这部分主要是开源Java EE框架方面的内容，包括Hibernate、MyBatis、Spring、Spring MVC等，由于Struts 2已经是明日黄花，在这里就不讨论Struts 2的面试题，如果需要了解相关内容，可以参考我的另一篇文章《Java面试题集（86-115）》。此外，这篇文章还对企业应用架构、大型网站架构和应用服务器优化等内容进行了简单的探讨，这些内容相信对面试会很有帮助。

126、什么是ORM?

答：对象关系映射（Object-Relational Mapping，简称ORM）是一种为了解决程序的面向对象模型与数据库的关系模型互不匹配问题的技术；简单的说，ORM是通过使用描述对象和数据库之间映射的元数据（在Java中可以用XML或者是注解），将程序中的对象自动持久化到关系数据库中或者将关系数据库表中的行转换成Java对象，其本质上就是将数据从一种形式转换到另外一种形式。

127、持久层设计要考虑的问题有哪些？你用过的持久层框架有哪些?

答：所谓"持久"就是将数据保存到可掉电式存储设备中以便今后使用，简单的说，就是将内存中的数据保存到关系型数据库、文件系统、消息队列等提供持久化支持的设备中。持久层就是系统中专注于实现数据持久化的相对独立的层面。

持久层设计的目标包括：

- 数据存储逻辑的分离，提供抽象化的数据访问接口。
- 数据访问底层实现的分离，可以在不修改代码的情况下切换底层实现。
- 资源管理和调度的分离，在数据访问层实现统一的资源调度（如缓存机制）。
- 数据抽象，提供更面向对象的数据操作。

持久层框架有：

- Hibernate
- MyBatis
- TopLink
- Guzz
- jOOQ
- Spring Data
- ActiveJDBC

128、Hibernate中SessionFactory是线程安全的吗？Session是线程安全的吗（两个线程能够共享同一个Session吗）？

答：SessionFactory对应Hibernate的一个数据存储的概念，它是线程安全的，可以被多个线程并发访问。SessionFactory一般只会在启动的时候构建。对于应用程序，最好将SessionFactory通过单例模式进行封装以便于访问。Session是一个轻量级非线程安全的对象（线程间不能共享session），它表示与数据库进行交互的一个工作单元。Session是由SessionFactory创建的，在任务完成之后它会被关闭。Session是持久层服务对外提供的主要接口。Session会延迟获取数据库连接（也就是在需要的时候才会获取）。为了避免创建太多的session，可以使用ThreadLocal将session和当前线程绑定在一起，这样可以让同一个线程获得的总是同一个session。Hibernate 3中SessionFactory的getCurrentSession()方法就可以做到。

129、Hibernate中Session的load和get方法的区别是什么?

答：主要有以下三项区别：

① 如果没有找到符合条件的记录，get方法返回null，load方法抛出异常。

② get方法直接返回实体类对象，load方法返回实体类对象的代理。

③ 在Hibernate 3之前，get方法只在一级缓存中进行数据查找，如果没有找到对应的数据则越过二级缓存，直接发出SQL语句完成数据读取；load方法则可以从二级缓存中获取数据；从Hibernate 3开始，get方法不再是对二级缓存只写不读，它也是可以访问二级缓存的。

> **说明：**对于load()方法Hibernate认为该数据在数据库中一定存在可以放心的使用代理来实现延迟加载，如果没有数据就抛出异常，而通过get()方法获取的数据可以不存在。

130、Session的save()、update()、merge()、lock()、saveOrUpdate()和persist()方法分别是做什么的？有什么区别?

答：Hibernate的对象有三种状态：瞬时态（transient）、持久态（persistent）和游离态（detached），如第135题中的图所示。瞬时态的实例可以通过调用save()、persist()或者saveOrUpdate()方法变成持久态；游离态的实例可以通过调用update()、saveOrUpdate()、lock()或者replicate()变成持久态。save()和persist()将会引发SQL的INSERT语句，而update()或merge()会引发UPDATE语句。save()和update()的区别在于一个是将瞬时态对象变成持久态，一个是将游离态对象变为持久态。merge()方法可以完成save()和update()方法的功能，它的意图是将新的状态合并到已有的持久化对象上或创建新的持久化对象。对于persist()方法，按照官方文档的说明：① persist()方法把一个瞬时态的实例持久化，但是并不保证标识符被立刻填入到持久化实例中，标识符的填入可能

被推迟到flush的时间；② persist()方法保证当它在一个事务外部被调用的时候并不触发一个INSERT语句，当需要封装一个长会话流程的时候，persist()方法是很有必要的；③ save()方法不保证第②条，它要返回标识符，所以它会立即执行INSERT语句，不管是在事务内部还是外部。至于lock()方法和update()方法的区别，update()方法是把一个已经更改过的脱管状态的对象变成持久状态；lock()方法是把一个没有更改过的脱管状态的对象变成持久状态。

131、阐述Session加载实体对象的过程。

答：Session加载实体对象的步骤是：

① Session在调用数据库查询功能之前，首先会在一级缓存中通过实体类型和主键进行查找，如果一级缓存查找命中且数据状态合法，则直接返回；

② 如果一级缓存没有命中，接下来Session会在当前NonExists记录（相当于一个查询黑名单，如果出现重复的无效查询可以迅速做出判断，从而提升性能）中进行查找，如果NonExists中存在同样的查询条件，则返回null；

③ 如果一级缓存查询失败则查询二级缓存，如果二级缓存命中则直接返回；

④ 如果之前的查询都未命中，则发出SQL语句，如果查询未发现对应记录则将此次查询添加到Session的NonExists中加以记录，并返回null；

⑤ 根据映射配置和SQL语句得到ResultSet，并创建对应的实体对象；

⑥ 将对象纳入Session（一级缓存）的管理；

⑦ 如果有对应的拦截器，则执行拦截器的onLoad方法；

⑧ 如果开启并设置了要使用二级缓存，则将数据对象纳入二级缓存；

⑨ 返回数据对象。

132、Query接口的list方法和iterate方法有什么区别?

答：

① list()方法无法利用一级缓存和二级缓存（对缓存只写不读），它只能在开启查询缓存的前提下使用查询缓存；iterate()方法可以充分利用缓存，如果目标数据只读或者读取频繁，使用iterate()方法可以减少性能开销。

② list()方法不会引起N+1查询问题，而iterate()方法可能引起N+1查询问题

> **说明：**关于N+1查询问题，可以参考CSDN上的一篇文章《什么是N+1查询》

133、Hibernate如何实现分页查询?

答：通过Hibernate实现分页查询，开发人员只需要提供HQL语句（调用Session的createQuery()方法）或查询条件（调用Session的createCriteria()方法）、设置查询起始行数（调用Query或Criteria接口的setFirstResult()方法）和最大查询行数（调用Query或Criteria接口的setMaxResults()方法），并调用Query或Criteria接口的list()方法，Hibernate会自动生成分页查询的SQL语句。

134、锁机制有什么用？简述Hibernate的悲观锁和乐观锁机制。

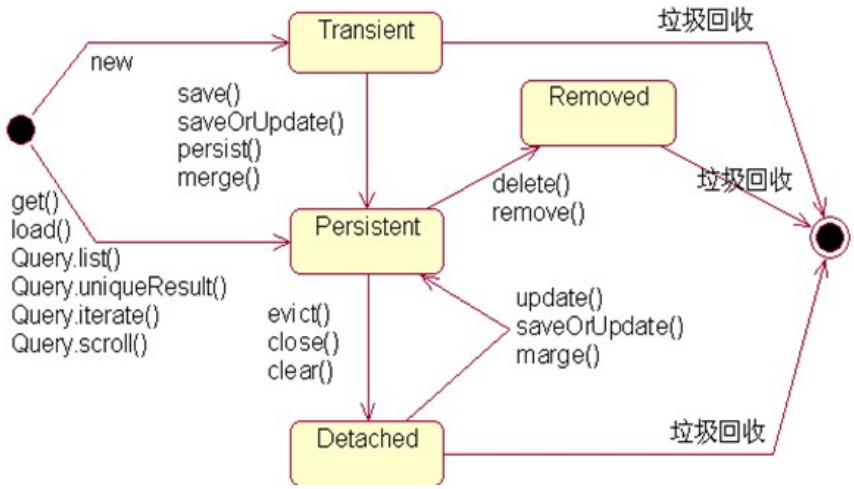
答：有些业务逻辑在执行过程中要求对数据进行排他性的访问，于是需要通过一些机制保证在此过程中数据被锁住不会被外界修改，这就是所谓的锁机制。

Hibernate支持悲观锁和乐观锁两种锁机制。悲观锁，顾名思义悲观的认为在数据处理过程中极有可能存在修改数据的并发事务（包括本系统的其他事务或来自外部系统的事务），于是将处理的数据设置为锁定状态。悲观锁必须依赖数据库本身的锁机制才能真正保证数据访问的排他性，关于数据库的锁机制和事务隔离级别在《Java面试题大全（上）》中已经讨论过了。乐观锁，顾名思义，对并发事务持乐观态度（认为对数据的并发操作不会经常性的发生），通过更加宽松的锁机制来解决由于悲观锁排他性的数据访问对系统性能造成的严重影响。最常见的乐观锁是通过数据版本标识来实现的，读取数据时获得数据的版本号，更新数据时将此版本号加1，然后和数据库表对应记录的当前版本号进行比较，如果提交的数据版本号大于数据库中此记录的当前版本号则更新数据，否则认为是过期数据无法更新。Hibernate中通过Session的get()和load()方法从数据库中加载对象时可以通过参数指定使用悲观锁；而乐观锁可以通过给实体类加整型的版本字段再通过XML或@Version注解进行配置。

> **提示：**使用乐观锁会增加了一个版本字段，很明显这需要额外的空间来存储这个版本字段，浪费了空间，但是乐观锁会让系统具有更好的并发性，这是对时间的节省。因此乐观锁也是典型的空间换时间的策略。

