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；

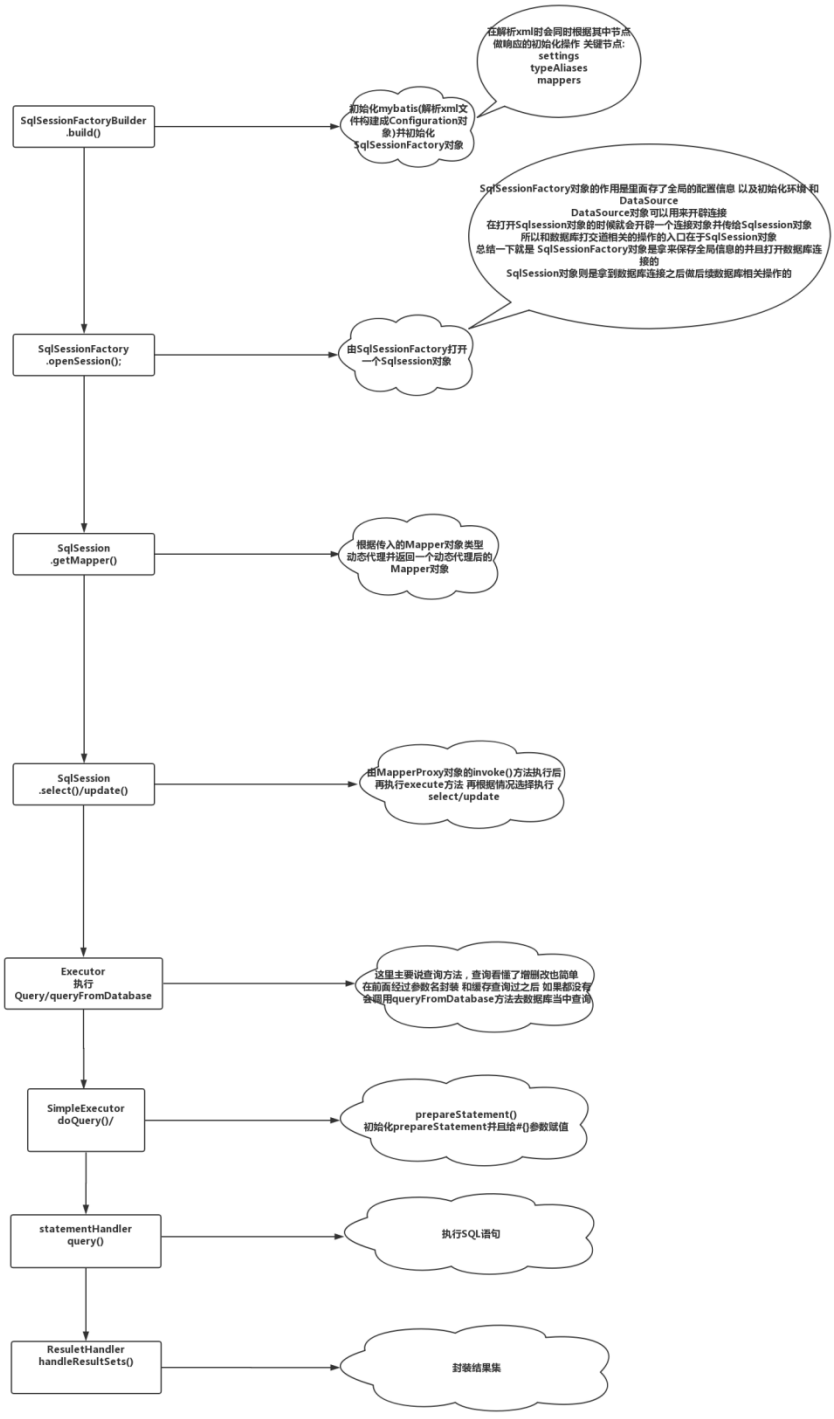
d.如果没命中：去数据库中查询数据，得到查询结果；将 key 和查询到的结果分别作为 key,value 对存储到 Cache 中；将查询结果返回.

