java 高薪训练营 04 期

【Mybatis】

谈谈对 Mybatis 的一级、二级缓存的认识

1）一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置 <cache/> ；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

简述 Mybatis 的插件运行原理，以及如何编写一个插件。

Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 使用 JDK 的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，会拦截那些你指定需要拦截的方法。编写插件：实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，最后需要在核心配置文件中个配置插件,自定义的插件才会生效.

简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系？

Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中，<parameterMap>标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。<resultMap>标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个<select>、<insert>、<update>、<delete>标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。

Mybatis 是如何进行分页的？分页插件的原理是什么？

Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页。可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

MyBaits 相对于 JDBC 和 Hibernate 有哪些优缺点？

相对于 JDBC

优点：

1. 基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 SQL 与程序代码的耦合，便于统一管理；提供 XML

标签，支持编写动态 SQL 语句，并可重用；

2. 与 JDBC 相比，减少了代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；

3. 很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）；

4. 提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

缺点：

1. SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求；

2. SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

相对于 Hibernate

优点:

MyBatis 直接编写原生态 SQL，可以严格控制 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，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。

缺点:

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。

从以下几个方面谈谈对 mybatis 的一级缓存

1)mybaits 中如何维护一级缓存

2)一级缓存的生命周期

3)mybatis 一级缓存何时失效

4)一级缓存的工作流程

1)答案

BaseExecutor 成员变量之一的 PerpetualCache，是对 Cache 接口最基本的实现，其实现非常简单，内部持有 HashMap，对一级缓存的操作实则是对 HashMap 的操作。

2)答案

MyBatis 一级缓存的生命周期和 SqlSession 一致;

MyBatis 的一级缓存最大范围是 SqlSession 内部，有多个 SqlSession 或者分布式的环境下，数据库写操作会引起脏数据;

MyBatis 一级缓存内部设计简单，只是一个没有容量限定的 HashMap，在缓存的功能性上有所欠缺

3)答案

a. MyBatis 在开启一个数据库会话时，会创建一个新的 SqlSession 对象，SqlSession 对象中会有一个新的 Executor 对象，Executor 对象中持有一个新的 PerpetualCache 对象；当会话结束时，SqlSession 对象及其内部的 Executor 对象还有 PerpetualCache 对象也一并释放掉。

b. 如果 SqlSession 调用了 close()方法，会释放掉一级缓存 PerpetualCache 对象，一级缓存将不可用；

c. 如果 SqlSession 调用了 clearCache()，会清空 PerpetualCache 对象中的数据，但是该对象仍可使用；

d. SqlSession 中执行了任何一个 update 操作 update()、delete()、insert() ，都会清

空 PerpetualCache 对象的数据

4)答案

a.对于某个查询，根据 statementId,params,rowBounds 来构建一个 key 值，根据这个 key 值去缓存 Cache 中取出对应的 key 值存储的缓存结果；