135、阐述实体对象的三种状态以及转换关系。

答：最新的Hibernate文档中为Hibernate对象定义了四种状态（原来是三种状态，面试的时候基本上问的也是三种状态），分别是：瞬时态（new, or transient）、持久态（managed, or persistent）、游状态（detached）和移除态（removed，以前Hibernate文档中定义的三种状态中没有移除态），如下图所示，就以前的Hibernate文档中移除态被视为是瞬时态。



- 瞬时态：当new一个实体对象后，这个对象处于瞬时态，即这个对象只是一个保存临时数据的内存区域，如果没有变量引用这个对象，则会被JVM的垃圾回收机制回收。这个对象所保存的数据与数据库没有任何关系，除非通过Session的save()、saveOrUpdate()、persist()、merge()方法把瞬时态对象与数据库关联，并把数据插入或者更新到数据库，这个对象才转换为持久态对象。
- 持久态：持久态对象的实例在数据库中有对应的记录，并拥有一个持久化标识（ID）。对持久态对象进行delete操作后，数据库中对应的记录将被删除，那么持久态对象与数据库记录不再存在对应关系，持久态对象变成移除态（可以视为瞬时态）。持久态对象被修改变更后，不会马上同步到数据库，直到数据库事务提交。
- 游离态：当Session进行了close()、clear()、evict()或flush()后，实体对象从持久态变成游离态，对象虽然拥有持久和与数据库对应记录一致的标识值，但是因为对象已经从会话中清除掉，对象不在持久化管理之内，所以处于游离态（也叫脱管态）。游离态的对象与临时状态对象是十分相似的，只是它还含有持久化标识。

> 提示：关于这个问题，在Hibernate的官方文档中有更为详细的解读。

136、如何理解Hibernate的延迟加载机制？在实际应用中，延迟加载与Session关闭的矛盾是如何处理的？
答：延迟加载就是并不是在读取的时候就把数据加载进来，而是等到使用时再加载。Hibernate使用了虚拟代理机制实现延迟加载，我们使用Session的load()方法加载数据或者一对多关联映射在使用延迟加载的情况下从一的一方加载多的一方，得到的都是虚拟代理，简单的说返回给用户的并不是实体本身，而是实体对象的代理。代理对象在用户调用getter方法时才会去数据库加载数据。但加载数据就需要数据库连接。而当我们把会话关闭时，数据库连接就同时关闭了。

延迟加载与session关闭的矛盾一般可以这样处理：
① 关闭延迟加载特性。这种方式操作起来比较简单，因为Hibernate的延迟加载特性是可以通过映射文件或者注解进行配置的，但这种解决方案存在明显的缺陷。首先，出现"no session or session was closed"通常说明系统中已经存在主外键关联，如果去掉延迟加载的话，每次查询的开销都会变得很大。
② 在session关闭之前先获取需要查询的数据，可以使用工具方法Hibernate.isInitialized()判断对象是否被加载，如果没有被加载则可以使用Hibernate.initialize()方法加载对象。
③ 使用拦截器或过滤器延长Session的生命周期直到视图获得数据。Spring整合Hibernate提供的OpenSessionInViewFilter和OpenSessionInViewInterceptor就是这种做法。

137、举一个多对多关联的例子，并说明如何实现多对多关联映射。
答：例如：商品和订单、学生和课程都是典型的多对多关系。可以在实体类上通过@ManyToMany注解配置多对多关联或者通过映射文件中的和标签配置多对多关联，但是实际项目开发中，很多时候都是将多对多关联映射转换成两个多对一关联映射来实现的。

138、谈一下你对继承映射的理解。
答：继承关系的映射策略有三种：
① 每个继承结构一张表（table per class hierarchy），不管多少个子类都用一张表。

② 每个子类一张表（table per subclass），公共信息放一张表，特有信息放单独的表。
③ 每个具体类一张表（table per concrete class），有多少个子类就有多少张表。
第一种方式属于单表策略，其优点在于查询子类对象的时候无需表连接，查询速度快，适合多态查询；缺点是可能导致表很大。后两种方式属于多表策略，其优点在于数据存储紧凑，其缺点是需要进行连接查询，不适合多态查询。

139、简述Hibernate常见优化策略。
答：这个问题应当挑自己使用过的优化策略回答，常用的有：
① 制定合理的缓存策略（二级缓存、查询缓存）。
② 采用合理的Session管理机制。
③ 尽量使用延迟加载特性。
④ 设定合理的批处理参数。
⑤ 如果可以，选用UUID作为主键生成器。
⑥ 如果可以，选用基于版本号的乐观锁替代悲观锁。
⑦ 在开发过程中，开启hibernate.show_sql选项查看生成的SQL，从而了解底层的状况；开发完成后关闭此选项。
⑧ 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的DBA（数据库管理员）提供支持。

140、谈一谈Hibernate的一级缓存、二级缓存和查询缓存。
答：Hibernate的Session提供了一级缓存的功能，默认总是有效的，当应用程序保存持久化实体、修改持久化实体时，Session并不会立即把这种改变提交到数据库，而是缓存在当前的Session中，除非显示调用了Session的flush()方法或通过close()方法关闭Session。通过一级缓存，可以减少程序与数据库的交互，从而提高数据库访问性能。
SessionFactory级别的二级缓存是全局性的，所有的Session可以共享这个二级缓存。不过二级缓存默认是关闭的，需要显示开启并指定需要使用哪种二级缓存实现类（可以使用第三方提供的实现）。一旦开启了二级缓存并设置了需要使用二级缓存的实体类，SessionFactory就会缓存访问过的该实体类的每个对象，除非缓存的数据超出了指定的缓存空间。
一级缓存和二级缓存都是对整个实体进行缓存，不会缓存普通属性，如果希望对普通属性进行缓存，可以使用查询缓存。查询缓存是将HQL或SQL语句以及它们的查询结果作为键值对进行缓存，对于同样的查询可以直接从缓存中获取数据。查询缓存默认也是关闭的，需要显示开启。

141、Hibernate中DetachedCriteria类是做什么的?
答：DetachedCriteria和Criteria的用法基本上是一致的，但Criteria是由Session的createCriteria()方法创建的，也就意味着离开创建它的Session，Criteria就无法使用了。DetachedCriteria不需要Session就可以创建（使用DetachedCriteria.forClass()方法创建），所以通常也称其为离线的Criteria，在需要进行查询操作的时候再和Session绑定（调用其getExecutableCriteria(Session)方法），这也就意味着一个DetachedCriteria可以在需要的时候和不同的Session进行绑定。

142、@OneToMany注解的mappedBy属性有什么作用?
答：@OneToMany用来配置一对多关联映射，但通常情况下，一对多关联映射都由多的一方来维护关联关系，例如学生和班级，应该在学生类中添加班级属性来维持学生和班级的关联关系（在数据库中是由学生表中的外键班级编号来维护学生表和班级表的多对一关系），如果要使用双向关联，在班级类中添加一个容器属性来存放学生，并使用@OneToMany注解进行映射，此时mappedBy属性就非常重要。如果使用XML进行配置，可以用<set>标签的inverse="true"设置来达到同样的效果。

143、MyBatis中使用`#`和`$`书写占位符有什么区别?
答：`#`将传入的数据都当成一个字符串，会对传入的数据自动加上引号；`$`将传入的数据直接显示生成在SQL中。注意：使用`$`占位符可能会导致SQL注射攻击，能用`#`的地方就不要使用`$`，写order by子句的时候应该用`$`而不是`#`。

144、解释一下MyBatis中命名空间（namespace）的作用。
答：在大型项目中，可能存在大量的SQL语句，这时候为每个SQL语句起一个唯一的标识（ID）就变得并不容易了。为了解决这个问题，在MyBatis中，可以为每个映射文件起一个唯一的命名空间，这样定义在这个映射文件中的每个SQL语句就成了定义在这个命名空间中的一个ID。只要我们能够保证每个命名空间中这个ID是唯一的，即使在不同映射文件中的语句ID相同，也不会再产生冲突了。

145、MyBatis中的动态SQL是什么意思?
答：对于一些复杂的查询，我们可能会指定多个查询条件，但是这些条件可能存在也可能不存在，例如在58同城上面找房子，我们可能会指定面积、楼层和所在位置来查找房源，也可能会指定面积、价格、户型和所在位置来查找房源，此时就需要根据用户指定的条件动态生成SQL语句。如果不使用持久层框架我们可能需要自己拼装SQL语句，还好MyBatis提供了动态SQL的功能来解决这个问题。MyBatis中用于实现动态SQL的元素主要有：
– if
– choose / when / otherwise
– trim

– where
– set
– foreach

下面是映射文件的片段。

```
    <select id="foo" parameterType="Blog" resultType="Blog">
        select * from t_blog where 1 = 1
        <if test="title != null">
            and title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            and owner = #{owner}
        </if>
    </select>
```

当然也可以像下面这些书写。

```
    <select id="foo" parameterType="Blog" resultType="Blog">
        select * from t_blog where 1 = 1
        <choose>
            <when test="title != null">
                and title = #{title}
            </when>
            <when test="content != null">
                and content = #{content}
            </when>
            <otherwise>
                and owner = "owner1"
            </otherwise>
        </choose>
    </select>
```

再看看下面这个例子。

```
    <select id="bar" resultType="Blog">
        select * from t_blog where id in
        <foreach collection="array" index="index"
            item="item" open="(" separator="," close=")">
            #{item}
        </foreach>
    </select>
```

146、什么是IoC和DI？DI是如何实现的？

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

举个例子：一个类A需要用到接口B中的方法，那么就需要为类A和接口B建立关联或依赖关系，最原始的方法是在类A中创建一个接口B的实现类C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类A中定义好用于关联接口B的方法（构造器或setter方法），将类A和接口B的实现类C放入容器中，通过对容器的配置来实现二者的关联。

依赖注入可以通过setter方法注入（设值注入）、构造器注入和接口注入三种方式来实现，Spring支持setter注入和构造器注入，通常使用构造器注入来注入必须的依赖关系，对于可选的依赖关系，则setter注入是更好的选择，setter注入需要类提供无参构造器或者无

参的静态工厂方法来创建对象。

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

> **补充：**设计模式中的创建型模式中也有一个原型模式，原型模式也是一个常用的模式，例如做一个室内设计软件，所有的素材都在工具箱中，而每次从工具箱中取出的都是素材对象的一个原型，可以通过对象克隆来实现原型模式。