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

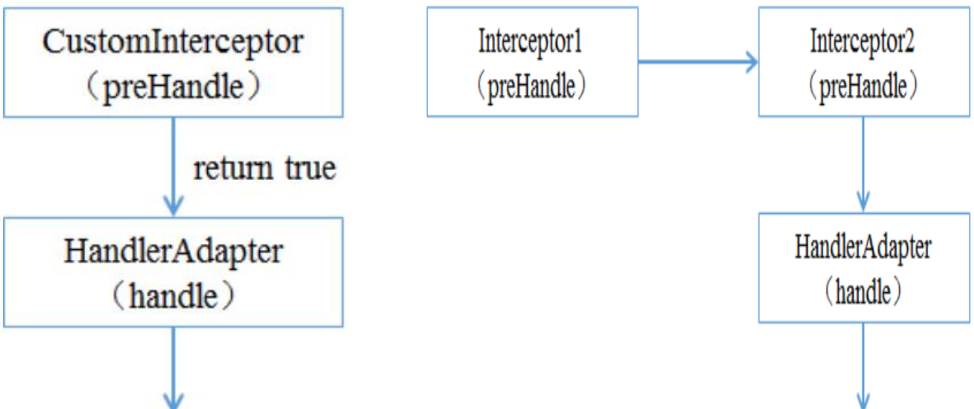
DispatcherServlet 向客户端返回响应。

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；

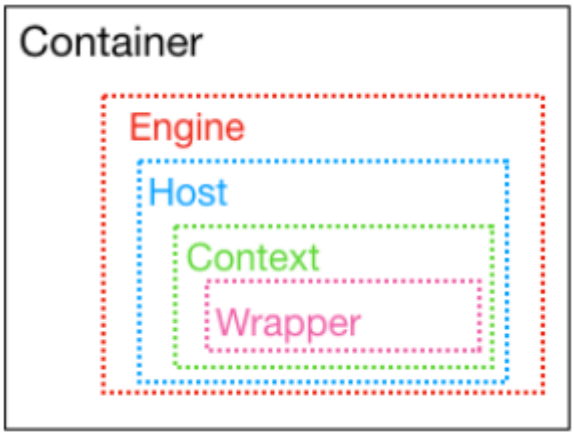
知道了整个 Tomcat 顶层的分层架构和各个组件之间的关系以及作用，对于绝大多数的开发人员来说 Server 和 Service 对我们来说确实很远，而我们开发中绝大部分进行配置的内容是属于 Connector 和 Container 的。

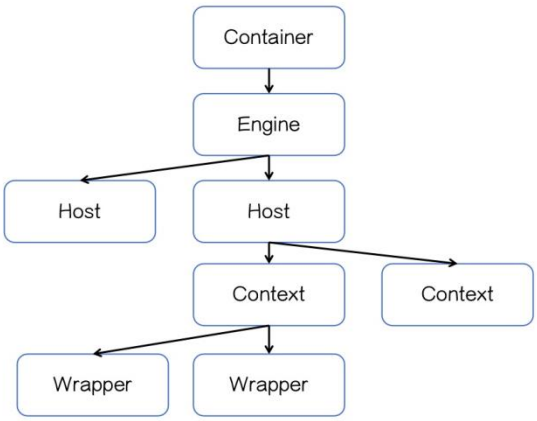
描述 Connector 和 Container 的关系

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

Tomcat 有几种部署方式？优点是什么？

tomcat 中四种部署项目的方法

第一种方法：

在 tomcat 中的 conf 目录中，在 server.xml 中的，<host/>节点中添加：

<Context path="/hello" docBase="D:/eclipse3.2.2/forwebtoolsworkspacehello/WebRoot" debug="0" privileged="true">

</Context>

第二种方法：

将 web 项目文件件拷贝到 webapps 目录中。

第三种方法：

很灵活，在 conf 目录中，新建 Catalina（注意大小写）\ localhost 目录，在该目录中新建一个 xml 文件，名字可以随意取，只要和当前文件中的文件名不重复就行了，该 xml 文件的内容为：

<Context path="/hello" docBase="D:eclipse3.2.2forwebtoolsworkspacehelloWebRoot" debug="0" privileged="true">

</Context>

第三种方法有个优点，可以定义别名。服务器端运行的项目名称为 path，外部访问的 URL 则使用 XML 的文件名。这个方法很方便的隐藏了项目的名称，对一些项目名称被固定不能更换，但外部访问时又想换个路径，非常有效。

第二、三种还有个优点，可以定义一些个性配置，如数据源的配置等。

Nginx 是如何实现高并发的？

ervice nginx start 之后，然后输入#ps -ef|grep nginx，会发现 Nginx 有一个 master 进程和若干个 worker 进程，这些 worker 进程是平等的，都是被 master fork 过来的。在 master 里面，先建立需要 listen 的 socket（listenfd），然后再 fork 出多个 worker 进程。当用户进入 nginx 服务的时候，每个 worker 的 listenfd 变的可读，并且这些 worker 会抢一个叫 accept_mutex 的东西，accept_mutex 是互斥的，一个 worker 得到了，其他的 worker 就歇菜了。而抢到这个 accept_mutex 的 worker 就开始“读取请求－解析请求－处理请求”，数据彻底返回客户端之后（目标网页出现在电脑屏幕上），这个事件就算彻底结束。

为什么要做动、静分离？

在我们的软件开发中，有些请求是需要后台处理的（如：.jsp,.do 等等），有些请求是不需要经过后台处理的（如：css、html、jpg、js 等等文件）

这些不需要经过后台处理的文件称为静态文件，否则动态文件。因此我们后台处理忽略静态文件。这会有人又说那我后台忽略静态文件不就完了吗

当然这是可以的，但是这样后台的请求次数就明显增多了。在我们对资源的响应速度有要求的时候，我们应该使用这种动静分离的策略去解决

动、静分离将网站静态资源（HTML，JavaScript，CSS，img 等文件）与后台应用分开部署，提高用户访问静态代码的速度，降低对后台应用访问

这里我们将静态资源放到 nginx 中，动态资源转发到 tomcat 服务器中

Nginx 的负载均衡算法都有哪些？

轮询(Round-Robin，RR)：默认情况下 Nginx 服务器实现负载均衡的算法就是轮询，轮询策略按照顺序选择组内服务器处理请求。如果一个服务器在处理请求的过程中出现错误，请求会被顺次交给组内的下一个服务器进行处理，以此类推，直到返回正常的响应为止。但如果所有的组内服务器都出错，则返回最后一个服务器的处理结果。

加权轮询(Weighted Round-Robin,WRR)：为组内服务器设置权重，权重值高的服务器被优先用于处理请求。此时组内服务器的选择策略为加权轮询。组内所有服务器的权重默认设

置为 1，即采用轮询处理请求。

ip_hash：ip_hash 用于实现会话保持功能，将某个客户端的多次请求定向到组内同一台服务器上，保证客户端与服务器之间建立稳定的会话。只有当服务器处于无效(down)的状态时，客户端请求才会被下一个服务器接收和处理。注意：使用 ip_hash 后不能使用 weight，ip_hash 和主要根据客户端 IP 地址分配服务器，因此在整个系统中，Nginx 服务器应该是处于最前端的服务器，这样才可以获取到客户端 IP 地址，否则它得到的 IP 地址将是位于它前面的服务器地址，从而就会产生问题。

