## 四、关于框架

### 1、spring

### 2、springIOC的理解；

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

深入理解：

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

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

DI：

DI—Dependency Injection，即“依赖注入

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

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

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

●为什么需要依赖：容器管理对象需要IoC容器来提供对象需要的外部资源；

●谁注入谁：很明显是IoC容器注入某个对象，也就是注入“依赖对象”；

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

### 4、springIOC的java底层原理；

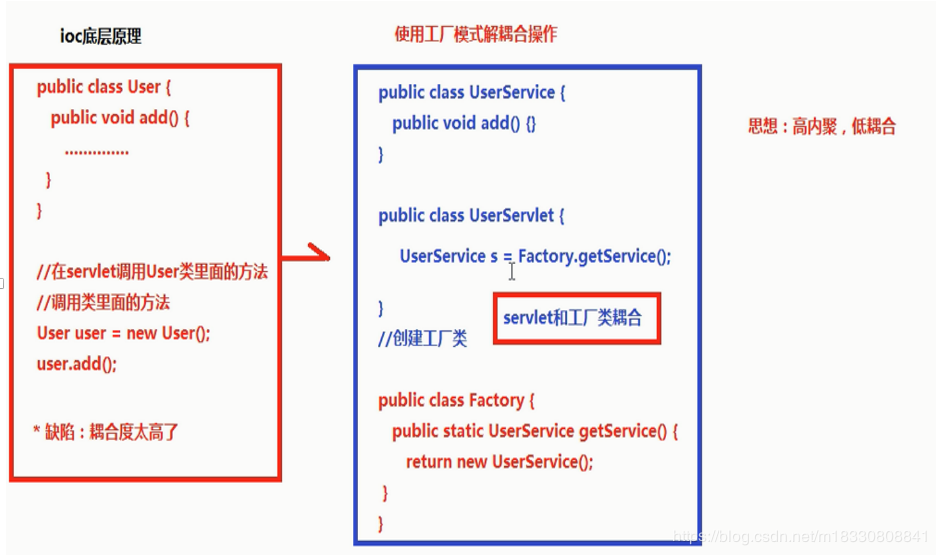
ioc底层原理使用技术

（1）xml配置文件

（2）dom4j解决xml

（3）工厂设计模式

（4）反射





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

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

spring的ioc操作，就是通过spring配置创建类对象。

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

### 6、如何控制事务；

Spring 的事务，可以说是 Spring AOP 的一种实现。

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

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

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

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

### 7、事务的隔离级别和传播机制

事务四大特性:

　　\* 原子性(Atomicity)  :强调的事务的不可分割.

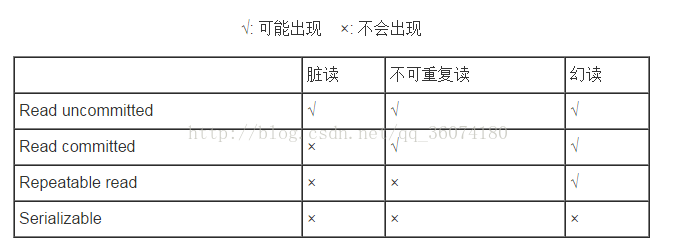
　　\* 一致性(Consistency)    :强调的事务的执行前后,数据库的的完整性保持一致.

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

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

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

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



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

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

      这种隔离级别会产生脏读，不可重复读和幻像读。

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

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

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

一个事务范围内两个相同的查询却返回了不同数据，这就是不可重复读

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

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

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

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

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

      除了防止脏读，不可重复读外，还避免了幻像读。

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

**脏读：A事务处理过程中读取了B事务未提交的数据，这就是脏读**

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

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

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

### 9、springAOP的应用场景；

**场景一： 记录日志**

**场景二： 监控方法运行时间 （监控性能）**

**场景三： 权限控制**

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

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

### 10、mybatis的结构？

### 11、配置文件如何配置？

### 12、mybatis处理多个参数的几种方式？

### 13、占位符有几种？区别是什么？

### 14、一级缓存和二级缓存的区别？

### 15、Mybatis的类型转换器？

### 16、mybatis接口没有实现类，他是如何操作数据库的？

### hibernate

### 17、mybatis 与hibernate的对比；优势，劣势，原理，如何选择；

| **项目** | **Mybatis** | **Hibernate** |
| --- | --- | --- |
| 开发效率 | 较容易上手；需要手动管理sql | 较难上手；无需管理sql，专注业务 |
| 入门 | 简单 | 较难 |
| 灵活性 | sql更加灵活 | 几乎不用写sql，封装性较大 |
| 性能 | 相对较高 | 相对较低 |
| 日志系统 | 除了基本记录功能外，功能薄弱很多 | 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 |
| 优化 | 更容易 | 更难 |
| 自动性 | 半自动 | 全自动 |
| 缓存 |  | 有更好的二级缓存机制 |
| 移植性 | 移植性低（绑定了sql） | 移植性高 |
| 适用场合 | 1.有复杂的查询业务;2.无hibernate实践经验的 | 1.无复杂的查询业务；2.能够熟练应用hibernate的 |

### 18、springboot与springmvc最大的优势在哪？

### 19、一个加了事务注解的“新增方法”，抛出异常，数据有没有添加成功？