Spring 2.x中针对WebApplicationContext新增了3个作用域，分别是：request（每次HTTP请求都会创建一个新的Bean）、session（同一个HttpSession共享同一个Bean，不同的HttpSession使用不同的Bean）和globalSession（同一个全局Session共享一个Bean）。

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

ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。ThreadLocal，顾名思义是线程的一个本地化对象，当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量。

ThreadLocal类非常简单好用，只有四个方法，能用上的也就是下面三个方法：
– void set(T value)：设置当前线程的线程局部变量的值。
– T get()：获得当前线程所对应的线程局部变量的值。
– void remove()：删除当前线程中线程局部变量的值。

ThreadLocal是如何做到为每一个线程维护一份独立的变量副本的呢？在ThreadLocal类中有一个Map，键为线程对象，值是其线程对应的变量的副本，自己要模拟实现一个ThreadLocal类其实并不困难，代码如下所示：

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class MyThreadLocal<T> {
    private Map<Thread, T> map = Collections.synchronizedMap(new HashMap<Thread, T>());

    public void set(T newValue) {
        map.put(Thread.currentThread(), newValue);
    }

    public T get() {
        return map.get(Thread.currentThread());
    }

    public void remove() {
        map.remove(Thread.currentThread());
    }
}
```

148、解释一下什么叫AOP（面向切面编程）？
答：AOP（Aspect-Oriented Programming）指一种程序设计范型，该范型以一种称为切面（aspect）的语言构造为基础，切面是一种新的模块化机制，用来描述分散在对象、类或方法中的横切关注点（crosscutting concern）。

149、你是如何理解"横切关注"这个概念的?
答："横切关注"是会影响到整个应用程序的关注功能，它跟正常的业务逻辑是正交的，没有必然的联系，但是几乎所有的业务逻辑都会涉及到这些关注功能。通常，事务、日志、安全性等关注就是应用中的横切关注功能。

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

答：

a. 连接点（Joinpoint）：程序执行的某个特定位置（如：某个方法调用前、调用后，方法抛出异常后）。一个类或一段程序代码拥有一些具有边界性质的特定点，这些代码中的特定点就是连接点。Spring仅支持方法的连接点。

b. 切点（Pointcut）：如果连接点相当于数据中的记录，那么切点相当于查询条件，一个切点可以匹配多个连接点。Spring AOP的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点。

c. 增强（Advice）：增强是织入到目标类连接点上的一段程序代码。Spring提供的增强接口都是带方位名的，如：BeforeAdvice、AfterReturningAdvice、ThrowsAdvice等。很多资料上将增强译为“通知”，这明显是个词不达意的翻译，让很多程序员困惑了许久。

> **说明：** Advice在国内的很多书面资料中都被翻译成"通知"，但是很显然这个翻译无法表达其本质，有少量的读物上将这个词翻译为"增强"，这个翻译是对Advice较为准确的诠释，我们通过AOP将横切关注功能加到原有的业务逻辑上，这就是对原有业务逻辑的一种增强，这种增强可以是前置增强、后置增强、返回后增强、抛异常时增强和包围型增强。

d. 引介（Introduction）：引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过引介功能，可以动态的未该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

e. 织入（Weaving）：织入是将增强添加到目标类具体连接点上的过程，AOP有三种织入方式：①编译期织入：需要特殊的Java编译期（例如AspectJ的ajc）；②装载期织入：要求使用特殊的类加载器，在装载类的时候对类进行增强；③运行时织入：在运行时为目标类生成代理实现增强。Spring采用了动态代理的方式实现了运行时织入，而AspectJ采用了编译期织入和装载期织入的方式。

f. 切面（Aspect）：切面是由切点和增强（引介）组成的，它包括了对横切关注功能的定义，也包括了对连接点的定义。

> **补充：** 代理模式是GoF提出的23种设计模式中最为经典的模式之一，代理模式是对象的结构模式，它给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。简单的说，代理对象可以完成比原对象更多的职责，当需要为原对象添加横切关注功能时，就可以使用原对象的代理对象。我们在打开Office系列的Word文档时，如果文档中有插图，当文档刚加载时，文档中的插图都只是一个虚框占位符，等用户真正翻到某页要查看该图片时，才会真正加载这张图，这其实就是对代理模式的使用，代替真正图片的虚框就是一个虚拟代理；Hibernate的load方法也是返回一个虚拟代理对象，等用户真正需要访问对象的属性时，才向数据库发出SQL语句获得真实对象。

下面用一个找枪手代考的例子演示代理模式的使用：

```
/**
 * 参考人员接口
 * @author 骆昊
 *
 */
public interface Candidate {

    /**
     * 答题
     */
    public void answerTheQuestions();
}
```

```
/**
 * 懒学生
 * @author 骆昊
 *
 */
public class LazyStudent implements Candidate {
    private String name;        // 姓名

    public LazyStudent(String name) {
        this.name = name;
    }

    @Override
    public void answerTheQuestions() {
        // 懒学生只能写出自己的名字不会答题
        System.out.println("姓名：" + name);
```

```
    }

}
```

```
/**
 * 枪手
 * @author 骆昊
 *
 */
public class Gunman implements Candidate {
    private Candidate target;   // 被代理对象

    public Gunman(Candidate target) {
        this.target = target;
    }

    @Override
    public void answerTheQuestions() {
        // 枪手要写上代考的学生的姓名
        target.answerTheQuestions();
        // 枪手要帮助懒学生答题并交卷
        System.out.println("奋笔疾书正确答案");
        System.out.println("交卷");
    }

}
```

```
public class ProxyTest1 {

    public static void main(String[] args) {
        Candidate c = new Gunman(new LazyStudent("王小二"));
        c.answerTheQuestions();
    }
}
```

说明：从JDK 1.3开始，Java提供了动态代理技术，允许开发者在运行时创建接口的代理实例，主要包括Proxy类和InvocationHandler接口。下面的例子使用动态代理为ArrayList编写一个代理，在添加和删除元素时，在控制台打印添加或删除的元素以及ArrayList的大小：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.List;

public class ListProxy<T> implements InvocationHandler {
    private List<T> target;

    public ListProxy(List<T> target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        Object retVal = null;
        System.out.println("[" + method.getName() + ": " + args[0] + "]");
        retVal = method.invoke(target, args);
        System.out.println("[size=" + target.size() + "]");
        return retVal;
    }

}
```

```java
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.List;

public class ProxyTest2 {

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        Class<?> clazz = list.getClass();
        ListProxy<String> myProxy = new ListProxy<String>(list);
        List<String> newList = (List<String>)
                Proxy.newProxyInstance(clazz.getClassLoader(),
                clazz.getInterfaces(), myProxy);
        newList.add("apple");
        newList.add("banana");
        newList.add("orange");
        newList.remove("banana");
    }
}
```

> **说明：**使用Java的动态代理有一个局限性就是代理的类必须要实现接口，虽然面向接口编程是每个优秀的Java程序都知道的规则，但现实往往不尽如人意，对于没有实现接口的类如何为其生成代理呢？继承！继承是最经典的扩展已有代码能力的手段，虽然继承常常被初学者滥用，但继承也常常被进阶的程序员忽视。CGLib采用非常底层的字节码生成技术，通过为一个类创建子类来生成代理，它弥补了Java动态代理的不足，因此Spring中动态代理和CGLib都是创建代理的重要手段，对于实现了接口的类就用动态代理为其生成代理类，而没有实现接口的类就用CGLib通过继承的方式为其创建代理。

151、Spring中自动装配的方式有哪些?

答：

– no：不进行自动装配，手动设置Bean的依赖关系。

– byName：根据Bean的名字进行自动装配。

– byType：根据Bean的类型进行自动装配。

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

– autodetect：如果有默认的构造器，则通过constructor的方式进行自动装配，否则使用byType的方式进行自动装配。

> **说明：**自动装配没有自定义装配方式那么精确，而且不能自动装配简单属性（基本类型、字符串等），在使用时应注意。

152、Spring中如何使用注解来配置Bean？有哪些相关的注解?

答：首先需要在Spring配置文件中增加如下配置：

```xml
<context:component-scan base-package="org.example"/>
```

然后可以用@Component、@Controller、@Service、@Repository注解来标注需要由Spring IoC容器进行对象托管的类。这几个注解没有本质区别，只不过@Controller通常用于控制器，@Service通常用于业务逻辑类，@Repository通常用于仓储类（例如我们的DAO实现类），普通的类用@Component来标注。

153、Spring支持的事务管理类型有哪些？你在项目中使用哪种方式?

