring整合Dao，要配置数据库启动，配置SqlSessionFactory，配置Mapper动态代理。**

**Spring整合Service，要配置包扫描器（开启注解驱动），要配置事务管理（事务管理器，事务通知，事务切面）。**

**Spring整合Web，要配置包扫描器（扫描Controller）,配置处理器映射器和处理器适配器（采用注解驱动的方法），配置视图解析器。**

**在web.xml，要加载Spring容器，并且要配置一个springMVC的前端控制器。**

**组件**

**SqlSessionFactoryBuilder：构造器**

**SqlSessionFactory：工厂接口，由Builder生成**

**基础文件配置，用于配置数据库信息等**

**typeAlias:定义元素别名，代表com.learn.ssm.chapter3.pojo.Role类可以使用自定义别名**

**environment:配置使用数据库的链接信息**

**dataSource:设置mubatis内部提供连接池的方式**

**mapper:引入自定的映射器配置文件**

**映射文件配置，配置结合业务所实现的SQL语句关系**

**SqlSession：会话，发送sql，返回结果**

**核心接口，等同于JDBC里面的Connection链接对象，**

**1、获取Mapper接口**

**2、发送SQL个数据库**

**3、控制数据库事务**

**DefaultSqlSession**

**单线程使用**

**SqlSessionManager**

**多线程使用**

**SQL Mapper：映射器，由java和xml文件构成，构造映射规则，发送SQL，执行完后返回结果**

**组成部分**

**java实体对象**

**interface Mapper类**

**xml映射文件**

**namespace：mapper接口对应的全路径名称，例：com.sun.ssm.dao.UserDao**

**resultMap：定义的model实体对象，id=map配置的名称，type=对象实体的全路径名称（com.sun.ssm.model.User），注意id的名称为下面sql查询配置所使用到实体类型**

**<id column="id" property="id" jdbcType="BIGINT"/> 设置主键类型**

**<result column="user\_name" property="userName" jdbcType="VARCHAR"/>**

**column对应数据库列名**

**property对应实体对象属性名**

**jdbcType数据库映射类型**

**select、update、delete、add对应数据库的增删改查语句**

**id：对应interface接口里面的方法名称**

**resultMap：sql语句返回的类型，对应resultMap设置的id名称**

**parameterType：如果sql语句需要单个参数，设置请求参数类型**

**#{参数设置}**

**单个参数：SELECT \* FROM t\_user WHERE id = #{userId} ---对应---selectUserById(@Param("userId") Long userId);**

**多个参数：SELECT \* FROM t\_user WHERE user\_email = #{emailOrPhone} AND user\_state = #{state} ---对应---selectUserByPhoneOrEmail(@Param("emailOrPhone") String emailOrPhone, @Param("state") Short state);**