b.判断从 Cache 中根据特定的 key 值取的数据数据是否为空，即是否命中；

c.如果命中，则直接将缓存结果返回；

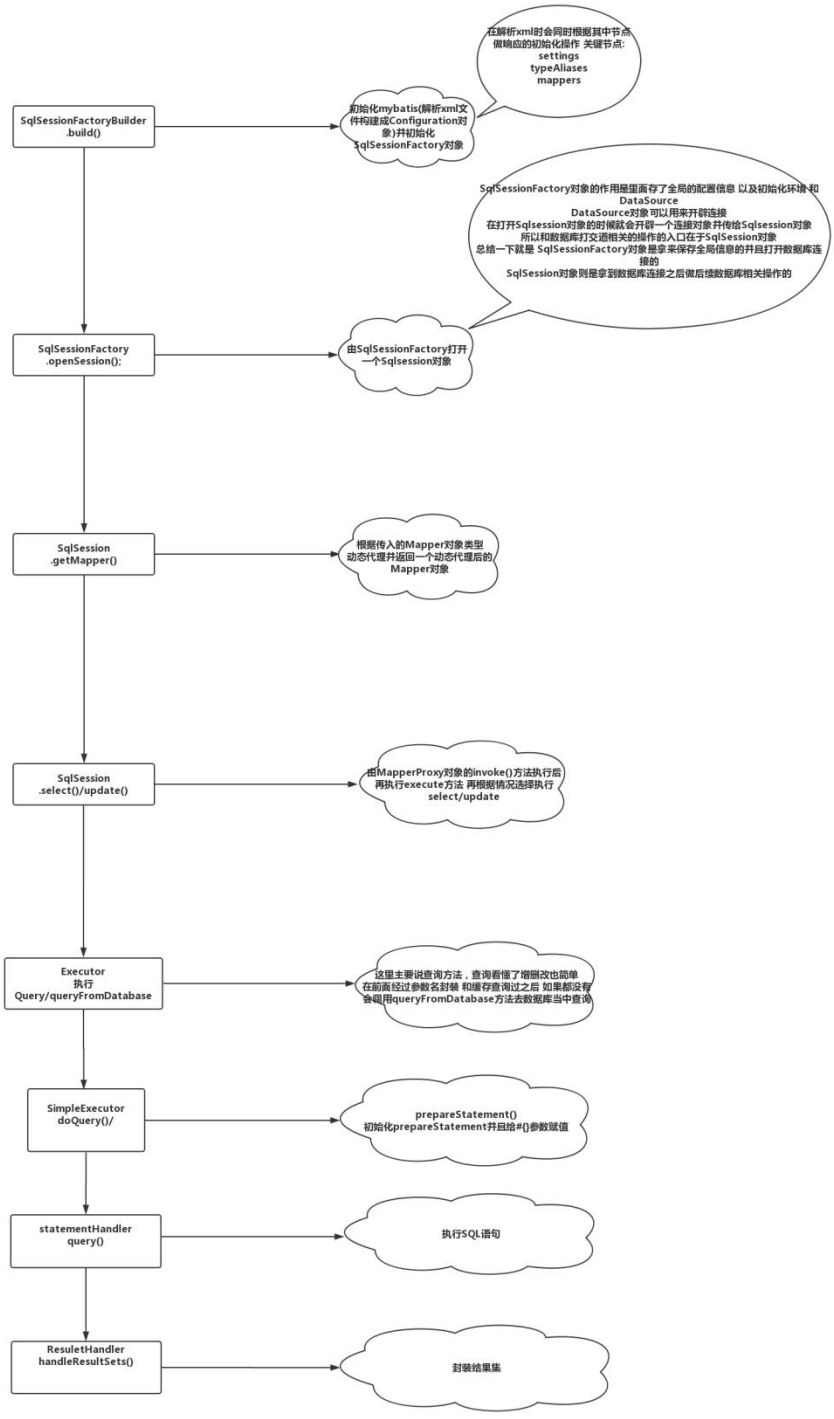
d.如果没命中：去数据库中查询数据，得到查询结果；将 key 和查询到的结果分别作为 key,value 对存储到 Cache 中；将查询结果返回.

Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能否定义在 A 标签的后面，还是说必须定义在 A 标签的前面？

答案：虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。

原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不存在，此时，Mybatis 会将 A 标签标记为未解析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也就可以正常解析完成了。

给面试官画一个 mybatis 的执行流程图



mybatis 中 xml 解析是通过 SqlSessionFactoryBuilder.build()方法。初始化 mybatis(解析 xml 文件构建成 Configuration 对象)并初始化 SqlSessionFactory 对象，在解析 xml 时会同时根据其中节点做相应的初始化操作

关键节点： settings、typeAliases、mappers

通过 SqlSesssionFactory.openSession()方法打开一个 SqlSession 对象

SqlSessionFactory 对象的作用是里面存了全局的配置信息以及初始化环境和 DataSource，DataSource 对象可以用来开辟连接

SqlSessionFactory 对象是用来保存全局信息并且打开数据库连接，在打开 SqlSession 对象的时候就会开辟一个连接对象并传给 SqlSession 对象，和数据库打交道的操作入口在于 SqlSession 对象

通过 SqlSession.getMapper()根据传入的 Mapper 对象类型动态代理并返回一个动态代理后的 Mapper 对象，

由 SqlSession.select()/update()，MapperProxy 对象的 invoke()方法执行后再执行 execure 方法，再根据情况选择执行 select/update

Executor 执行 Query/queryFromDatabase，在前面经过参数名封装和缓存查询之后（缓存为空），会调用 queryFromDatabase 方法去数据库当中查