答：Spring支持编程式事务管理和声明式事务管理。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，因为编程式事务允许你通过代码控制业务。

事务分为全局事务和局部事务。全局事务由应用服务器管理，需要底层服务器JTA支持（如WebLogic、WildFly等）。局部事务和底层采用的持久化方案有关，例如使用JDBC进行持久化时，需要使用Connetion对象来操作事务；而采用Hibernate进行持久化时，需要使用Session对象来操作事务。

Spring提供了如下所示的事务管理器。

事务管理器实现类	目标对象
DataSourceTransactionManager	注入DataSource
HibernateTransactionManager	注入SessionFactory
JdoTransactionManager	管理JDO事务
JtaTransactionManager	使用JTA管理事务
PersistenceBrokerTransactionManager	管理Apache的OJB事务

这些事务的父接口都是PlatformTransactionManager。Spring的事务管理机制是一种典型的策略模式，PlatformTransactionManager代表事务管理接口，该接口定义了三个方法，该接口并不知道底层如何管理事务，但是它的实现类必须提供getTransaction()方法（开启事务）、commit()方法（提交事务）、rollback()方法（回滚事务）的多态实现，这样就可以用不同的实现类代表不同的事务管理策略。使用JTA全局事务策略时，需要底层应用服务器支持，而不同的应用服务器所提供的JTA全局事务可能存在细节上的差异，因此实际配置全局事务管理器是可能需要使用JtaTransactionManager的子类，如：WebLogicJtaTransactionManager（Oracle的WebLogic服务器提供）、UowJtaTransactionManager（IBM的WebSphere服务器提供）等。

编程式事务管理如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/sch
    xmlns:p="http://www.springframework.org/schema/context"
     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/sch
     http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring

     <context:component-scan base-package="com.jackfrued"/>

     <bean id="propertyConfig"
         class="org.springframework.beans.factory.config.
  PropertyPlaceholderConfigurer">
         <property name="location">
             <value>jdbc.properties</value>
         </property>
     </bean>

     <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
         <property name="driverClassName">
             <value>${db.driver}</value>
         </property>
         <property name="url">
             <value>${db.url}</value>
         </property>
         <property name="username">
             <value>${db.username}</value>
         </property>
         <property name="password">
             <value>${db.password}</value>
         </property>
     </bean>

     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
         <property name="dataSource">
             <ref bean="dataSource" />
         </property>
     </bean>

     <!-- JDBC事务管理器 -->
     <bean id="transactionManager"
         class="org.springframework.jdbc.datasource.
```

```
       DataSourceTransactionManager"  scope="singleton">
         <property name="dataSource">
             <ref bean="dataSource" />
         </property>
     </bean>

     <!-- 声明事务模板 -->
     <bean id="transactionTemplate"
         class="org.springframework.transaction.support.
   TransactionTemplate">
         <property name="transactionManager">
             <ref bean="transactionManager" />
         </property>
     </bean>

</beans>
```

```
package com.jackfrued.dao.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;

import com.jackfrued.dao.EmpDao;
import com.jackfrued.entity.Emp;

@Repository
public class EmpDaoImpl implements EmpDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public boolean save(Emp emp) {
        String sql = "insert into emp values (?,?,?)";
        return jdbcTemplate.update(sql, emp.getId(), emp.getName(), emp.getBirthday()) == 1;
    }

}
```

```
package com.jackfrued.biz.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

import com.jackfrued.biz.EmpService;
import com.jackfrued.dao.EmpDao;
import com.jackfrued.entity.Emp;

@Service
public class EmpServiceImpl implements EmpService {
    @Autowired
    private TransactionTemplate txTemplate;
    @Autowired
    private EmpDao empDao;

    @Override
    public void addEmp(final Emp emp) {
        txTemplate.execute(new TransactionCallbackWithoutResult() {

            @Override
            protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
```