**注解形式实现(此种形式不建议使用)**

**创建新的接口 interface2**

**@select("select \*from tb where id=#{id}")**

**配置文件mapper 链接由resource改为class，地址为接口全路径**

**生命周期**

**SqlSessionFactoryBuilder用于创建SqlSessionFactory，创建成功后即失效**

**SqlSessionFactory等同于多个数据库的连接池（整个mysql），在整个mybatis的生命周期中，如果创建多个，则会耗光连接池，造成应用崩溃，所以一般为单例模式，整个应用进行共享**

**SqlSession对应单个数据库进行的命令操作，如果使用完，则会把链接返还给SqlSessionFactory连接池**

**Mapper由session创建，一个mapper请求对应一个业务处理**

**配置**

**配置文件**

**配置文件节点顺序不能颠倒**

**配置文件xml**

**configuration：根节点**

**properties：属性**

**settings：设置**

**typeAliases：类型命名**

**typeHandlers：类型处理器**

**objectFactory：对象工厂**

**plugins：插件**

**environments：配置环境**

**environment：环境变量**

**transactionManager：事务管理器**

**dataSource：数据源**

**databaseIdProvider：数据库厂商标识**

**mappers：对象映射器**

**xml 属性节点介绍**

**properties：属性（优先级：程序代码＞文件＞子元素）**

**property子元素**

**<property name="data.driver" value="com.mysql.jdbc.Driver">**

**<property name="data.url" value="jdbc:mysql://localhost:3306">**

**然后在dataSource里面使用<property name="driver" value="${data.driver}">等同于占位符**

**properties文件**

**创建一个jdbc.properties文件**

**data.url=jdbc:mysql://......**

**data.username=root**

**<properties resource="jdbc.properties">**

**程序代码传递**

**如果需要对字符串加密，则在程序里面读取配置文件，然后写入解密后的内容**

**settings：设置**

**<setting name=“cacheEnabled” value=“true”>是否启用缓存**

**等等。。。还有很多配置项**

**cacheEnabled--true | false--true（name--value--默认值）该配置影响的所有映射器中配置的缓存的全局开关**

**lazyLoadingEnabled--true | false--false 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态**

**aggressiveLazyLoading--true | false--true 当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性将会按需加载。**

**multipleResultSetsEnabled--true | false--true 是否允许单一语句返回多结果集（需要兼容驱动）。**

**useColumnLabel--true | false--true 使用列标签代替列名。不同的驱动在这方面会有不同的表现， 具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。**

**useGeneratedKeys--true | false--false 允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。**

**autoMappingBehavior--NONE, PARTIAL, FULL--PARTIAL 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。**

**defaultExecutorType--SIMPLE, REUSE, BATCH--SIMPLE 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。**

**defaultStatementTimeout--int-- 设置超时时间，它决定驱动等待数据库响应的秒数。**

**safeRowBoundsEnabled--true | false--False允许在嵌套语句中使用分页（RowBounds）。**

**mapUnderscoreToCamelCase--true | false--False是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。**

**localCacheScope--SESSION | STATEMENT--SESSION MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。**

**jdbcTypeForNull-- 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。**

**lazyLoadTriggerMethods --equals,clone,hashCode,toString 指定哪个对象的方法触发一次延迟加载。**

**defaultScriptingLanguage--org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver 指定动态 SQL 生成的默认语言。**

**callSettersOnNulls--true | false--false 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。**

**logPrefix--指定 MyBatis 增加到日志名称的前缀 String**

**logImpl--指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING**

**proxyFactory--CGLIB | JAVASSIST--JAVASSIST指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。**

**typeAliases：别名**

**可以用简写的形式自定义类型名称**

**比如：int--Integer（mybits--java）**

**使用代码进行注册**

**自定义别名**

**xml：<typeAlias alias="user" type="类的全称地址">**

**扫描形式：<package name="需要扫描的包"> 如果当前包里面有2个类，都会自动转成pacal命名法的别名（首字母小写，其余大写）**

**代码里面可以针对类设置 在class类上面增加 @Alias("sun")，则在扫描时，名称为自定义名称**

**typeHandler：类型转换器**

**用于承担 数据库与java之间的类型转换，jdbcType--javaType，mybatis会进行自动处理**

**自定义Handler（可以用于枚举转int，等等）**

**集成baseTypeHandler接口，自己实现4个方法**

**在配置文件中使用**

**在全局xml使用：<typeHandler jdbcType="数据库类型" javaType="java类型" jandler="handler实现类的包名全路径">**

**在mapper映射xml文件使用：<resule property="java类型" colum="数据库类型" typeHandler="handler实现类的包名全路径">**

**如果自定义处理类过多，使用扫描包形式加载**

**<package name="包的全路径">**

**此种形式无法定义类型映射，可以在自定义实现类class上面使用注解形式添加映射，如：@MappedTypes（String.class） @MappedjdbcTypes（jdbcType.VARCHAR）**

**objectFactory：对象工厂**

**一般使用默认就好，创建结果集时，mybatis会创建结果集的实例，用于返回给用户。**

**plugins：插件**

**environments：运行环境**

**用于配置数据库的链接信息**

**transactionManager：事务管理器**

**JdbcTransaction，以jdbc的事务形式进行处理**

**ManagedTransaction，使用mybits自定义的对象事务处理，提交和回滚不进行操作，把事务交给容器处理，默认会关闭链接。**

**dataSource**

**分为3种形式链接数据库，可以进行同时配置<dataSource type="POOLED">...**

**UNPOOLED**

**使用非连接池形式，每次请求，创建一个新的连接**

**POOLED**

**使用连接池形式，节省资源，可以设置最大连接数**

**poolMaxmumActiveConnections：任意时间活跃的链接，默认10**

**poolMaxmumIdleConnections：任意时间空闲的连接数**

**poolMaxmumCheckoutTime：池中检出的时间，默认20000（20秒）**

**poolTimeToWait：如果获取链接时间过长，会打印状态日志**

**poolPingQuery：侦测查询，用于判断数据库是否可以正常查询**

**poolPingEnabled：是否启用侦测查询，默认不开启（如果开启，请设置一个效率非常快的SQL，例如 select 1）**

**poolPingConnectionNotUsedFor：配置侦测查询的频度**

**JNDI**

**databaseIdProvider：数据库厂商标识**

**用于配置支持多种数据库操作**

**<databaseIdProvider type="DB\_VENDOR"><property name="MYSQL" value="mysql" /><property name="Oracle" value="oracle" /><property name="DB2" value="db2" /></databaseIdProvider >**