SimpleExecurot 执行 doQuery()方法，初始化 prepareStatement 并且给#{}参数赋值

StatementHandler 执行 query()方法，执行 sql 语句

ResuletHandler.handleResultSets()方法封装结果集

【Spring】

请描述你对 Spring Bean 的生命周期的理解。

SpringBean 的生命周期指一个 Bean 对象从创建、到销毁的过程。SpringBean 不等于普通对象，实例化一个 java 对象只是 Bean 生命周期过程的一步，只有走完了流程，才称之为 SpringBean。核心过程如下：

（1）实例化 Bean：

主要通过反射技术，实例化 Java 对象

（2）设置对象属性（依赖注入）：

向实例化后的 Java 对象中注入属性

（3）处理 Aware 接口：

接着，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：

①如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值；

②如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory()方法，传递的是 Spring 工厂自身。

③如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文；

（4）BeanPostProcessor：

如果想对 Bean 进行一些自定义的处理，那么可以让 Bean 实现了 BeanPostProcessor 接口，那将会调用 postProcessBeforeInitialization(Object obj, String s)方法。

（5）InitializingBean 与 init-method：

实现接口 InitializingBean 完成一些初始化逻辑

如果 Bean 在 Spring 配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法，完成一些初始化逻辑。

（6）如果这个 Bean 实现了 BeanPostProcessor 接口，那将会调用 postProcessAfterInitialization(Object obj, String s)方法，在这个过程中比如可以做代理增强

（7）DisposableBean：

当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用其实现的 destroy()方法；

（8）destroy-method：

最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置

的销毁方法。

如何理解 IOC 和 DI，他们是什么关系。

追问 1：依赖注入有哪几种形式?

追问 2： Service Locator vs. Dependency Injection 有哪些不同

IoC 是一种设计模式，是一种思想，相当于一个容器，而 DI 就好比是实现 IOC 的一种方式。所谓依赖注入，就是由 IoC 容器在运行期间，动态地将某种依赖关系注入到对象之中。

构造器注入（Constructor Injection）：Ioc 容器会智能地选择选择和调用合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数，Ioc 容器在调用构造函数之前解析注册的依赖关系并自行获得相应参数对象；

属性注入（Property Injection）：如果需要使用到被依赖对象的某个属性，在被依赖对象被创建之后，Ioc 容器会自动初始化该属性；

方法注入（Method Injection）：如果被依赖对象需要调用某个方法进行相应的初始化，在该对象创建之后，Ioc 容器会自动调用该方法。

我们面临 Service Locator 和 Dependency Injection 之间的选择。应该注意，尽管我们前面那个简单的例子不足以表现出来，实际上这两个模式都提供了基本的解耦合能力。无论使用哪个模式，应用程序代码都不依赖于服务接口的具体实现。两者之间最重要的区别在于：具体实现以什么方式提供给应用程序代码。使用 Service Locator 模式时，应用程序代码直接向服务定位器发送一个消息，明确要求服务的实现；使用 Dependency Injection 模式时，应用程序代码不发出显式的请求，服务的实现自然会出现在应用程序代码中，这也就是所谓控制反转。

谈谈 Spring 中都用到了哪些设计模式？并举例说明。

工厂设计模式 : Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。

代理设计模式 : Spring 的 AOP 功能用到了 JDK 的动态代理和 CGLIB 字节码生成技术；

单例设计模式 : Spring 中的 Bean 默认都是单例的。

模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。

包装器设计模式 : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。

观察者模式: Spring 事件驱动模型就是观察者模式很经典的一个应用。

适配器模式 :Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller。

解释一下代理模式（Proxy）

代理模式： 代理模式就是本该我做的事，我不做，我交给代理人去完成。就比如，我生产了一些产品，我自己不卖，我委托代理商帮我卖，让代理商和顾客打交道，我自己负责主要产品的生产就可以了。 代理模式的使用，需要有本类，和代理类，本类和代理类共同实现统一的接口。然后在 main 中调用就可以了。本类中的业务逻辑一般是不会变动的，在我们需要的时候可以不断的添加代理对象，或者修改代理类来实现业务的变更。

代理模式可以分为： 静态代理 优点：可以做到在不修改目标对象功能的前提下，对目标功能扩展 缺点：因为本来和代理类要实现统一的接口，所以会产生很多的代理类，类太多，一旦接口增加方法，目标对象和代理对象都要维护。 动态代理（JDK 代理/接口代理）代理对象，不需要实现接口，代理对象的生成，是利用 JDK 的 API，动态的在内存中构建代