最快响应时间（fair）：智能调度算法，动态地根据后端服务器的处理处理效率和响应时间对请求进行均衡分配，响应时间短、处理效率高的服务器分配到请求的概率高，响应时间长、处理效率低的服务器分配到请求的概率低。使用这种算法需要安装 upstream_fair 模块。

URL 绑定（url_hash）：按照访问的 URL 的 hash 结果分配请求，相同的 URL 会访问同一个后端服务器，在一定程度上可以提高缓存的效率。使用这种算法需要安装 Nginx 的 hash 软件包。

最小连接数（least_conn）：least_conn 用于为网络连接分配服务器组内的服务器，在功能上实现了最小连接数负载均衡算法，在选择组内的服务器时，考虑各服务器权重的同时，每次选择的都是当前网络连接最少的那台服务器，如果这样的服务器有多台，就采用加权轮询选择权重值大的服务器。

nginx 和 tomcat 的区别？

轻量级，同样起 web 服务，比 tomcat 占用更少的内存及资源

抗并发，nginx 处理请求是异步非阻塞的，而 tomcat 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能

高度模块化的设计，编写模块相对简单

最核心的区别在于 tomcat 是同步多进程模型，一个连接对应一个进程；nginx 是异步的，多个连接（万级别）可以对应一个进程。

描述 Spring AOP 实现原理及实现？

实现 AOP 的技术，主要分为两大类：

一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成，并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异，AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。

Spring AOP 使用动态代理技术在运行期织入增强代码。使用两种代理机制：基于 JDK 的动态代理（JDK 本身只提供接口的代理）和基于 CGlib 的动态代理。

(1) JDK 的动态代理

JDK 的动态代理主要涉及 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。其中 InvocationHandler 只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而 Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。

其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理,而不能针对类

(2)CGLib

CGLib 采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦

截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.所以无法通知 final、private 的方法,因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类,覆盖其中方法.

在 spring 中默认情况下使用 JDK 动态代理实现 AOP,如果 proxy-target-class 设置为 true 或者使用了优化策略那么会使用 CGLIB 来创建动态代理.Spring AOP 在这两种方式的实现上基本一样．以 JDK 代理为例，会使用 JdkDynamicAopProxy 来创建代理，在 invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入．最终返回代理对象。

JAVA 中有几种引用类型，他们的主要特点是什么。

四种。

1. 强引用

在 Java 中最常见的就是强引用， 把一个对象赋给一个引用变量，这个引用变量就是一个强引

用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即

使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之

一。

2. 软引用

软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它

不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。

3. 弱引用

弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象

来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。

4. 虚引用

虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。虚

引用的主要作用是跟踪对象被垃圾回收的状态。

Spring 最核心的功能是什么？使用 Spring 框架的最核心的原因是什么？

Spring 框架中核心组件有三个：Core、Context 和 Beans。其中最核心的组件就是 Beans，Spring 提供的最核心的功能就是 Bean Factory。

Spring 解决了的最核心的问题就是把对象之间的依赖关系转为用配置文件来管理，也就是 Spring 的依赖注入机制。这个注入机制是在 Ioc 容器中进行管理的。

Bean 组件是在 Spring 的 org.springframework.beans 包下。这个包主要解决了如下功能：Bean 的定义、Bean 的创建以及对 Bean 的解析。对 Spring 的使用者来说唯一需要关心的就是 Bean 的创建，其他两个由 Spring 内部机制完成。 Spring Bean 的创建采用典型的工厂模式，他的顶级接口是 BeanFactory。

BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和 AutowireCapableBeanFactory。但是从上图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory，他实现了所有的接口。那为何要定义这么多层次的接口呢？查

阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如 ListableBeanFactory 接口表示这些 Bean 是可列表的，而 HierarchicalBeanFactory 表示的是这些 Bean 是有继承关系的，也就是每个 Bean 有可能有父 Bean。AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。这四个接口共同定义了 Bean 的集合、Bean 之间的关系、以及 Bean 行为。

Bean 的定义就是完整的描述了在 Spring 的配置文件中你定义的 <bean/> 节点中所有的信息，包括各种子节点。当 Spring 成功解析你定义的一个 <bean/> 节点后，在 Spring 的内部他就被转化成 BeanDefinition 对象。以后所有的操作都是对这个对象完成的。Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean 的解析主要就是对 Spring 配置文件的解析。

Netty 组件有哪些，分别有什么关联？

Channel

基础的 IO 操作，如绑定、连接、读写等都依赖于底层网络传输所提供的原语，在 Java 的网络编程中，基础核心类是 Socket，而 Netty 的 Channel 提供了一组 API，极大地简化了直接与 Socket 进行操作的复杂性，并且 Channel 是很多类的父类，如 EmbeddedChannel、LocalServerChannel、NioDatagramChannel、NioSctpChannel、NioSocketChannel 等。

EventLoop

EventLoop 定义了处理在连接过程中发生的事件的核心抽象，之后会进一步讨论，其中 Channel、EventLoop、Thread 和 EventLoopGroup 之间的关系如下图所示：