**配置完成之后，在Mapper映射器里面 对sql语句进行不同数据的查询切换操作，例如<select id="aa" ... datanaseId="oracle">或者<select id="aa" ... datanaseId="mysql">**

**mappers ：对象映射器**

**先编写接口代码，创建对应增删改查接口，映射文件里面首行<mapper>标签会使用namespace定义接口代码全路径，下面设置sql查询语句（select、delete、....）**

**引入方式：**

**文件引入：<mapper resource="xml全路径">**

**包名引入：<package name="包全路径">**

**类注册引入：<mapper class="映射接口类的全路径">**

**userMapper.xml：暂时不考虑**

**映射器**

**一个映射xml对应一个接口类型，然后对应对象**

**select**

**默认sql返回的列名需要和对象属性一一对应上，例如，role\_name对应roleName，按照驼峰命名法则映射，如果列名直接使用role\_name as roleName形式返回，则不用使用默认驼峰映射**

**xml节点属性配置**

**id：方法名称，供Mybatis使用，如果相同命名空间id不唯一，则会报错**

**parameterTyppe：传递参数类型，可以是int、string、或者类**

**List<map>：接口类使用map，传递查询参数，类似.net的HashTable，不方便阅读维护**

**@Param("占位符名称1")：接口类也可以使用多个Param进行传递，xml映射文件可以直接使用变量，例如：<select id="findName" resultType="返回对象类">select id,role\_Name as roleName,create\_Time as createTime from tb\_User where role\_Name like concat('%',#{roleName},'%') and id=#{id}</select>，参数过多时不方便使用**

**使用javaBean形式（java class类对象）：传入一个class类型作用请求，paramType="类的全路径"**

**混合使用：javaBean与Param同时使用。@Param("user") User user,@Param("name") Name name，在xml占位符中则为#{user.age}**

**resultType：返回结果类型，可以是int、string、或者类**

**resultMap：返回结果自定义使用，比如说使用typeHandler**

**xml：映射配置文件上方 会使用resultMap定义一个数据库查询类型与POJO对象的映射配置节点，下方select查询语句中使用resultMap="上方定义节点的id属性的名称"**

**自定义Handler：**

**flushCache：调用完SQL之后，是否清空查询的本地缓存和二级缓存，默认false，不进行清理**

**useCache：是否启动二级缓存，默认true，启用二级缓存**

**timeout：设置当前语句查询超时时间**

**fetchSize：获取记录返回总条数，如果设定100，最大返回100条数据**

**statementType：设置mybatis使用JDBC的哪种Statement工作，默认PREPARED,（个人不太理解）**

**resultSetType:**

**databaseId：使用多种数据库引擎支持的ID**

**resultOrdered：**

**resultSets：使用于多个结果集情况，一次查询返回多个dataTable表格数据**

**RowBounds：myBatis内置分页类对象。在DAO接口类中，定义RowBounds请求参数，xml不需要进行配置，因为已经内置在sqlSession中。（因为他的原理是先把sql数据全部查出来在内存里面进行分页，大数据量并不合适）**

**insert**

**xml节点属性配置（大部分同select相同，只记录不同项）**

**useGeneratedKeys：主键回填，是否启用使用jdbc里面的同名方法获取内部生成的主键（类似sqlserver自增主键），默认false（启用之后，keyProperty与keyColumn选一个）。使用方法：<insert ..... useGeneratedKeys="true" keyProperty="id">，意思是-如果插入完数据，则会回填keyProperty设置的属性值**

**keyProperty：会根据useGeneratedKeys的返回值，标记一个属性返回值，默认不启用**

**keyColumn：不能和keyProperty同时使用**

**xml节点内部配置：**

**selectKey：自定义主键，可以根据自定义的设置规则，生成主键值，进行回填和插入，例如：下图**

**结构层级：insert → selectKey（keyProperty：回填主键名称，resultType：回填值类型，order：是插入语句执行前使用还是执行后使用，默认BEFORE），设置了order之后，这样在insert语句真正执行前，id主键会按照自定义的规则生成，然后插入到对应列中**