理对象，需要我们指定代理对象/目标对象实现的接口的类型。 Cglib 代理 特点: 在内存中构建一个子类对象，从而实现对目标对象功能的扩展。

使用场景： 修改代码的时候。不用随便去修改别人已经写好的代码，如果需要修改的话，可以通过代理的方式来扩展该方法。 隐藏某个类的时候，可以为其提供代理类 当我们要扩展某个类功能的时候，可以使用代理类 当一个类需要对不同的调用者提供不同的调用权限的时候，可以使用代理类来实现。 减少本类代码量的时候。 需要提升处理速度的时候。就比如我们在访问某个大型系统的时候，一次生成实例会耗费大量的时间，我们可以采用代理模式，当用来需要的时候才生成实例，这样就能提高访问的速度。

Spring 如何处理线程并发问题的？

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，因为 Spring 对一些 Bean 中非线程安全状态采用 ThreadLocal 进行处理，解决线程安全问题。

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。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而 ThreadLocal 采用了“空间换时间”的方式。

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。

【MVC】

描述一下 Spring MVC 请求处理流程

第一步：用户发送请求至前端控制器 DispatcherServlet

第二步： DispatcherServlet 收到请求调用HandlerMapping 处理器映射器

第三步：处理器映射器根据请求 Url 找到具体的 Handler（后端控制器），生成处理器对象及处理器拦截

器(如果 有则生成)一并返回 DispatcherServlet

第四步： DispatcherServlet 调用HandlerAdapter 处理器适配器去调用Handler

第五步：处理器适配器执行Handler

第六步： Handler 执行完成给处理器适配器返回 ModelAndView

第七步：处理器适配器向前端控制器返回 ModelAndView， ModelAndView 是 SpringMVC 框架的一个

底层对 象，包括 Model 和 View

第八步：前端控制器请求视图解析器去进行视图解析，根据逻辑视图名来解析真正的视图。

第九步：视图解析器向前端控制器返回 View

第十步：前端控制器进行视图渲染，就是将模型数据（在 ModelAndView 对象中）填充到 request 域

第十一步：前端控制器向用户响应结果

Spring MVC 的主要组件有哪些？并简述。

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了 DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器 HandlerMapping（不需要程序员开发）

作用：根据请求的 URL 来查找 Handler

（3）处理器适配器 HandlerAdapter

注意：在编写 Handler 的时候要按照 HandlerAdapter 要求的规则去编写，这样适配器 HandlerAdapter 才可以正确的去执行 Handler。