```
                empDao.save(emp);
            }
        });
    }


}
```

声明式事务如下图所示，以Spring整合Hibernate 3为例，包括完整的DAO和业务逻辑代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.2.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">

    <!-- 配置由Spring IoC容器托管的对象对应的被注解的类所在的包 -->
    <context:component-scan base-package="com.jackfrued" />

    <!-- 配置通过自动生成代理实现AOP功能 -->
    <aop:aspectj-autoproxy />

    <!-- 配置数据库连接池 (DBCP) -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- 配置驱动程序类 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- 配置连接数据库的URL -->
        <property name="url" value="jdbc:mysql://localhost:3306/myweb" />
        <!-- 配置访问数据库的用户名 -->
        <property name="username" value="root" />
        <!-- 配置访问数据库的口令 -->
        <property name="password" value="123456" />
        <!-- 配置最大连接数 -->
        <property name="maxActive" value="150" />
        <!-- 配置最小空闲连接数 -->
        <property name="minIdle" value="5" />
        <!-- 配置最大空闲连接数 -->
        <property name="maxIdle" value="20" />
        <!-- 配置初始连接数 -->
        <property name="initialSize" value="10" />
        <!-- 配置连接被泄露时是否生成日志 -->
        <property name="logAbandoned" value="true" />
        <!-- 配置是否删除超时连接 -->
        <property name="removeAbandoned" value="true" />
        <!-- 配置删除超时连接的超时门限值(以秒为单位) -->
        <property name="removeAbandonedTimeout" value="120" />
        <!-- 配置超时等待时间(以毫秒为单位) -->
        <property name="maxWait" value="5000" />
        <!-- 配置空闲连接回收器线程运行的时间间隔(以毫秒为单位) -->
        <property name="timeBetweenEvictionRunsMillis" value="300000" />
        <!-- 配置连接空闲多长时间后(以毫秒为单位)被断开连接 -->
        <property name="minEvictableIdleTimeMillis" value="60000" />
    </bean>
```

```
55
56      <!-- 配置Spring提供的支持注解ORM映射的Hibernate会话工厂 -->
57      <bean id="sessionFactory"
58          class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
59          <!-- 通过setter注入数据源属性 -->
60          <property name="dataSource" ref="dataSource" />
61          <!-- 配置实体类所在的包 -->
62          <property name="packagesToScan" value="com.jackfrued.entity" />
63          <!-- 配置Hibernate的相关属性 -->
64          <property name="hibernateProperties">
65              <!-- 在项目调试完成后要删除show_sql和format_sql属性否则对性能有显著影响 -->
66              <value>
67                  hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
68              </value>
69          </property>
70      </bean>
71
72      <!-- 配置Spring提供的Hibernate事务管理器 -->
73      <bean id="transactionManager"
74          class="org.springframework.orm.hibernate3.HibernateTransactionManager">
75          <!-- 通过setter注入Hibernate会话工厂 -->
76          <property name="sessionFactory" ref="sessionFactory" />
77      </bean>
78
79      <!-- 配置基于注解配置声明式事务 -->
80      <tx:annotation-driven />
81
82  </beans>
```

```
1   package com.jackfrued.dao;
2
3   import java.io.Serializable;
4   import java.util.List;
5
6   import com.jackfrued.comm.QueryBean;
7   import com.jackfrued.comm.QueryResult;
8
9   /**
10   * 数据访问对象接口(以对象为单位封装CRUD操作)
11   * @author 骆昊
12   *
13   * @param <E> 实体类型
14   * @param <K> 实体标识字段的类型
15   */
16  public interface BaseDao <E, K extends Serializable> {
17
18      /**
19       * 新增
20       * @param entity 业务实体对象
21       * @return 增加成功返回实体对象的标识
22       */
23      public K save(E entity);
24
25      /**
26       * 删除
27       * @param entity 业务实体对象
28       */
29      public void delete(E entity);
30
31      /**
32       * 根据ID删除
33       * @param id 业务实体对象的标识
34       * @return 删除成功返回true否则返回false
35       */
```

```
    public boolean deleteById(K id);

    /**
     * 修改
     * @param entity 业务实体对象
     * @return 修改成功返回true否则返回false
     */
    public void update(E entity);

    /**
     * 根据ID查找业务实体对象
     * @param id 业务实体对象的标识
     * @return 业务实体对象对象或null
     */
    public E findById(K id);

    /**
     * 根据ID查找业务实体对象
     * @param id 业务实体对象的标识
     * @param lazy 是否使用延迟加载
     * @return 业务实体对象对象
     */
    public E findById(K id, boolean lazy);

    /**
     * 查找所有业务实体对象
     * @return 装所有业务实体对象的列表容器
     */
    public List<E> findAll();

    /**
     * 分页查找业务实体对象
     * @param page 页码
     * @param size 页面大小
     * @return 查询结果对象
     */
    public QueryResult<E> findByPage(int page, int size);

    /**
     * 分页查找业务实体对象
     * @param queryBean 查询条件对象
     * @param page 页码
     * @param size 页面大小
     * @return 查询结果对象
     */
    public QueryResult<E> findByPage(QueryBean queryBean, int page, int size);

}
```

```
package com.jackfrued.dao;

import java.io.Serializable;
import java.util.List;

import com.jackfrued.comm.QueryBean;
import com.jackfrued.comm.QueryResult;

/**
 * BaseDao的缺省适配器
 * @author 骆昊
 *
 * @param <E> 实体类型
 * @param <K> 实体标识字段的类型
 */
```

```
public abstract class BaseDaoAdapter<E, K extends Serializable> implements
        BaseDao<E, K> {

    @Override
    public K save(E entity) {
        return null;
    }

    @Override
    public void delete(E entity) {
    }

    @Override
    public boolean deleteById(K id) {
        E entity = findById(id);
        if(entity != null) {
            delete(entity);
            return true;
        }
        return false;
    }

    @Override
    public void update(E entity) {
    }

    @Override
    public E findById(K id) {
        return null;
    }

    @Override
    public E findById(K id, boolean lazy) {
        return null;
    }

    @Override
    public List<E> findAll() {
        return null;
    }

    @Override
    public QueryResult<E> findByPage(int page, int size) {
        return null;
    }

    @Override
    public QueryResult<E> findByPage(QueryBean queryBean, int page, int size) {
        return null;
    }

}
```

```
package com.jackfrued.dao;

import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
```

```java
import org.springframework.beans.factory.annotation.Autowired;

import com.jackfrued.comm.HQLQueryBean;
import com.jackfrued.comm.QueryBean;
import com.jackfrued.comm.QueryResult;

/**
 * 基于Hibernate的BaseDao实现类
 * @author 骆昊
 *
 * @param <E> 实体类型
 * @param <K> 主键类型
 */
@SuppressWarnings(value = {"unchecked"})
public abstract class BaseDaoHibernateImpl<E, K extends Serializable> extends BaseDaoAdapter<E, K> {
    @Autowired
    protected SessionFactory sessionFactory;

    private Class<?> entityClass;       // 业务实体的类对象
    private String entityName;          // 业务实体的名字

    public BaseDaoHibernateImpl() {
        ParameterizedType pt = (ParameterizedType) this.getClass().getGenericSuperclass();
        entityClass = (Class<?>) pt.getActualTypeArguments()[0];
        entityName = entityClass.getSimpleName();
    }

    @Override
    public K save(E entity) {
        return (K) sessionFactory.getCurrentSession().save(entity);
    }

    @Override
    public void delete(E entity) {
        sessionFactory.getCurrentSession().delete(entity);
    }

    @Override
    public void update(E entity) {
        sessionFactory.getCurrentSession().update(entity);
    }

    @Override
    public E findById(K id) {
        return findById(id, false);
    }

    @Override
    public E findById(K id, boolean lazy) {
        Session session = sessionFactory.getCurrentSession();
        return (E) (lazy? session.load(entityClass, id) : session.get(entityClass, id));
    }

    @Override
    public List<E> findAll() {
        return sessionFactory.getCurrentSession().createCriteria(entityClass).list();
    }

    @Override
    public QueryResult<E> findByPage(int page, int size) {
        return new QueryResult<E>(
            findByHQLAndPage("from " + entityName , page, size),
            getCountByHQL("select count(*) from " + entityName)
        );
```

```
76      }
77
78      @Override
79      public QueryResult<E> findByPage(QueryBean queryBean, int page, int size) {
80          if(queryBean instanceof HQLQueryBean) {
81              HQLQueryBean hqlQueryBean = (HQLQueryBean) queryBean;
82              return new QueryResult<E>(
83                  findByHQLAndPage(hqlQueryBean.getQueryString(), page, size, hqlQueryBean.getParameters(
84                  getCountByHQL(hqlQueryBean.getCountString(), hqlQueryBean.getParameters())
85              );
86          }
87          return null;
88      }
89
90      /**
91       * 根据HQL和可变参数列表进行查询
92       * @param hql 基于HQL的查询语句
93       * @param params 可变参数列表
94       * @return 持有查询结果的列表容器或空列表容器
95       */
96      protected List<E> findByHQL(String hql, Object... params) {
97          return this.findByHQL(hql, getParamList(params));
98      }
99
100     /**
101      * 根据HQL和参数列表进行查询
102      * @param hql 基于HQL的查询语句
103      * @param params 查询参数列表
104      * @return 持有查询结果的列表容器或空列表容器
105      */
106     protected List<E> findByHQL(String hql, List<Object> params) {
107         List<E> list = createQuery(hql, params).list();
108         return list != null && list.size() > 0 ? list : Collections.EMPTY_LIST;
109     }
110
111     /**
112      * 根据HQL和参数列表进行分页查询
113      * @param hql 基于HQL的查询语句
114      * @page 页码
115      * @size 页面大小
116      * @param params 可变参数列表
117      * @return 持有查询结果的列表容器或空列表容器
118      */
119     protected List<E> findByHQLAndPage(String hql, int page, int size, Object... params) {
120         return this.findByHQLAndPage(hql, page, size, getParamList(params));
121     }
122
123     /**
124      * 根据HQL和参数列表进行分页查询
125      * @param hql 基于HQL的查询语句
126      * @page 页码
127      * @size 页面大小
128      * @param params 查询参数列表
129      * @return 持有查询结果的列表容器或空列表容器
130      */
131     protected List<E> findByHQLAndPage(String hql, int page, int size, List<Object> params) {
132         List<E> list = createQuery(hql, params)
133                 .setFirstResult((page - 1) * size)
134                 .setMaxResults(size)
135                 .list();