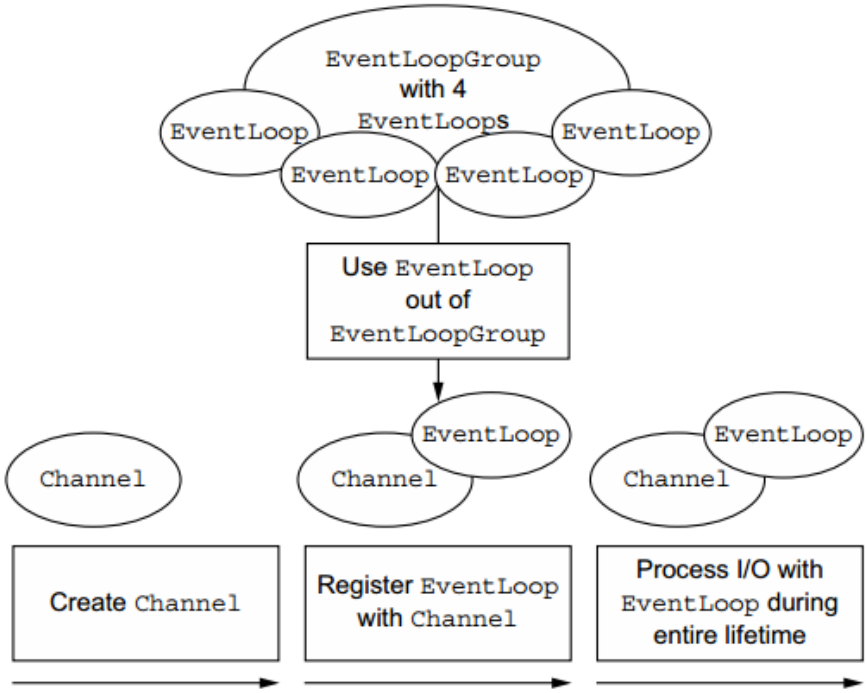
ChannelHandler 和 ChannelPipeline

从应用开发者看来，ChannelHandler 是最重要的组件，其中存放用来处理进站和出站数据的用户逻辑。ChannelHandler 的方法被网络事件触发，ChannelHandler 可以用于几乎任何类型的操作，如将数据从一种格式转换为另一种格式或处理抛出的异常。例如，其子接口 ChannelInboundHandler，接受进站的事件和数据以便被用户定义的逻辑处理，或者当响应所连接的客户端时刷新 ChannelInboundHandler 的数据。

ChannelPipeline 为 ChannelHandler 链提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API。当创建 Channel 时，会自动创建一个附属的 ChannelPipeline。

Bootstrap 和 ServerBootstrap

Netty 的引导类应用程序网络层配置提供容器，其涉及将进程绑定到给定端口或连接一个进程到在指定主机上指定端口上运行的另一进程。引导类分为客户端引导 Bootstrap 和服务端引导 ServerBootstrap。



Netty 高性能体现在哪些方面？

传输：IO 模型在很大程度上决定了框架的性能，相比于 bio，netty 建议采用异步通信模式，因为 nio 一个线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 IO 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。正如代码中所示，使用的是 NioEventLoopGroup 和 NioSocketChannel 来提升传输效率。

协议：采用什么样的通信协议，对系统的性能极其重要，netty 默认提供了对 Google Protobuf 的支持，也可以通过扩展 Netty 的编解码接口，用户可以实现其它的高性能序列化框架。

线程：netty 使用了 Reactor 线程模型，但 Reactor 模型不同，对性能的影响也非常大，下面介绍常用的 Reactor 线程模型有三种，分别如下：

Reactor 单线程模型：单线程模型的线程即作为 NIO 服务端接收客户端的 TCP 连接，又作为 NIO 客户端向服务端发起 TCP 连接，即读取通信对端的请求或者应答消息，又向通信对端发送消息请求或者应答消息。理论上一个线程可以独立处理所有 IO 相关的操作，但一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的编码、解码、读取和发送，又因为当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，NIO 线程会成为系统的性能瓶颈。

Reactor 多线程模型：有专门一个 NIO 线程用于监听服务端，接收客户端的 TCP 连接请求；网络 IO 操作(读写)由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池实现。但百万客户端并发连接时，一个 nio 线程用来监听和接受明显不够，因此有了主从多线程模型。

主从 Reactor 多线程模型：利用主从 NIO 线程模型，可以解决 1 个服务端监听线程无法有效处理所有客户端连接的性能不足问题，即把监听服务端，接收客户端的 TCP 连接请求分给一个线程池。因此，在代码中可以看到，我们在 server 端选择的就是这种方式，并且也推荐使用该线程模型。在启动类中创建不同的 EventLoopGroup 实例并通过适当的参数配置，就可以支持上述三种 Reactor 线程模型。

说说 Netty 的执行流程？

创建 ServerBootStrap 实例

设置并绑定 Reactor 线程池：EventLoopGroup，EventLoop 就是处理所有注册到本线程的 Selector 上面的 Channel

设置并绑定服务端的 channel

创建处理网络事件的 ChannelPipeline 和 handler，网络时间以流的形式在其中流转，handler 完成多数的功能定制：比如编解码 SSl 安全认证

绑定并启动监听端口

当轮训到准备就绪的 channel 后，由 Reactor 线程：NioEventLoop 执行 pipline 中的方法，最终调度并执行 channelHandler

RPC 需要解决的三个问题？

RPC 要达到的目标：远程调用时，要能够像本地调用一样方便，让调用者感知不到远程调用的逻辑。

Call ID 映射。我们怎么告诉远程机器我们要调用哪个函数呢？在本地调用中，函数体是直接通过函数指针来指定的，我们调用具体函数，编译器就自动帮我们调用它相应的函数指针。但是在远程调用中，是无法调用函数指针的，因为两个进程的地址空间是完全不一样。所以，在 RPC 中，所有的函数都必须有自己的一个 ID。这个 ID 在所有进程中都是唯一确定的。客户端在做远程过程调用时，必须附上这个 ID。然后我们还需要在客户端和服务端分别维护一个 {函数 <--> Call ID} 的对应表。两者的表不一定需要完全相同，但相同的函数对应的 Call ID 必须相同。当客户端需要进行远程调用时，它就查一下这个表，找出相应的 Call ID，然后把它传给服务端，服务端也通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码。