（4）处理器 Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图 View（需要程序员开发 jsp）

SpringMVC 中拦截器和多个拦截器的执行流程分别是怎样的？

单个：

1）程序先执行preHandle()方法，如果该方法的返回值为 true，则程序会继续向下执行处理器中的方

法，否则将不再向下执行。

2）在业务处理器（即控制器 Controller 类）处理完请求后，会执行postHandle()方法，然后会通过

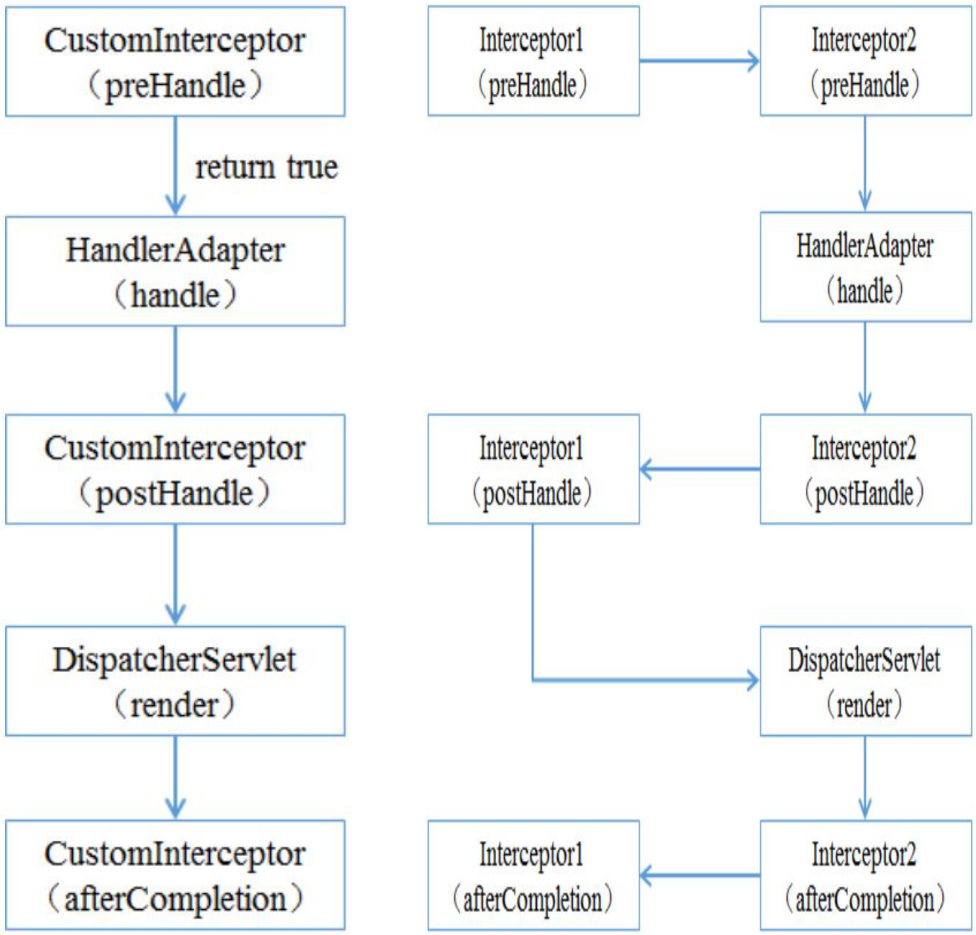
DispatcherServlet 向客户端返回响应。

3）在 DispatcherServlet 处理完请求后，才会执行afterCompletion()方法。

多个：

当有多个拦截器同时工作时，它们的 preHandle()方法会按照配置文件中拦截器的配置

顺序执行，而它们的 postHandle()方法和 afterCompletion()方法则会按照配置顺序的反序执行。



如何在 Spring 里注入 Java 集合?请举例?

Spring 提供了四种类型的配置元素集合,如下:

<list>:帮助注入一组值,允许重复。

<set>:帮助注入一组值,不允许重复。

< map>:帮助注入一个 K-V 的集合,名称和值可以是任何类型的。

<props>:帮助注入一个名称-值对集合，名称和值都是字符串。

```
<!-- java.util.List -->
<property name="customList">
  <list>
     <value>INDIA</value>
     <value>Pakistan</value>
     <value>USA</value>
     <value>UK</value>
  </list>
</property>

<!-- java.util.Set -->
<property name="customSet">
   <set>
      <value>INDIA</value>
      <value>Pakistan</value>
      <value>USA</value>
      <value>UK</value>
   </set>
 </property>

  <!-- java.util.Map -->
  <property name="customMap">

     <map>
        <entry key="1" value="INDIA"/>
        <entry key="2" value="Pakistan"/>
        <entry key="3" value="USA"/>
        <entry key="4" value="UK"/>
     </map>

   </property>

 <!-- java.util.Properties -->
<property name="customProperies">
   <props>
      <prop key="admin">admin@nospam.com</prop>
      <prop key="support">support@nospam.com</prop>
   </props>
</property>
```

简述注解原理？

注解本质是一个继承了 Annotation 的特殊接口，其具体实现类是 Java 运行时生成的动态代理类。我们通过反射获取注解时，返回的是 Java 运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用 AnnotationInvocationHandler 的 invoke 方法。该方法会从 memberValues 这个 Map 中索引出对应的值。而 memberValues 的来源是 Java 常量池。

描述事务的四大特性

原子性（Atomicity） 原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都

不发生。

从操作的角度来描述，事务中的各个操作要么都成功要么都失败

一致性（Consistency） 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

例如转账前 A 有 1000， B 有 1000。转账后 A+B 也得是 2000。

一致性是从数据的角度来说的，（1000， 1000） （900， 1100），不应该出现（900，1000）

隔离性（Isolation） 事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，

每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

比如：事务 1 给员工涨工资 2000，但是事务 1 尚未被提交，员工发起事务 2 查询工资，发现工资涨了 2000

块钱，读到了事务 1 尚未提交的数据（脏读）

持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障

也不应该对其有任何影响

Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景；

解释一下代理模式（Proxy）

代理模式： 代理模式就是本该我做的事，我不做，我交给代理人去完成。就比如，我生产了一些产品，我自己不卖，我委托代理商帮我卖，让代理商和顾客打交道，我自己负责主要产品的生产就可以了。 代理模式的使用，需要有本类，和代理类，本类和代理类共同实现统一的接口。然后在 main 中调用就可以了。本类中的业务逻辑一般是不会变动的，在我们需要的时候可以不断的添加代理对象，或者修改代理类来实现业务的变更。