136         return list != null && list.size() > 0 ? list : Collections.EMPTY_LIST;
137     }
138
139     /**
```

```
     * 查询满足条件的记录数
     * @param hql 基于HQL的查询语句
     * @param params 可变参数列表
     * @return 满足查询条件的总记录数
     */
    protected long getCountByHQL(String hql, Object... params) {
        return this.getCountByHQL(hql, getParamList(params));
    }

    /**
     * 查询满足条件的记录数
     * @param hql 基于HQL的查询语句
     * @param params 参数列表容器
     * @return 满足查询条件的总记录数
     */
    protected long getCountByHQL(String hql, List<Object> params) {
        return (Long) createQuery(hql, params).uniqueResult();
    }

    // 创建Hibernate查询对象(Query)
    private Query createQuery(String hql, List<Object> params) {
        Query query = sessionFactory.getCurrentSession().createQuery(hql);
        for(int i = 0; i < params.size(); i++) {
            query.setParameter(i, params.get(i));
        }
        return query;
    }

    // 将可变参数列表组装成列表容器
    private List<Object> getParamList(Object... params) {
        List<Object> paramList = new ArrayList<>();
        if(params != null) {
            for(int i = 0; i < params.length; i++) {
                paramList.add(params[i]);
            }
        }
        return paramList.size() == 0? Collections.EMPTY_LIST : paramList;
    }

}
```

```
package com.jackfrued.comm;

import java.util.List;

/**
 * 查询条件的接口
 * @author 骆昊
 *
 */
public interface QueryBean {

    /**
     * 添加排序字段
     * @param fieldName 用于排序的字段
     * @param asc 升序还是降序
     * @return 查询条件对象自身(方便级联编程)
     */
    public QueryBean addOrder(String fieldName, boolean asc);

    /**
     * 添加排序字段
     * @param available 是否添加此排序字段
     * @param fieldName 用于排序的字段
```

```
     * @param asc 升序还是降序
     * @return 查询条件对象自身(方便级联编程)
     */
    public QueryBean addOrder(boolean available, String fieldName, boolean asc);

    /**
     * 添加查询条件
     * @param condition 条件
     * @param params 替换掉条件中参数占位符的参数
     * @return 查询条件对象自身(方便级联编程)
     */
    public QueryBean addCondition(String condition, Object... params);

    /**
     * 添加查询条件
     * @param available 是否需要添加此条件
     * @param condition 条件
     * @param params 替换掉条件中参数占位符的参数
     * @return 查询条件对象自身(方便级联编程)
     */
    public QueryBean addCondition(boolean available, String condition, Object... params);

    /**
     * 获得查询语句
     * @return 查询语句
     */
    public String getQueryString();

    /**
     * 获取查询记录数的查询语句
     * @return 查询记录数的查询语句
     */
    public String getCountString();

    /**
     * 获得查询参数
     * @return 查询参数的列表容器
     */
    public List<Object> getParameters();
}
```

```
package com.jackfrued.comm;

import java.util.List;

/**
 * 查询结果
 * @author 骆昊
 *
 * @param <T> 泛型参数
 */
public class QueryResult<T> {
    private List<T> result;     // 持有查询结果的列表容器
    private long totalRecords;  // 查询到的总记录数

    /**
     * 构造器
     */
    public QueryResult() {
    }

    /**
     * 构造器
     * @param result 持有查询结果的列表容器
```

```
     * @param totalRecords 查询到的总记录数
     */
    public QueryResult(List<T> result, long totalRecords) {
        this.result = result;
        this.totalRecords = totalRecords;
    }

    public List<T> getResult() {
        return result;
    }

    public void setResult(List<T> result) {
        this.result = result;
    }

    public long getTotalRecords() {
        return totalRecords;
    }

    public void setTotalRecords(long totalRecords) {
        this.totalRecords = totalRecords;
    }
}
```

```
package com.jackfrued.dao;

import com.jackfrued.comm.QueryResult;
import com.jackfrued.entity.Dept;

/**
 * 部门数据访问对象接口
 * @author 骆昊
 *
 */
public interface DeptDao extends BaseDao<Dept, Integer> {

    /**
     * 分页查询顶级部门
     * @param page 页码
     * @param size 页码大小
     * @return 查询结果对象
     */
    public QueryResult<Dept> findTopDeptByPage(int page, int size);

}
```

```
package com.jackfrued.dao.impl;

import java.util.List;

import org.springframework.stereotype.Repository;

import com.jackfrued.comm.QueryResult;
import com.jackfrued.dao.BaseDaoHibernateImpl;
import com.jackfrued.dao.DeptDao;
import com.jackfrued.entity.Dept;

@Repository
public class DeptDaoImpl extends BaseDaoHibernateImpl<Dept, Integer> implements DeptDao {
    private static final String HQL_FIND_TOP_DEPT = " from Dept as d where d.superiorDept is null ";

    @Override
    public QueryResult<Dept> findTopDeptByPage(int page, int size) {
```

```
        List<Dept> list = findByHQLAndPage(HQL_FIND_TOP_DEPT, page, size);
        long totalRecords = getCountByHQL(" select count(*) " + HQL_FIND_TOP_DEPT);
        return new QueryResult<>(list, totalRecords);
    }

}
```

```
package com.jackfrued.comm;

import java.util.List;

/**
 * 分页器
 * @author 骆昊
 *
 * @param <T> 分页数据对象的类型
 */
public class PageBean<T> {
    private static final int DEFAUL_INIT_PAGE = 1;
    private static final int DEFAULT_PAGE_SIZE = 10;
    private static final int DEFAULT_PAGE_COUNT = 5;

    private List<T> data;           // 分页数据
    private PageRange pageRange;    // 页码范围
    private int totalPage;          // 总页数
    private int size;               // 页面大小
    private int currentPage;        // 当前页码
    private int pageCount;          // 页码数量

    /**
     * 构造器
     * @param currentPage 当前页码
     * @param size 页码大小
     * @param pageCount 页码数量
     */
    public PageBean(int currentPage, int size, int pageCount) {
        this.currentPage = currentPage > 0 ? currentPage : 1;
        this.size = size > 0 ? size : DEFAULT_PAGE_SIZE;
        this.pageCount = pageCount > 0 ? size : DEFAULT_PAGE_COUNT;
    }

    /**
     * 构造器
     * @param currentPage 当前页码
     * @param size 页码大小
     */
    public PageBean(int currentPage, int size) {
        this(currentPage, size, DEFAULT_PAGE_COUNT);
    }

    /**
     * 构造器
     * @param currentPage 当前页码
     */
    public PageBean(int currentPage) {
        this(currentPage, DEFAULT_PAGE_SIZE, DEFAULT_PAGE_COUNT);
    }

    /**
     * 构造器
     */
    public PageBean() {
        this(DEFAUL_INIT_PAGE, DEFAULT_PAGE_SIZE, DEFAULT_PAGE_COUNT);
    }
```

```
58
59      public List<T> getData() {
60          return data;
61      }
62
63      public int getStartPage() {
64          return pageRange != null ? pageRange.getStartPage() : 1;
65      }
66
67      public int getEndPage() {
68          return pageRange != null ? pageRange.getEndPage() : 1;
69      }
70
71      public long getTotalPage() {
72          return totalPage;
73      }
74
75      public int getSize() {
76          return size;
77      }
78
79      public int getCurrentPage() {
80          return currentPage;
81      }
82
83      /**
84       * 将查询结果转换为分页数据
85       * @param queryResult 查询结果对象
86       */
87      public void transferQueryResult(QueryResult<T> queryResult) {
88          long totalRecords = queryResult.getTotalRecords();
89
90          data = queryResult.getResult();
91          totalPage = (int) ((totalRecords + size - 1) / size);
92          totalPage = totalPage >= 0 ? totalPage : Integer.MAX_VALUE;
93          this.pageRange = new PageRange(pageCount, currentPage, totalPage);
94      }
95
96  }
```

```
1   package com.jackfrued.comm;
2
3   /**
4    * 页码范围
5    * @author 骆昊
6    *
7    */
8   public class PageRange {
9       private int startPage;  // 起始页码
10      private int endPage;    // 终止页码
11
12      /**
13       * 构造器
14       * @param pageCount 总共显示几个页码
15       * @param currentPage 当前页码
16       * @param totalPage 总页数
17       */
18      public PageRange(int pageCount, int currentPage, int totalPage) {
19          startPage = currentPage - (pageCount - 1) / 2;
20          endPage = currentPage + pageCount / 2;
21          if(startPage < 1) {
22              startPage = 1;
23              endPage = totalPage > pageCount ? pageCount : totalPage;
24          }
```

```
        if (endPage > totalPage) {
            endPage = totalPage;
            startPage = (endPage - pageCount > 0) ? endPage - pageCount + 1 : 1;
        }
    }

    /**
     * 获得起始页页码
     * @return 起始页页码
     */
    public int getStartPage() {
        return startPage;
    }

    /**
     * 获得终止页页码
     * @return 终止页页码
     */
    public int getEndPage() {
        return endPage;
    }

}
```

```
package com.jackfrued.biz;

import com.jackfrued.comm.PageBean;
import com.jackfrued.entity.Dept;

/**
 * 部门业务逻辑接口
 * @author 骆昊
 *
 */
public interface DeptService {

    /**
     * 创建新的部门
     * @param department 部门对象
     * @return 创建成功返回true否则返回false
     */
    public boolean createNewDepartment(Dept department);

    /**
     * 删除指定部门
     * @param id 要删除的部门的编号
     * @return 删除成功返回true否则返回false
     */
    public boolean deleteDepartment(Integer id);

    /**
     * 分页获取顶级部门
     * @param page 页码
     * @param size 页码大小
     * @return 部门对象的分页器对象
     */
    public PageBean<Dept> getTopDeptByPage(int page, int size);

}
```

```
package com.jackfrued.biz.impl;

import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.jackfrued.biz.DeptService;
import com.jackfrued.comm.PageBean;
import com.jackfrued.comm.QueryResult;
import com.jackfrued.dao.DeptDao;
import com.jackfrued.entity.Dept;

@Service
@Transactional  // 声明式事务的注解
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptDao deptDao;

    @Override
    public boolean createNewDepartment(Dept department) {
        return deptDao.save(department) != null;
    }

    @Override
    public boolean deleteDepartment(Integer id) {
        return deptDao.deleteById(id);
    }

    @Override
    public PageBean<Dept> getTopDeptByPage(int page, int size) {
        QueryResult<Dept> queryResult = deptDao.findTopDeptByPage(page, size);
        PageBean<Dept> pageBean = new PageBean<>(page, size);
        pageBean.transferQueryResult(queryResult);
        return pageBean;
    }

}
```