序列化和反序列化。客户端怎么把参数值传给远程的函数呢？在本地调用中，我们只需要把参数压到栈里，然后让函数自己去栈里读就行。但是在远程过程调用时，客户端跟服务端是不同的进程，不能通过内存来传递参数。甚至有时候客户端和服务端使用的都不是同一种语言（比如服务端用 C++，客户端用 Java 或者 Python）。这时候就需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式。这个过程叫序列化和反序列化。同理，从服务端返回的值也需要序列化反序列化的过程。

网络传输。远程调用往往是基于网络的，客户端和服务端是通过网络连接的。所有的数据都需要通过网络传输，因此就需要有一个网络传输层。网络传输层需要把 Call ID 和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。只要能完成这两者的，都可以作为传输层使用。因此，它所使用的协议其实是不限的，能完成传输就行。尽管大部分 RPC 框架都使用 TCP 协议，但其实 UDP 也可以，而 gRPC 干脆就用了 HTTP2。Java 的 Netty 也属于这层的东西。

主流 RPC 框架有哪些？各自的特点是什么？

1、RMI

利用 java.rmi 包实现，基于 Java 远程方法协议(Java Remote Method Protocol) 和 java 的原生序列化。

2、Hessian

是一个轻量级的 remoting onhttp 工具，使用简单的方法提供了 RMI 的功能。 基于 HTTP 协议，采用二进制编解码。

3、protobuf-rpc-pro

是一个 Java 类库，提供了基于 Google 的 Protocol Buffers 协议的远程方法调用的框架。基于 Netty 底层的 NIO 技术。支持 TCP 重用/ keep-alive、SSL 加密、RPC 调用取消操作、嵌入式日志等功能。

4、Thrift

是一种可伸缩的跨语言服务的软件框架。它拥有功能强大的代码生成引擎，无缝地支持 C++，C#，Java，Python 和 PHP 和 Ruby。thrift 允许你定义一个描述文件，描述数据类型和服务接口。依据该文件，编译器方便地生成 RPC 客户端和服务器通信代码。

最初由 facebook 开发用做系统内个语言之间的 RPC 通信，2007 年由 facebook 贡献到 apache 基金 ，现在是 apache 下的 opensource 之一 。支持多种语言之间的 RPC 方式的通信：php 语言 client 可以构造一个对象，调用相应的服务方法来调用 java 语言的服务，跨越语言的 C/S RPC 调用。底层通讯基于 SOCKET。

5、Avro

出自 Hadoop 之父 Doug Cutting，在 Thrift 已经相当流行的情况下推出 Avro 的目标不仅是提供一套类似 Thrift 的通讯中间件,更是要建立一个新的，标准性的云计算的数据交换和存储的 Protocol。支持 HTTP，TCP 两种协议。

6、Dubbo

Dubbo 是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring 框架无缝集成。

Netty 的零拷贝提体现在哪里，与操作系统上的有什么区别？

Zero-copy 就是在操作数据时，不需要将数据 buffer 从一个内存区域拷贝到另一个内存区域。 少了一次内存的拷贝,CPU 的效率就得到的提升。在 OS 层面上的 Zero-copy 通常指避免在 用户态(User-space) 与 内核态(Kernel-space)之间来回拷贝数据。Netty 的 Zero-copy 完全是在用户态(Java 层面)的，更多的偏向于优化数据操作。

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。

Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

请简述 zookeeper 投票机制，并讲述当 5 台服务器，均没有数据，编号分别是 1,2,3,4,5,按编号依次启动，它们的选举过程。

1. 每个 Server 启动以后都询问其它的 Server 它要投票给谁。对于其他 server 的询问，server 每次根据自己的状态都回复自己推荐的 leader 的 id 和上一次处理事务的 zxid （系统启动时每个 server 都会推荐自己）

2. 收到所有 Server 回复以后，就计算出 zxid 最大的哪个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。

3. 计算这过程中获得票数最多的的 sever 为获胜者，如果获胜者的票数超过半数，则改 server 被选为 leader。否则，继续这个过程，直到 leader 被选举出来

4. leader 就会开始等待 server 连接

5. Follower 连接 leader，将最大的 zxid 发送给 leader

6. Leader 根据 follower 的 zxid 确定同步点，至此选举阶段完成。

7. 选举阶段完成 Leader 同步后通知 follower 已经成为 uptodate 状态

8. Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了

它们的选择举过程如下：

1. 服务器 1 启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器 1 的状态一直属于 Looking。

2. 服务器 2 启动，给自己投票，同时与之前启动的服务器 1 交换结果，由于服务器 2 的编号大所以服务器 2 胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是 LOOKING。

3. 服务器 3 启动，给自己投票，同时与之前启动的服务器 1,2 交换信息，由于服务器 3 的编号最大所以服务器 3 胜出，此时投票数正好大于半数，所以服务器 3 成为领导者，服务器 1,2 成为小弟。

4. 服务器 4 启动，给自己投票，同时与之前启动的服务器 1,2,3 交换信息，尽管服务器 4 的编号大，但之前服务器 3 已经胜出，所以服务器 4 只能成为小弟。

5. 服务器 5 启动，后面的逻辑同服务器 4 成为小弟。

描述一下实现高可用 RPC 框架需要考虑到的问题

既然系统采用分布式架构，那一个服务势必会有多个实例，要解决如何获取实例的问题。所以需要一个服务注册中心，比如在 Dubbo 中，就可以使用 Zookeeper 作为注册中心，在调用时，从 Zookeeper 获取服务的实例列表，再从中选择一个进行调用；