代理模式可以分为： 静态代理 优点：可以做到在不修改目标对象功能的前提下，对目标功能扩展 缺点：因为本来和代理类要实现统一的接口，所以会产生很多的代理类，类太多，一旦接口增加方法，目标对象和代理对象都要维护。 动态代理（JDK 代理/接口代理）代理对象，不需要实现接口，代理对象的生成，是利用 JDK 的 API，动态的在内存中构建代

理对象，需要我们指定代理对象/目标对象实现的接口的类型。 Cglib 代理 特点: 在内存中构建一个子类对象，从而实现对目标对象功能的扩展。

使用场景： 修改代码的时候。不用随便去修改别人已经写好的代码，如果需要修改的话，可以通过代理的方式来扩展该方法。 隐藏某个类的时候，可以为其提供代理类 当我们要扩展某个类功能的时候，可以使用代理类 当一个类需要对不同的调用者提供不同的调用权限的时候，可以使用代理类来实现。 减少本类代码量的时候。 需要提升处理速度的时候。就比如我们在访问某个大型系统的时候，一次生成实例会耗费大量的时间，我们可以采用代理模式，当用来需要的时候才生成实例，这样就能提高访问的速度。

FileSystemResource 和 ClassPathResource 之间的区别是什么？

在 FileSystemResource 中你需要给出 spring-config.xml(Spring 配置)文件相对于您的项目的相对路径或文件的绝对位置。

在 ClassPathResource 中 Sping 查找文件使用 ClassPath，因此 spring-config.xml 应该包含在类路径下。

一句话,ClassPathResource 在类路径下搜索和 FileSystemResource 在文件系统下搜索。

Spring Boot 中如何解决跨域问题 ？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过（CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS ，现在则是通过 @CrossOrigin 注解来解决跨域问题。

说说 SpringBoot 自动配置原理

SpringBoot 自动配置主要是通过 @EnableAutoConfiguation,@Conditional,@EnableConfigProperties ， @ConfigurationProperties 等注解实现的。

具体流程如下：当我们写好一个启动类后，我们会在启动类上加一个 @SpringBootApplication, 我们可以点开这个注解可以看到它内部有一个 @EnableAutoConfiguation 的注解，我们继续进入这个注解可以看到一个@Import 的注解，这个注解引入了一个 AutoConfigurationImportSelector 的类。我们继续打开这个类可以看到它有一个 selectorImport 的方法，这个方法又调用了一个 getCandidateConfigurations 方法，这个方法内部通过 SpringFactoriesLoader.loadFactoryNames 最终调用 loadSpringFactories 加载到一个 META-INF 下的 spring.factories 文件。打开这个文件可以看到是一组一组的 key=value 的形式，其中一个 key 是 EnableAutoConfiguration 类的全类名，而它的 value 是一个 xxxxAutoConfiguration 的类名的列表，这些类名以逗号分隔。当我们通过 springApplication.run 启动的时候内部就会执行 selectImports 方法从而找到配置类对应的 class。然后将所有自动配置类加载到 Spring 容器中，进而实现自动配置。

谈谈 Tomcat 顶层架构

（1）Tomcat 中只有一个 Server，一个 Server 可以有多个 Service，一个 Service 可以有多个 Connector 和一个 Container；