**update、delete（比较简单，一般使用单个参数或者javaBean参数形式，返回结果为int型影响行数）**

**sql：允许定义一部分sql，在各个地方引用**

**在mapper根节点下，设置sql节点，里面包含通用变量，则在select或者insert等操作时统一调用，例如：<sql id="fileds"> id,name,age</sql>，在select语句里面，<select id="findOne" .....>select <include refid="fileds" /> fom User </select>，insert 同理。**

**节点还支持变量传递 <sql id="fileds"> ${U}.id,${U}.name,${U}.age</sql>，整体使用：<select id="findOne" .....> select <include refid="fileds"> <propety name="U" value="tb" /> </include> from TbUser as tb</select>。（refid的值为自定义sql的id，里面propety节点代表属性内容， name值为自定义sql使用占位符变量，value值为 查询语句中表的别名，name与value为映射关系）**

**resultMap：描述数据库结果集映射到的对象**

**xml节点**

**resultMap**

**constructor：适用于对象里面带参数的构造函数**

**idArg**

**arg**

**id**

**result**

**association**

**collection**

**discriminator**

**case**

**节点注释**

**resultMap：type属性对应对象的类型**

**id：设置对应的主键**

**result：字段映射节点，例如：<result property="对象属性" column="数据库列名" />**

**association：级联查询，适用于一对一的情况**

**例子：<association property="当前对象子对象的class类型" column="对应关联属性的列名" select="要调用其他接口类的方法全路径" />**

**property：通过调用select对应的接口方法，返回的对象类型**

**column：子查询所需要传入的父查询ID，例如：select (select top 1 name from CHIRLD as b where b.pid=a.id),\* from PARENT as A，等同于传入子查询的a.id的功能**

**collection：级联查询，适用于一对多的情况**

**例子：<collection property="当前对象子对象的class类型" column="对应关联属性的列名" select="要调用其他接口类的方法全路径" />**

**节点属性等同于association**

**discriminator：级联查询，选择器查询，根据不同的值调用不用的查询器**

**例子：<discriminator javaType="选择列的数据类型" column="选择列的名称" > <case value="如果是当前值，1" resultMap="当前页面其他查询器ID1" /><case value="如果是当前值，2" resultMap="当前页面其他查询器ID2" /> </discriminator>，当前映射xml文件会存在多个resultMap节点，其他从节点：<resultMap type="返回对象类型的全路径" id="当前页面其他查询器ID1" extends="继承与哪个resultMap(为主节点ID)"> <association property="返回子对象class类型(首字母小写)" column="对应关联属性的列名" select="要调用其他接口类的方法全路径" /></resultMap>，这样通过不同的case 则会调用不同类型的接口查询。**

**javaType：用于指定传入列的类型**

**column：传入列的列名**

**还需要case子节点**

**value：具体值**

**resultMap：其他resultMap从节点的ID**

**还需要指定其他从resultMap节点**

**type：返回对象类型**

**id：查询器id**

**extends：继承于哪个resultMap(为resultMap主节点ID)**

**子节点：association 或者 collection**

**property：返回对象属性名称**

**select：使用哪个查询接口类的全路径**

**column：传入这个选择的内容是主sql的哪个字段（多个参数，使用逗号分隔）**

**级联查询的性能问题**

**因为固定映射写死，所以获取对象则会将级联的数据给查询出来，因为有的时候不需要，所以造成性能浪费。**

**解决问题如下：**

**延迟加载**

**使用setting节点里面的 lazyLoadingEnabled（是否启用） 和 aggressiveLazyLoading （是否执行加载）节点组合**

**fetchType：设置在级联查询中的节点属性，（association、collection存在，discriminator选择器没有），有两个值：eager、lazy，仍需要配合lazyLoadingEnabled使用，不过需要设置aggressiveLazyLoading=false**