如何选择实例呢？就要考虑负载均衡，例如 dubbo 提供了 4 种负载均衡策略；

如果每次都去注册中心查询列表，效率很低，那么就要加缓存；

客户端总不能每次调用完都等着服务端返回数据，所以就要支持异步调用；

服务端的接口修改了，老的接口还有人在用，这就需要版本控制；

服务端总不能每次接到请求都马上启动一个线程去处理，于是就需要线程池；

RPC 和 SOA、SOAP、REST 的区别

1、REST

可以看着是 HTTP 协议的一种直接应用，默认基于 JSON 作为传输格式,使用简单,学习成本低效率高,但是安全性较低。

2、SOAP

SOAP 是一种数据交换协议规范,是一种轻量的、简单的、基于 XML 的协议的规范。而 SOAP 可以看着是一个重量级的协议，基于 XML、SOAP 在安全方面是通过使用 XML-Security 和 XML-Signature 两个规范组成了 WS-Security 来实现安全控制的,当前已经得到了各个厂商的支持 。

它有什么优点？简单总结为：易用、灵活、跨语言、跨平台。

3、SOA

面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是 SOA 的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。

SOA 是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。SOA 可以看作是 B/S 模型、XML（标准通用标记语言的子集）/Web Service 技术之后的自然延伸。

4、REST 和 SOAP、RPC 有何区别呢?

没什么太大区别，他们的本质都是提供可支持分布式的基础服务，最大的区别在于他们各自

的的特点所带来的不同应用场景 。

什么情况下不会执行类初始化？

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化。
5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

注册中心挂了可以继续通信吗？

可以的，启动 dubbo 时，消费者会从 zk 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用

注册中心对等集群，任意一台宕掉后，会自动切换到另一台

注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯

服务提供者无状态，任一台 宕机后，不影响使用

服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

Dubbo 有些哪些注册中心？

Multicast 注册中心： Multicast 注册中心不需要任何中心节点，只要广播地 址，就能进行服务注册和发现。基于网络中组播传输实现；

Zookeeper 注册中心： 基于分布式协调系统 Zookeeper 实现，采用 Zookeeper 的 watch 机制实现数据变更；

redis 注册中心： 基于 redis 实现，采用 key/Map 存储，住 key 存储服务名 和类型，Map 中 key 存储服务 URL，value 服务过期时间。基于 redis 的发 布/订阅模式通知数据变更；

Simple 注册中心：注册中心本身就是一个普通的 Dubbo 服务，可以减少第三方依赖，使整体通讯方式一致。此 SimpleRegistryService 只是简单实现，不支持集群，可作为自定义注册中心的参考，但不适合直接用于生产环境。

Nacos 注册中心。

思维题

A 告诉 B 他生日的月份,告诉 C 他生日的日期

B 说：“如果我不知道 A 的生日,那 C 肯定也不知道."

C 说：”本来我不知道,现在我知道了.“

B 说：”哦,那我也知道了.“

A 的生日可能是：

11 月 4 日 11 月 5 日 11 月 8 日 1 月 4 日 1 月 22 日 3 月 1 日 3 月 5 日 7 月 1 日 7 月 2 日 7 月 8 日

请问 A 的生日是几月几日?

11 月 4 日 11 月 5 日 11 月 8 日
1 月 4 日 1 月 22 日
3 月 1 日 3 月 5 日
7 月 1 日 7 月 2 日 7 月 8 日
1 月与 7 月首先排除因为

B 说："如果我不知道 A 的生日,那 C 肯定也不知道."
可见,如果是 1 月或者 7 月,那么日期可能是 1 月 22 日、7 月 2 日
此时,知道日期的 C 是可以知道 A 的生日的.（因为 22 日与 2 日没有和其他日期重合）
剩余可选日期为
11 月 4 日 11 月 5 日 11 月 8 日
3 月 1 日 3 月 5 日
C 说："本来我不知道,现在我知道了."
排除 5 日,假如是 5 日,C 应该依然不能确定准确的生日.
故剩余日期为
11 月 4 日 11 月 8 日
3 月 1 日
B 说："哦,那我也知道了."
同理排除 11 月,假如是 11 月,则 B 应该依然不能确定准确生日.
综上所述为 3 月 1 日

说一下服务雪崩和服务熔断

服务雪崩效应产生于服务堆积在同一个线程池中，因为所有的请求都是同一个线程池进行处理，这时候如果在高并发情况下，所有的请求全部访问同一个接口，
这时候可能会导致其他服务没有线程进行接受请求，这就是服务雪崩效应。
服务熔断：当下游的服务因为某种原因突然变得不可用或响应过慢，上游服务为了保证自己整体服务的可用性，不再继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

服务降级中手动降级有哪些方法

(1)简化执行流程
自己梳理出核心业务流程和非核心业务流程。然后在非核心业务流程上加上开关，一旦发现系统扛不住，关掉开关，结束这些次要流程。
(2)关闭次要功能
一个微服务下肯定有很多功能，那自己区分出主要功能和次要功能。然后次要功能加上开关，需要降级的时候，把次要功能关了吧！
(3)降低一致性
假设，你在业务上发现执行流程没法简化了，愁啊！也没啥次要功能可以关了，桑心啊！那只能降低一致性了，即将核心业务流程的同步改异步，将强一致性改最终一致性！

SOA 服务和微服务区别

SOA	微服务架构
应用程序服务的可重用性的最大化	专注于解耦
系统性的改变需要修改整体	系统性的改变是创建一个新的服务
DevOps 和持续交付正在变得流行，但还不是主	强烈关注 DevOps 和持续交付