（2） Server 掌管着整个 Tomcat 的生死大权；

（4）Service 是对外提供服务的；

（5）Connector 用于接受请求并将请求封装成 Request 和 Response 来具体处理；

（6）Container 用于封装和管理 Servlet，以及具体处理 request 请求；

知道了整个 Tomcat 顶层的分层架构和各个组件之间的关系以及作用，对于绝大多数的

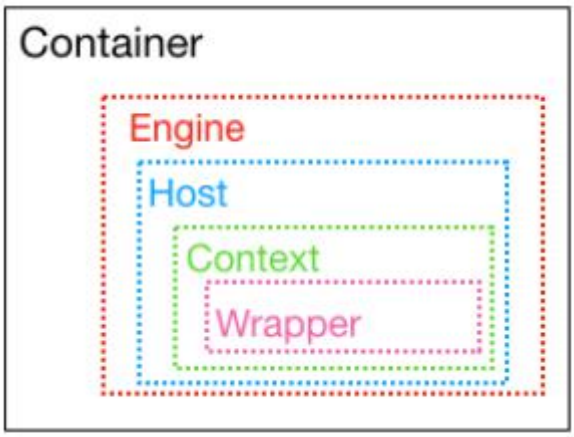
开发人员来说 Server 和 Service 对我们来说确实很远，而我们开发中绝大部分进行配置的内容是属于 Connector 和 Container 的。

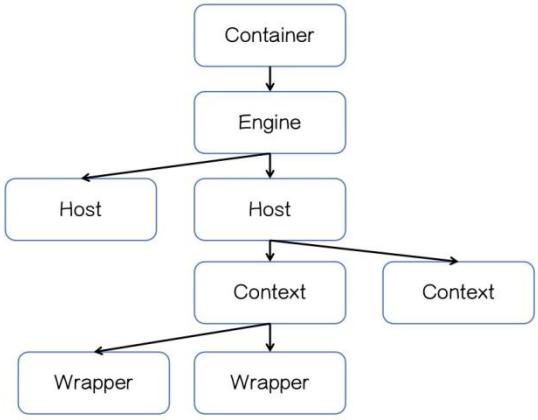
描述 Connector 和 Container 的关系

一个请求发送到 Tomcat 之后，首先经过 Service 然后会交给我们的 Connector，Connector 用于接收请求并将接收的请求封装为 Request 和 Response 来具体处理，Request 和 Response 封装完之后再交由 Container 进行处理，Container 处理完请求之后再返回给 Connector，最后在由 Connector 通过 Socket 将处理的结果返回给客户端，这样整个请求的就处理完了！

Connector 最底层使用的是 Socket 来进行连接的，Request 和 Response 是按照 HTTP 协议来封装的，所以 Connector 同时需要实现 TCP/IP 协议和 HTTP 协议！

Container 有哪些具体的组件，他们之间的关系是什么，并简要说明一下每个组件的职责





Container 组件下有几种具体的组件：

分别是 Engine、 Host、 Context 和 Wrapper。

这 4 种组件（容器）是父子关系。 Tomcat 通过一种分层的架构，使得 Servlet 容器具有很好的灵活性。

Engine

表示整个 Catalina 的 Servlet 引擎，用来管理多个虚拟站点，一个 Service 最多只能有一个 Engine，但是一个引擎可包含多个 Host

Host

代表一个虚拟主机，或者说一个站点，可以给 Tomcat 配置多个虚拟主机地址，而一个虚拟主机下可包含多个 Context

Context

表示一个 Web 应用程序， 一个 Web 应用可包含多个 Wrapper

Wrapper

表示一个 Servlet， Wrapper 作为容器中的最底层，不能包含子容器

上述组件的配置其实就体现在 conf/server.xml 中。

描述 BIO,NIO,AIO 有什么区别?

BIO (Blocking I/O): 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

NIO (New I/O): NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector，Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。 NIO 提供了与传统 BIO 模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

深度解析：https://blog.csdn.net/madongyu1259892936/article/details/79311671

Nginx 是如何处理一个请求的？

首先，nginx 在启动时，会解析配置文件，得到需要监听的端口与 ip 地址，然后在 nginx 的 master 进程里面先初始化好这个监控的 socket(创建 socket，设置 addrreuse 等选项，绑定到指定的 ip 地址端口，再 listen)

然后再 fork(一个现有进程可以调用 fork 函数创建一个新进程。由 fork 创建的新进程被称为子进程)出多个子进程出来

然后子进程会竞争 accept 新的连接。此时，客户端就可以向 nginx 发起连接了。当客户端与 nginx 进行三次握手，与 nginx 建立好一个连接后

此时，某一个子进程会 accept 成功，得到这个建立好的连接的 socket，然后创建 nginx 对连接的封装，即 ngx_connection_t 结构体

接着，设置读写事件处理函数并添加读写事件来与客户端进行数据的交换。最后，nginx 或客户端来主动关掉连接，到此，一个连接就寿终正寝了