154、如何在Web项目中配置Spring的IoC容器?

答：如果需要在Web项目中使用Spring的IoC容器，可以在Web项目配置文件web.xml中做出如下配置：

```xml
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>

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

155、如何在Web项目中配置Spring MVC?

答：要使用Spring MVC需要在Web项目配置文件中配置其前端控制器DispatcherServlet，如下所示：

```xml
<web-app>

    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
```

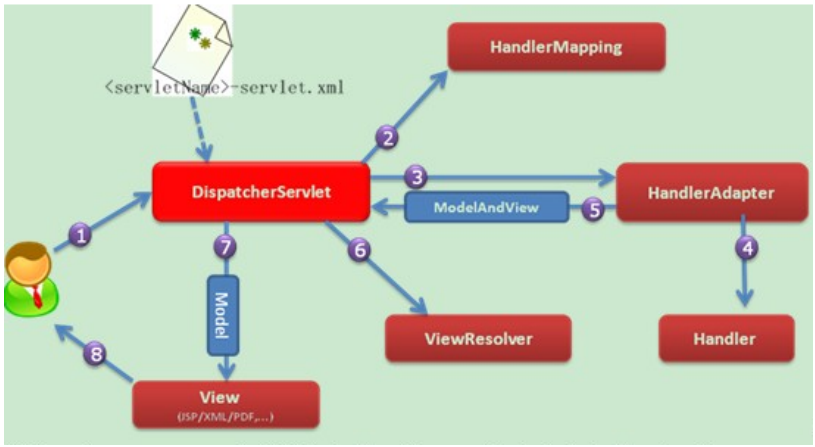
```
        <servlet-name>example</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

</web-app>
```

> **说明**：上面的配置中使用了*.html的后缀映射，这样做一方面不能够通过URL推断采用了何种服务器端的技术，另一方面可以欺骗搜索引擎，因为搜索引擎不会搜索动态页面，这种做法称为伪静态化。

156、Spring MVC的工作原理是怎样的?

答：Spring MVC的工作原理如下图所示：



① 客户端的所有请求都交给前端控制器DispatcherServlet来处理，它会负责调用系统的其他模块来真正处理用户的请求。
② DispatcherServlet收到请求后，将根据请求的信息（包括URL、HTTP协议方法、请求头、请求参数、Cookie等）以及HandlerMapping的配置找到处理该请求的Handler（任何一个对象都可以作为请求的Handler）。
③在这个地方Spring会通过HandlerAdapter对该处理器进行封装。
④ HandlerAdapter是一个适配器，它用统一的接口对各种Handler中的方法进行调用。
⑤ Handler完成对用户请求的处理后，会返回一个ModelAndView对象给DispatcherServlet，ModelAndView顾名思义，包含了数据模型以及相应的视图的信息。
⑥ ModelAndView的视图是逻辑视图，DispatcherServlet还要借助ViewResolver完成从逻辑视图到真实视图对象的解析工作。
⑦ 当得到真正的视图对象后，DispatcherServlet会利用视图对象对模型数据进行渲染。
⑧ 客户端得到响应，可能是一个普通的HTML页面，也可以是XML或JSON字符串，还可以是一张图片或者一个PDF文件。

157、如何在Spring IoC容器中配置数据源?

答：

DBCP配置：

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

C3P0配置：

```
<bean id="dataSource"
        class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
```

```
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

> 提示： DBCP的详细配置在第153题中已经完整的展示过了。

158、如何配置配置事务增强?

答：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the transactional advice -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true"/>
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*"/>
  </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
    of an operation defined by the FooService interface -->
  <aop:config>
  <aop:pointcut id="fooServiceOperation"
    expression="execution(* x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>

  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

159、选择使用Spring框架的原因（Spring框架为企业级开发带来的好处有哪些）？

答：可以从以下几个方面作答：

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

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

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

– MVC：Spring的MVC框架是非常优秀的，从各个方面都可以甩Struts 2几条街，为Web表示层提供了更好的解决方案。

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

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

160、Spring IoC容器配置Bean的方式?

答：

– 基于XML文件进行配置。

– 基于注解进行配置。

– 基于Java程序进行配置（Spring 3+）

```
package com.jackfrued.bean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Person {
    private String name;
    private int age;
    @Autowired
    private Car car;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";
    }

}
```

```
package com.jackfrued.bean;

import org.springframework.stereotype.Component;

@Component
public class Car {
    private String brand;
    private int maxSpeed;

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }
```

```
14
15      @Override
16      public String toString() {
17          return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
18      }
19
20  }
```

```
1   package com.jackfrued.config;
2
3   import org.springframework.context.annotation.Bean;
4   import org.springframework.context.annotation.Configuration;
5
6   import com.jackfrued.bean.Car;
7   import com.jackfrued.bean.Person;
8
9   @Configuration
10  public class AppConfig {
11
12      @Bean
13      public Car car() {
14          return new Car("Benz", 320);
15      }
16
17      @Bean
18      public Person person() {
19          return new Person("骆昊", 34);
20      }
21  }
```

```
1   package com.jackfrued.test;
2
3   import org.springframework.context.ConfigurableApplicationContext;
4   import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5
6   import com.jackfrued.bean.Person;
7   import com.jackfrued.config.AppConfig;
8
9   class Test {
10
11      public static void main(String[] args) {
12          // TWR (Java 7+)
13          try(ConfigurableApplicationContext factory = new AnnotationConfigApplicationContext(AppConfig.c
14              Person person = factory.getBean(Person.class);
15              System.out.println(person);
16          }
17      }
18  }
```

161、阐述Spring框架中Bean的生命周期?

答:

① Spring IoC容器找到关于Bean的定义并实例化该Bean。

② Spring IoC容器对Bean进行依赖注入。

③ 如果Bean实现了BeanNameAware接口，则将该Bean的id传给setBeanName方法。

④ 如果Bean实现了BeanFactoryAware接口，则将BeanFactory对象传给setBeanFactory方法。

⑤ 如果Bean实现了BeanPostProcessor接口，则调用其postProcessBeforeInitialization方法。

⑥ 如果Bean实现了InitializingBean接口，则调用其afterPropertySet方法。

⑦ 如果有和Bean关联的BeanPostProcessors对象，则这些对象的postProcessAfterInitialization方法被调用。

⑧ 当销毁Bean实例时，如果Bean实现了DisposableBean接口，则调用其destroy方法。

162、依赖注入时如何注入集合属性?

答：可以在定义Bean属性时，通过<list> / <set> / <map> / <props>分别为其注入列表、集合、映射和键值都是字符串的映射属性。

163、Spring中的自动装配有哪些限制?

答：

– 如果使用了构造器注入或者setter注入，那么将覆盖自动装配的依赖关系。

– 基本数据类型的值、字符串字面量、类字面量无法使用自动装配来注入。

– 优先考虑使用显式的装配来进行更精确的依赖注入而不是使用自动装配。

164、在Web项目中如何获得Spring的IoC容器?