SOA	微服务架构
流	
专注于业务功能重用	更重视“上下文边界”的概念
通信使用企业服务总线 ESB	对于通信而言，使用较少精细和简单的消息系统
支持多种消息协议	使用轻量级协议，例如 HTTP，REST 或 Thrift API
对部署到它的所有服务使用通用平台	应用程序服务器不是真的被使用，通常使用云平台
容器（如 Docker）的使用不太受欢迎	容器在微服务方面效果很好
SOA 服务共享数据存储	每个微服务可以有一个独立的数据存储
共同的治理和标准	轻松的治理，更加关注团队协作和选择自由

下面进一步解释上表所述的不同之处：

- 开发方面 - 在这两种体系结构中，可以使用不同的编程语言和工具开发服务，从而将技术多样性带入开发团队。开发可以在多个团队中组织，但是在 SOA 中，每个团队都需要了解常见的通信机制。另一方面，使用微服务，服务可以独立于其他服务运行和部署。因此，频繁部署新版本的微服务或独立扩展服务会更容易。您可以在这里进一步阅读有关微服务的这些好处。

- “上下文边界” - SOA 鼓励组件的共享，而微服务尝试通过“上下文边界”来最小化共享。上下文边界是指以最小的依赖关系将组件及其数据耦合为单个单元。由于 SOA 依靠多个服务来完成业务请求，构建在 SOA 上的系统可能比微服务要慢。

- 通信 - 在 SOA 中，ESB 可能成为影响整个系统的单一故障点。由于每个服务都通过 ESB 进行通信，如果其中一个服务变慢，可能会阻塞 ESB 并请求该服务。另一方面，微服务在容错方面要好得多。例如，如果一个微服务存在内存错误，那么只有该微服务会受到影响。所有其他微服务将继续定期处理请求。

- 互操作性 - SOA 通过消息中间件组件促进了多种异构协议的使用。微服务试图通过减少集成选择的数量来简化架构模式。因此，如果您想要在异构环境中使用不同协议来集成多个系统，则需要考虑 SOA。如果您的所有服务都可以通过相同的远程访问协议访问，那么微服务对您来说是一个更好的选择。

- 大小 Size - 最后一点但并非最不重要的一点，SOA 和微服务的主要区别在于规模和范围。微服务架构中的前缀“微”是指内部组件的粒度，意味着它们必须比 SOA 架构的服务往往要小得多。微服务中的服务组件通常有一个单一的目的，他们做得很好。另一方面，在 SOA 服务中通常包含更多的业务功能，并且通常将它们实现为完整的子系统。

ribbon 和 feign 区别

Ribbon 添加 maven 依赖 spring-starter-ribbon 使用@RibbonClient(value="服务名称") 使用 RestTemplate 调用远程服务对应的方法
feign 添加 maven 依赖 spring-starter-feign 服务提供方提供对外接口 调用方使用 在接口上使用@FeignClient("指定服务名")
Ribbon 和 Feign 的区别：
Ribbon 和 Feign 都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon 用的是@RibbonClient，Feign 用的是@EnableFeignClients。
2.服务的指定位置不同，Ribbon 是在@RibbonClient 注解上声明，Feign 则是在定义抽象方法的接口中使用@FeignClient 声明。
3.调用方式不同，
Ribbon 需要自己构建 http 请求，模拟 http 请求然后使用 RestTemplate 发送给其他服务，步骤相当繁琐。
Feign 则是在 Ribbon 的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，
不需要自己构建 http 请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

zookeeper 如何保证主从节点状态同步？

Zookeeper 的核心是原子广播，这个机制保证了各个 Server 之间的同步。
实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。
恢复模式：当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。
因此，选主得到的 leader 保证了同步状态的进行，状态同步又保证了 leader 和 Server 具有相同的系统状态，当 leader 失去主权后可以在其他 follower 中选主新的 leader。

ZAB(Zookeeper Atomic Broadcast)协议是专门为 zookeeper 设计的一致性协议。
ZAB 协议包括两种基本的模式：消息广播和崩溃恢复

当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断崩溃退出与重启等异常情况时，ZAB 就会进入恢复模式并选举产生新的 Leader 服务器。
当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出崩溃恢复模式，进入消息广播模式。
当有新的服务器加入到集群中去，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播，那么新加入的服务器会自动进入数据恢复模式，找到 Leader 服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。
以上其实大致经历了三个步骤：

崩溃恢复：主要就是 Leader 选举过程。
数据同步：Leader 服务器与其他服务器进行数据同步。
消息广播：Leader 服务器将数据发送给其他服务器。

JVM 中类加载的过程?

1.1 加载
加载主要是将.class 文件（并不一定是.class。可以是 ZIP 包，网络中获取）中的二进制字节流读入到 JVM 中。
在加载阶段，JVM 需要完成 3 件事：
1）通过类的全限定名获取该类的二进制字节流；
2）将字节流所代表的静态存储结构转化为方法区的运行时数据结构；
3）在内存中生成一个该类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。
1.2 连接
1.2.1 验证
验证是连接阶段的第一步，主要确保加载进来的字节流符合 JVM 规范。
验证阶段会完成以下 4 个阶段的检验动作：
1）文件格式验证
2）元数据验证(是否符合 Java 语言规范)
3）字节码验证（确定程序语义合法，符合逻辑）
4）符号引用验证（确保下一步的解析能正常执行）
1.2.2 准备
准备是连接阶段的第二步，主要为静态变量在方法区分配内存，并设置默认初始值。
1.2.3 解析
解析是连接阶段的第三步，是虚拟机将常量池内的符号引用替换为直接引用的过程。
1.3 初始化
初始化阶段是类加载过程的最后一步，主要是根据程序中的赋值语句主动为类变量赋值。
注：
1）当有父类且父类为初始化的时候，先去初始化父类；
2）再进行子类初始化语句。