**eager：获取主对象之后立刻加载对应数据**

**lazy：获取对象之后，延迟加载关联数据**

**sql关联查询**

**使用sql关联将所有从表数据进行查询，设置好别名，wf\_id，如果wf\_id不为空，则进行调用级联查询，在主resultMap节点里面使用级联查询节点配置，缺点是，配置可能过长，不好理解**

**多对多级联，本质上多对多的关联查询只是表结构，实际业务总会以某张表作为主表进行关联的**

**cache：给定某个命名空间的缓存配置**

**将查询数据存入二级缓存**

**<cache type="使用缓存类的全路径"></cache>**

**type：自定义缓存类，可以使用Redis、MongoDB等缓存工具，必须要实现ibatis.cache.Cache接口**

**size：缓存对象的个数，默认1024**

**flushInterval：刷新缓存的时间（单位毫秒），insert、update、delete才有效**

**eviction：缓存策略，（默认LRU）**

**LRU：最近最少使用-移出最长时间不使用的对象**

**FIFO：先进先出**

**SORT：软引用，移出基于垃圾回收器状态和软引用规则的对象**

**WEAK：弱引用，比SORT更积极的回收对象，回收规则通sort一致**

**readOnly：缓存是否只读，默认false**

**blocking：使用使用阻塞式缓存，默认不使用**

**cache-ref：其他命名空间缓存配置的引用**

**存储过程：**

**参数：**

**参数传入的内容存在null形式，这个时候执行sql，可能会报错，如果需要进行自定义Handler处理，则需要在变量里面指定处理器，例如：#{age , javaType=int , jdbcType=NUMERIC , typeHandler=MyTypeHandler}，此时age传入的参数变量，则会经过myTypeHandler处理器进行处理。**

**存储过程：**

**输入参数（IN）：外部传入存储过程**

**输出参数（OUT）：存储过程返回结果**

**输入输出参数（INOUT）：传入的参数，经过存储过程处理，再返回出去**

**使用方法，传入参数#{age, mode=IN} 或者#{age, mode=INOUT}**

**使用方法：**

**设置POJO对象，用于对应存储过程的传入参数和传出参数**

**CURD都可以使用存储过程**

**在对应语句执行器上使用 statementType="CALLABLE"，代表使用存储过程**

**{ call 存储过程名称(参数1，参数2，参数3)}**

**参数1、2为输入参数，#{userName，mode=IN，jdbcType=VARCHAR}**

**userName为POJO对象的属性名称**

**此处不指定存储过程参数名称是因为，调用存储过程的参数是按照顺序传入的**

**参数3为输出参数，#{count，mode=OUT，jdbcType=INTEGER}**

**游标：**

**缓存**

**mybatis框架对于缓存分为一、二级缓存两层**

**一级缓存，存在SqlSession，默认开启**

**参数和sql没有变化，用同一个SqlSession的Mapper对象，则会将内存中进行读取**

**不同SqlSession不同享缓存**

**二级缓存**

**存在SqlSessionFactory，使用不同的sqlsession对象，查询完成之后，需要使用sqlsession.commit()，进行提交，提交之后才会进行缓存到Factory里面**

**开始二级缓存，在mapper的xml配置文件中使用<cache />标签**

**动态SQL**

**语法元素**

**if：条件判断**

**在select查询器里面，select \* from tb where disabled=0 <if test="roleName != null"> and name='#{name}'</if>**

**test：语句为sql判断格式**

**节点里面内容为要拼接的sql部分语句**

**choose：多条件判断，类似 case when then**

**使用原理同if， <choose><when test=""></when><when test=""></when><otherwise></otherwise> </choose>**

**otherwise：等同于最后一个else**

**trim：处理拼装的条件，例如：去掉多余的and 、or等**

**<trim prefix="" prefixOverrides=""><if test="roleName != null"> and name='#{name}'</if></trim>**

**在其他语句最外层包一层trim，用于去掉收尾指定的多余参数**

**prefix：在里面语句生成之后，前面追加固定字符串**

**prefixOverrides：去掉生成之后的最前方的固定字符串**

**suffixOverrides：去掉生成之后的最后方的固定字符串**

**set：用于更新语句**

**等同于 update Tb <set> <if></if><if></if> </set> where id=#{id}**

**set标签会自动把最后一个逗号给去掉**

**foreach：循环**

**通常应用于in操作，<foreach item="" index="" collection="" open="" separator="" close="">#{roleNo}</foreach>**

**item：当前元素对象**

**index：当前元素的下标**

**collection：传入的参数对象属性，通常为list，set等**

**open：包裹元素的前置固定字符串**

**separator：连接参数字符串**

**close：包裹元素的后置固定字符串**

**bind：自定义变量**

**<bind name="name1" value="'%'+ 参数+'%'" />，下面sql使用为 select \* from Tb where name like #{name1}，等同于传入变量使用**

**特点在于提升了可读性和移植性**

**————————————————**

**版权声明：本文为CSDN博主「一只小羊」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。**

**原文链接：https://blog.csdn.net/sun491922556/article/details/78777890**