答：

```
WebApplicationContext ctx =
WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

165. 大型网站在架构上应当考虑哪些问题?

答：

– 分层：分层是处理任何复杂系统最常见的手段之一，将系统横向切分成若干个层面，每个层面只承担单一的职责，然后通过下层为上层提供的基础设施和服务以及上层对下层的调用来形成一个完整的复杂的系统。计算机网络的开放系统互联参考模型（OSI/RM）和Internet的TCP/IP模型都是分层结构，大型网站的软件系统也可以使用分层的理念将其分为持久层（提供数据存储和访问服务）、业务层（处理业务逻辑，系统中最核心的部分）和表示层（系统交互、视图展示）。需要指出的是：（1）分层是逻辑上的划分，在物理上可以位于同一设备上也可以在不同的设备上部署不同的功能模块，这样可以使用更多的计算资源来应对用户的并发访问；（2）层与层之间应当有清晰的边界，这样分层才有意义，才更利于软件的开发和维护。

– 分割：分割是对软件的纵向切分。我们可以将大型网站的不同功能和服务分割开，形成高内聚低耦合的功能模块（单元）。在设计初期可以做一个粗粒度的分割，将网站分割为若干个功能模块，后期还可以进一步对每个模块进行细粒度的分割，这样一方面有助于软件的开发和维护，另一方面有助于分布式的部署，提供网站的并发处理能力和功能的扩展。

– 分布式：除了上面提到的内容，网站的静态资源（JavaScript、CSS、图片等）也可以采用独立分布式部署并采用独立的域名，这样可以减轻应用服务器的负载压力，也使得浏览器对资源的加载更快。数据的存取也应该是分布式的，传统的商业级关系型数据库产品基本上都支持分布式部署，而新生的NoSQL产品几乎都是分布式的。当然，网站后台的业务处理也要使用分布式技术，例如查询索引的构建、数据分析等，这些业务计算规模庞大，可以使用Hadoop以及MapReduce分布式计算框架来处理。

– 集群：集群使得有更多的服务器提供相同的服务，可以更好的提供对并发的支持。

– 缓存：所谓缓存就是用空间换取时间的技术，将数据尽可能放在距离计算最近的位置。使用缓存是网站优化的第一定律。我们通常说的CDN、反向代理、热点数据都是对缓存技术的使用。

– 异步：异步是实现软件实体之间解耦合的又一重要手段。异步架构是典型的生产者消费者模式，二者之间没有直接的调用关系，只要保持数据结构不变，彼此功能实现可以随意变化而不互相影响，这对网站的扩展非常有利。使用异步处理还可以提高系统可用性，加快网站的响应速度（用Ajax加载数据就是一种异步技术），同时还可以起到削峰作用（应对瞬时高并发）。&quot；能推迟处理的都要推迟处理"是网站优化的第二定律，而异步是践行网站优化第二定律的重要手段。

– 冗余：各种服务器都要提供相应的冗余服务器以便在某台或某些服务器宕机时还能保证网站可以正常工作，同时也提供了灾难恢复的可能性。冗余是网站高可用性的重要保证。

166、你用过的网站前端优化的技术有哪些?

答：

① 浏览器访问优化：

– 减少HTTP请求数量：合并CSS、合并JavaScript、合并图片（CSS Sprite）

– 使用浏览器缓存：通过设置HTTP响应头中的Cache-Control和Expires属性，将CSS、JavaScript、图片等在浏览器中缓存，当这些静态资源需要更新时，可以更新HTML文件中的引用来让浏览器重新请求新的资源

– 启用压缩

– CSS前置，JavaScript后置

– 减少Cookie传输

② CDN加速：CDN（Content Distribute Network）的本质仍然是缓存，将数据缓存在离用户最近的地方，CDN通常部署在网络运营商的机房，不仅可以提升响应速度，还可以减少应用服务器的压力。当然，CDN缓存的通常都是静态资源。

③ 反向代理：反向代理相当于应用服务器的一个门面，可以保护网站的安全性，也可以实现负载均衡的功能，当然最重要的是它缓存了用户访问的热点资源，可以直接从反向代理将某些内容返回给用户浏览器。

167、你使用过的应用服务器优化技术有哪些?

答:

① 分布式缓存:缓存的本质就是内存中的哈希表,如果设计一个优质的哈希函数,那么理论上哈希表读写的渐近时间复杂度为O(1)。缓存主要用来存放那些读写比很高、变化很少的数据,这样应用程序读取数据时先到缓存中读取,如果没有或者数据已经失效再去访问数据库或文件系统,并根据拟定的规则将数据写入缓存。对网站数据的访问也符合二八定律(Pareto分布,幂律分布),即80%的访问都集中在20%的数据上,如果能够将这20%的数据缓存起来,那么系统的性能将得到显著的改善。当然,使用缓存需要解决以下几个问题:

– 频繁修改的数据;

– 数据不一致与脏读;

– 缓存雪崩(可以采用分布式缓存服务器集群加以解决,memcached是广泛采用的解决方案);

– 缓存预热;

– 缓存穿透(恶意持续请求不存在的数据)。

② 异步操作:可以使用消息队列将调用异步化,通过异步处理将短时间高并发产生的事件消息存储在消息队列中,从而起到削峰作用。电商网站在进行促销活动时,可以将用户的订单请求存入消息队列,这样可以抵御大量的并发订单请求对系统和数据库的冲击。目前,绝大多数的电商网站即便不进行促销活动,订单系统都采用了消息队列来处理。

③ 使用集群。

④ 代码优化:

– 多线程:基于Java的Web开发基本上都通过多线程的方式响应用户的并发请求,使用多线程技术在编程上要解决线程安全问题,主要可以考虑以下几个方面:A. 将对象设计为无状态对象(这和面向对象的编程观点是矛盾的,在面向对象的世界中被视为不良设计),这样就不会存在并发访问时对象状态不一致的问题。B. 在方法内部创建对象,这样对象由进入方法的线程创建,不会出现多个线程访问同一对象的问题。使用ThreadLocal将对象与线程绑定也是很好的做法,这一点在前面已经探讨过了。C. 对资源进行并发访问时应当使用合理的锁机制。

– 非阻塞I/O: 使用单线程和非阻塞I/O是目前公认的比多线程的方式更能充分发挥服务器性能的应用模式,基于Node.js构建的服务器就采用了这样的方式。Java在JDK 1.4中就引入了NIO(Non-blocking I/O),在Servlet 3规范中又引入了异步Servlet的概念,这些都为在服务器端采用非阻塞I/O提供了必要的基础。

– 资源复用:资源复用主要有两种方式,一是单例,二是对象池,我们使用的数据库连接池、线程池都是对象池化技术,这是典型的用空间换取时间的策略,另一方面也实现对资源的复用,从而避免了不必要的创建和释放资源所带来的开销。

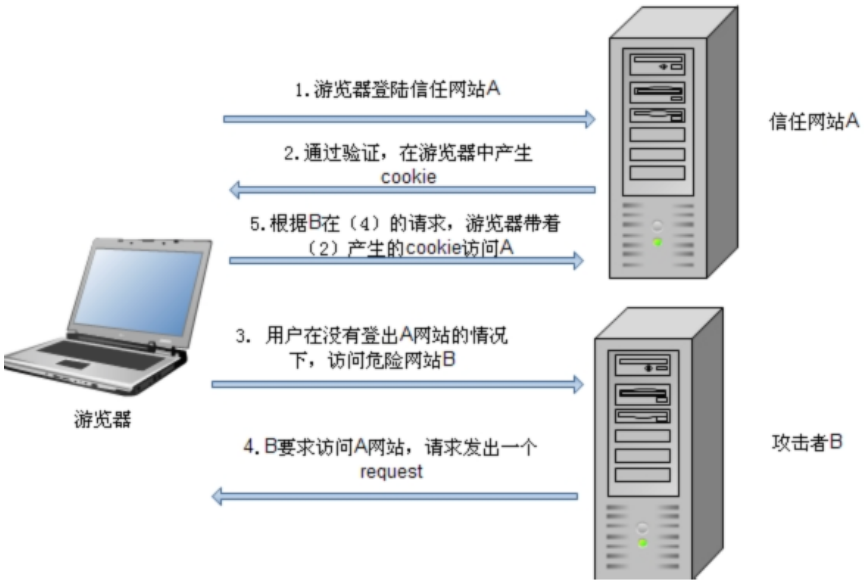
168、什么是XSS攻击?什么是SQL注入攻击?什么是CSRF攻击?

答:

– XSS(Cross Site Script,跨站脚本攻击)是向网页中注入恶意脚本在用户浏览网页时在用户浏览器中执行恶意脚本的攻击方式。跨站脚本攻击分有两种形式:反射型攻击(诱使用户点击一个嵌入恶意脚本的链接以达到攻击的目标,目前有很多攻击者利用论坛、微博发布含有恶意脚本的URL就属于这种方式)和持久型攻击(将恶意脚本提交到被攻击网站的数据库中,用户浏览网页时,恶意脚本从数据库中被加载到页面执行,QQ邮箱的早期版本就曾经被利用作为持久型跨站脚本攻击的平台)。XSS虽然不是什么新鲜玩意,但是攻击的手法却不断翻新,防范XSS主要有两方面:消毒(对危险字符进行转义)和HttpOnly(防范XSS攻击者窃取Cookie数据)。

– SQL注入攻击是注入攻击最常见的形式(此外还有OS注入攻击(Struts 2的高危漏洞就是通过OGNL实施OS注入攻击导致的)),当服务器使用请求参数构造SQL语句时,恶意的SQL被嵌入到SQL中交给数据库执行。SQL注入攻击需要攻击者对数据库结构有所了解才能进行,攻击者想要获得表结构有多种方式:(1)如果使用开源系统搭建网站,数据库结构也是公开的(目前有很多现成的系统可以直接搭建论坛,电商网站,虽然方便快捷但是风险是必须要认真评估的);(2)错误回显(如果将服务器的错误信息直接显示在页面上,攻击者可以通过非法参数引发页面错误从而通过错误信息了解数据库结构,Web应用应当设置友好的错误页,一方面符合最小惊讶原则,一方面屏蔽掉可能给系统带来危险的错误回显信息);(3)盲注。防范SQL注入攻击也可以采用消毒的方式,通过正则表达式对请求参数进行验证,此外,参数绑定也是很好的手段,这样恶意的SQL会被当做SQL的参数而不是命令被执行,JDBC中的PreparedStatement就是支持参数绑定的语句对象,从性能和安全性上都明显优于Statement。

– CSRF攻击(Cross Site Request Forgery,跨站请求伪造)是攻击者通过跨站请求,以合法的用户身份进行非法操作(如转账或发帖等)。CSRF的原理是利用浏览器的Cookie或服务器的Session,盗取用户身份,其原理如下图所示。防范CSRF的主要手段是识别请求者的身份,主要有以下几种方式:(1)在表单中添加令牌(token);(2)验证码;(3)检查请求头中的Referer(前面提到防图片盗链接也是用的这种方式)。令牌和验证都具有一次消费性的特征,因此在原理上一致的,但是验证码是一种糟糕的用户体验,不是必要的情况下不要轻易使用验证码,目前很多网站的做法是如果在短时间内多次提交一个表单未获得成功后才要求提供验证码,这样会获得较好的用户体验。

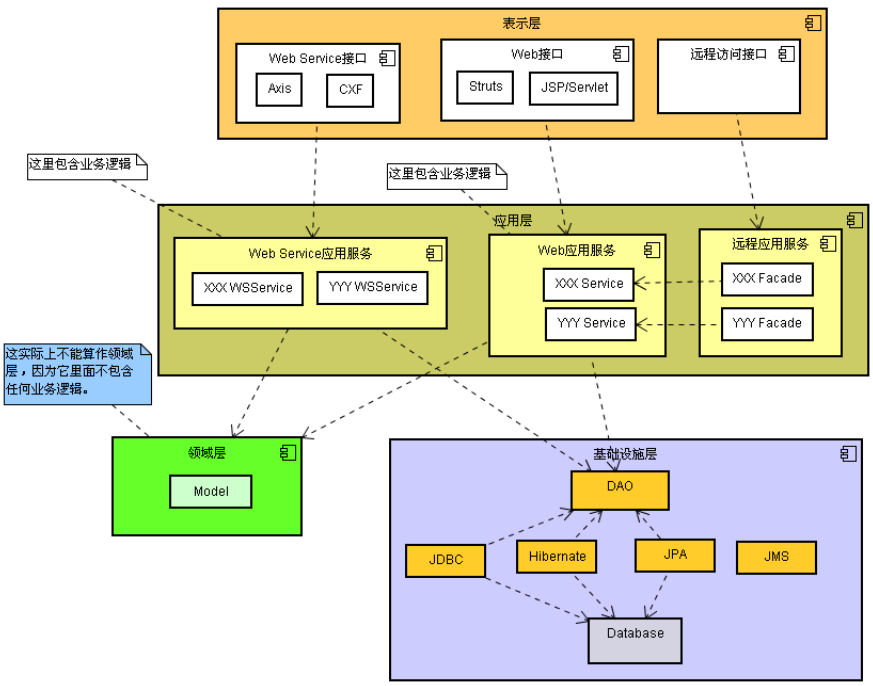


> 补充：防火墙的架设是Web安全的重要保障，ModSecurity是开源的Web防火墙中的佼佼者。企业级防火墙的架设应当有两级防火墙，Web服务器和部分应用服务器可以架设在两级防火墙之间的DMZ，而数据和资源服务器应当架设在第二级防火墙之后。

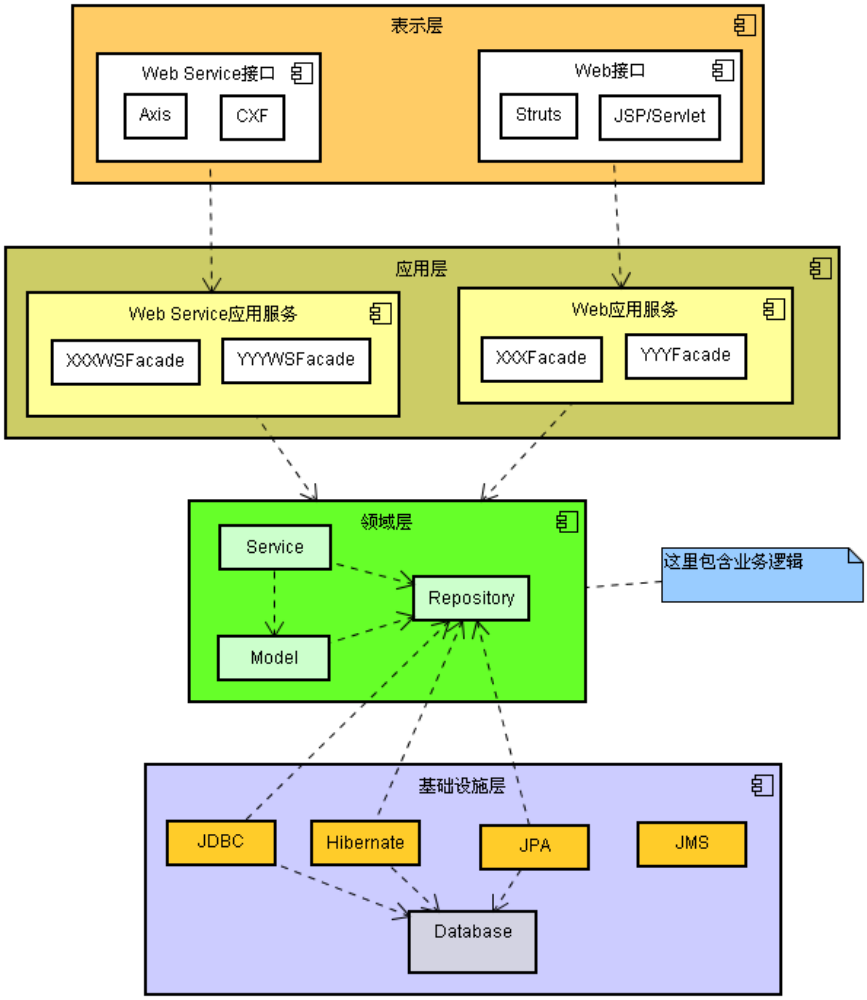
169. 什么是领域模型(domain model)？贫血模型(anaemic domain model)和充血模型(rich domain model)有什么区别？

答：领域模型是领域内的概念类或现实世界中对象的可视化表示，又称为概念模型或分析对象模型，它专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。贫血模型是指使用的领域对象中只有setter和getter方法（POJO），所有的业务逻辑都不包含在领域对象中而是放在业务逻辑层。有人将我们这里说的贫血模型进一步划分成失血模型（领域对象完全没有业务逻辑）和贫血模型（领域对象有少量的业务逻辑），我们这里就不对此加以区分了。充血模型将大多数业务逻辑和持久化放在领域对象中，业务逻辑（业务门面）只是完成对业务逻辑的封装、事务和权限等的处理。下面两张图分别展示了贫血模型和充血模型的分层架构。

贫血模型



充血模型



贫血模型下组织领域逻辑通常使用事务脚本模式，让每个过程对应用户可能要做的一个动作，每个动作由一个过程来驱动。也就是说在设计业务逻辑接口的时候，每个方法对应着用户的一个操作，这种模式有以下几个有点：
– 它是一个大多数开发者都能够理解的简单过程模型（适合国内的绝大多数开发者）。
– 它能够与一个使用行数据入口或表数据入口的简单数据访问层很好的协作。
– 事务边界的显而易见，一个事务开始于脚本的开始，终止于脚本的结束，很容易通过代理（或切面）实现声明式事务。
然而，事务脚本模式的缺点也是很多的，随着领域逻辑复杂性的增加，系统的复杂性将迅速增加，程序结构将变得极度混乱。开源中国社区上有一篇很好的译文《贫血领域模型是如何导致糟糕的软件产生》对这个问题做了比较细致的阐述。

170. 谈一谈测试驱动开发（TDD）的好处以及你的理解。
答：TDD是指在编写真正的功能实现代码之前先写测试代码，然后根据需要重构实现代码。在JUnit的作者Kent Beck的大作《测试驱动开发：实战与模式解析》（Test-Driven Development: by Example）一书中有这么一段内容：“消除恐惧和不确定性是编写测试驱动代码的重要原因”。因为编写代码时的恐惧会让你小心试探，让你回避沟通，让你羞于得到反馈，让你变得焦躁不安，而TDD是消除恐惧、让Java开发者更加自信更加乐于沟通的重要手段。TDD会带来的好处可能不会马上呈现，但是你在某个时候一定会发现，这些好处包括：
– 更清晰的代码 — 只写需要的代码
– 更好的设计
– 更出色的灵活性 — 鼓励程序员面向接口编程
– 更快速的反馈 — 不会到系统上线时才知道bug的存在

> 补充：敏捷软件开发的概念已经有很多年了，而且也部分的改变了软件开发这个行业，TDD也是敏捷开发所倡导的。

TDD可以在多个层级上应用，包括单元测试（测试一个类中的代码）、集成测试（测试类之间的交互）、系统测试（测试运行的系统）和系统集成测试（测试运行的系统包括使用的第三方组件）。TDD的实施步骤是：红（失败测试）– 绿（通过测试） – 重构。关于实施TDD的详细步骤请参考另一篇文章《测试驱动开发之初窥门径》。

在使用TDD开发时，经常会遇到需要被测对象需要依赖其他子系统的情况，但是你希望将测试代码跟依赖项隔离，以保证测试代码仅仅针对当前被测对象或方法展开，这时候你需要的是测试替身。测试替身可以分为四类：

– 虚设替身：只传递但是不会使用到的对象，一般用于填充方法的参数列表

– 存根替身：总是返回相同的预设响应，其中可能包括一些虚设状态

– 伪装替身：可以取代真实版本的可用版本（比真实版本还是会差很多）

– 模拟替身：可以表示一系列期望值的对象，并且可以提供预设响应

Java世界中实现模拟替身的第三方工具非常多，包括EasyMock、Mockito、jMock等。