1.4 使用，
1.5 卸载，

说一下 Dubbo 中都用到哪些设计模式

工厂模式
Provider 在 export 服务时，会调用 ServiceConfig 的 export 方法.ServiceConfig 中有个字段：
private static final Protocol protocol = ExtensionLoader.getExtensionLoader（ProtocoLclass）.getAdaptiveExtension（）；
Dubbo 里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了 JDK SPI 的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在 classpath 下瑁加个文件就可以了，代

码零侵入。另外，像上面的 Adaptive 实现，可以做到 调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码
调试比较麻烦，需要分析出实际调用的实现类。

装饰器模式
Dubb。在启动和调用阶段都大量使用了装饰器模式。以 Provider 提供的调用链为
例，具体的调用链代码是在 ProtocolFilterWrapper 的 buildlnvokerChain 完成的，具体是将注解中含有 group=provicer 的 Filter 实现，按照 order 排序，最
后的调用顺序是：EchoFilter -> ClassLoaderFilter -> GenericFilter -> ContextFilter-> ExecuteLimitFilter -> TraceFilter -> TimeoutFilter -> MonitorFilter -> ExceptionFilter
更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter 的作用是 判断是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像
ClassLoaderFilter 则只是在主功能上添加了功能，更改当前线程的 ClassLoader,
这是典型的装饰器模式。

观察者模式
Dubb。的 Provider 启动时，需要与注册中心交互，先注册自己的服务，再订阅目 己的服务，订阅时，采用了观察者模式，开启一个 listener.注册中心会每 5 秒定 时检查是否有服务更新，如果有更新，向该服务的提供者发送一个 notify 消息，
provider 接受到 notify 消息后，即运行 NotifyListener 的 notify 方法，执行监
听器方法。

动态代理模式
Dubbo 扩展 JDK SPI 的类 ExtensionLoader 的 Adaptive 实现是典型的动态代理
实现。Dubb。需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪 个实现类，所以采用先生成代理类的方法，能够做到灵活的调用。生成代理类的
代码是 ExtensionLoader 的 createAdaptiveExtensionClassCode 方法。代理美
的主要逻辑是，获取 URL 参数中指定参数的值作为获取实现类的 key.

抽象工厂模式
ProxyFactory 及其子类是 Dubbo 中使用抽象工厂模式的典型例子。ProxyFactory 提供两个方法，分别用来 生产 Proxy 和 Invoker（这两个方法签名看起来有些矛盾，因为 getProxy 方法需要传入一个 Invoker 对象， 而 getInvoker 方法需要传入一个 Proxy 对象，看起来会形成循环依赖，但其实两个方式使用的场景不一样 ）。AbstractProxyFactory 实现了 ProxyFactory 接口，作为具体实现类的抽象父类。然后定义了 JdkProxyFactory 和 JavassistProxyFactory 两个具体类，分别用来生产基于 jdk 代理机制和基于 javassist 代理机制的 Proxy 和 Invoker。

适配器模式
为了让用户根据自己的需求选择日志组件，Dubbo 自定义了自己的 Logger 接口，并为常见的日志组件（包 括 jcl, jdk, log4j, slf4j）提供相应的适配器。并且利用简单工厂模式提供一个 LoggerFactory，客户 可以创建抽象的 Dubbo 自定义 Logger，而无需关心实际使用的日志组件类型。在 LoggerFactory 初始化时 ，客户通过设置系统变量的方式选择自己所用的日志组件，这样提供了很大的灵活性。
责任链模式

责任链模式在 Dubbo 中发挥的作用举足轻重，就像是 Dubbo 框架的骨架。Dubbo 的调用链组织是用责任链模 式串连起来的。责任链中的每个节点实现 Filter 接口，然后由 ProtocolFilterWrapper，将所有 Filter 串 连起来。Dubbo 的许多功能都是通过 Filter 扩展实现的，比如监控、日志、缓存、安全、telnet 以及 RPC 本身都是。如果把 Dubbo 比作一列火车，责任链就像是火车的各车厢，每个车厢的功能不同。如果需要加 入新的功能，增加车厢就可以了，非常容易扩展。

Mysql 题

在生产过程中会创建过多的索引，到底有没有生效，或者说有没有提升你得查询性能，你们是怎么测试的
1.使用 EXPLAIN 命令来执行进行分析。 2.新建一个测试表，添加 10 万+数据，开启 mysql 时间监测（set profile=1 ），show profile 看当前耗时时间。然后添加索引，再重新查看时间，两个进行比较
聚簇索引和非聚簇索引那个查询数据更快
前者更快。 因为主键索引树的叶子节点直接就是我们要查询的整行数据了。而非主键索引的叶子节点是主键的值，查到主键的值以后，还需要再通过主键的值再进行一次查询
SELECT COUNT(*) 在哪个引擎执行更快？
SELECT COUNT(*) 常用于统计表的总行数，在 MyISAM 存储引擎中执行更快，前提是不能加有任何 WHERE 条件。 这是因为 MyISAM 对于表的行数做了优化，内部用一个变量存储了表的行数，如果查询条件没有 WHERE 条件则是查询表中一共有多少条数据，MyISAM 可以迅速返回结果，如果加 WHERE 条件就不行。 InnoDB 的表也有一个存储了表行数的变量，但这个值是一个估计值，所以并没有太大实际意义。