# 整体要把控的原则

为什么要选用这个技术？这个技术怎么用？用的过程中常见的坑？这个技术的实现原理是什么？

# BZK项目

项目背景

·原有项目是一个巨石型应用，部署不灵活不方便，将其改为微服务架构。原有项目支持Tomcat集群化扩展。

项目亮点

1. 根据与客户端的url协议规定进行模板化反射service调用，减少了重复代码的编写，更易于维护。

可以直接用zuul的url使用zuul进行路由了（比如服务目录与元数据访问微服务、业务管控微服务等），但是这部分为什么不用zuul(WMS、WMTS、WFS等)：简而言之是历史原因，这个模板化调用是在巨石型应用的时候就弄好的，重构的要求是不影响原来的客户端代码调用，如果想不影响原来与客户端的url协议规定且还使用zuul，势必在每个微服务(WMTS,WMS)中都引入重复的对于客户端url进行处理的代码，所以这部分url协议的服务端处理代码没有使用zuul进行路由，保留了原有的模板化调用方式。

1. 使用微服务架构使得项目更灵活稳定易于部署。对于SpringCloud的常用组件在项目中进行了应用，比如使用Feign的伪RPC方式进行服务间调用更优雅，使用断路器使得项目更加稳定，避免雪崩问题。了解SpringCloud常用组件(eureka、ribbon、hystrix、feign、zuul)的基本工作原理。
2. 使用rancher+docker的方式进行部署

## 项目简介：

国防预研项目，将栅格与矢量等地图数据按照OGC的标准进行服务发布，项目分为WMTS、WMS、WFS、业务管控系统、服务配置管理系统等共十个模块，为多种终端提供地图服务。

## 工作内容：

单体架构下(支持集群化部署)：主要做了两个模块的内容，一是网关模块，二是服务配置管理模块：

1. 设计模板方法+BeanFactory+反射的应用模式，制定接口对接各个模块的业务代码。
2. 缓存获取的Method对象减少反射过程查找和创建Method对象的开销
3. 使用了Spring MVC中的Interceptor进行初步的用户权限的验证(目前没有用到Spring Security、Shiro等框架是因为权限验证模块还没来的及开发)，使用Spring AOP为各模块配置相应切面进行业务管控信息记录。
4. 使用了Redis实现的分布式Session，存储用户的token(SnowFlake算法生成)与用户的相关信息之间的对应关系。(目前redis还没有进行分布式配置，没来的及仔细学习，之后会弄)。
5. 负责与合作单位（中科院\*\*所等）的接口对接和演示联调。

微服务架构下(画好架构图带上)：

1. 负责将项目从单体架构拆分为微服务架构。
2. 学习SpringCloud常用组件(eureka、ribbon、feign、hystrix、zuul)的使用方式及其基本工作原理
3. 使用在单体架构中封装的模板方法+ribbon作为服务网关(没有使用zuul)；使用Feign的伪RPC方式进行服务间调用；引入hystrix防止微服务雪崩问题(目前统一使用的是线程池隔离方式，之后在性能测试时会进行优化)；配置ribbon自定义负载均衡策略(具体策略从负载均衡微服务中获取，因为不同微服务对资源的要求不同，比如wfs、wmts是io密集型微服务，wps是cpu密集型微服务，负载均衡会收集各个微服务应用的状态决定把请求转发到哪个微服务中)，。
4. 配置rancher+docker的方式部署微服务，每个微服务都可以开启多个实例，提供有web管理界面。rancher开源可免费使用。

项目使用的底层存储有mysql、hadoop、hbase。Mysql主要用来存储了用户的相关信息，token的相关信息，服务配置模块的信息等，hadoop与hbase中存储了地理信息服务所需的核心数据。

## 可能遇到的问题解答：

### 一、项目主导人的任务？十个模块还有什么？

1. 项目架构的搭建
2. 核心代码的编写
3. 项目进度的规划
4. 部分业务逻辑的编写

除了简历中涉及到的模块，还包含WCS、打包下载、目录与元数据、地名、网关模块。其中八个模块对外提供符合OGC标准的服务，**因为我在实验室里对于Spring生态框架的使用算是比较熟悉的，所以项目中我主要负责框架相关的任务的开发，具体包含网关模块与服务配置管理系统模块，没有做OGC标准服务的具体逻辑实现。**

### 二、为什么要将项目转为微服务架构？

微服务优势：复杂度可控、独立部署、技术选型灵活、容错、扩展

为什么要向微服务架构演变：

1. 模块较多，开发部署起来都越来越不方便；每个模块需要的服务器资源不同，比如WMTS，WFS等为IO密集型，WPS为CPU密集型，拆分后可以更好的利用服务器资源。
2. 项目用户端涉及到的均为查询操作，无分布式事务问题
3. 调研结果是微服务部署可使用开源的组件rancher+docker部署
4. 项目涉及到各方之间的沟通交流，项目模块较多，改变之后项目开发更为方便

以上原因保证微服务思路可以落地。

总结：由于微服务开发部署上的灵活性，支持多种语言的特性，性能上相比集群可以更细粒度的按需扩展

#### 关联问题一、为什么选用Spring Cloud不用Dubbo？



1、Spring Cloud考虑的是微服务治理的方方面面，是一个生态。 Dubbo是一个非常优秀的服务治理和服务调用框架，在这两点上甚至做的比Dubbo还好，但缺少很多功能模块，例如网关、服务容错、链路追踪等。 如果我们项目中之后要用到这些功能的话，或是自行实现，或是需要我们自己去调研相应的技术，选型之后再适配到我们的项目中来，代价较大。

2、之后可能会涉及到语言上的异构性质，可能会使用C++语言提供的微服务。

### 三、将项目转为微服务架构时遵循的原则是什么？数据库的拆分原则是什么？

项目拆分原则：每个服务单一职责，低耦合，高内聚，争取不修改一个功能就涉及到多个服

务。

数据库拆分原则：由于单体架构下只有服务配置管理模块、业务管控模块使用到了MySql，其它模块使用的是Hadoop作为底层存储，业务逻辑比较清晰简单，拆分起来还是比较容易的。

### 四、简单描述网关模块与服务配置管理模块的功能？

**网关模块：**项目的对外统一入口，对外发布符合OGC标准接口的地图服务，对内制定接口与各个服务的具体实现对接。并在此基础之上提供用户权限验证，业务管控信息记录等功能。

**服务配置管理模块：**对应用对外发布的服务进行统一的管理，只有管理员可访问服务配置管理模块。管理员在服务配置管理模块对发布的服务进行统一的管理。举例来说Hadoop集群中存储有多个瓦片集合，对外发布WMTS服务时可以选择这个服务可用的瓦片集合，未被选择的瓦片集合用户在访问这个WMTS服务时是不可见的。

### 五、为什么要使用反射的应用模式与各个模块对接？

**访问url示例：http://192.168.3.102:8084/services/ServiceSpace1/wms/testOGCService/测试发布地图\_2?VERSION=1.3.0&REQUEST=GetMap&LAYERS=测试发布地图\_2-矢量&STYLES=&CRS=EPSG:3857&BBOX=12801751,4074048,12829541,4096109&WIDTH=300&HEIGHT=300&FORMAT=png**

1. 因为我们对外提供的服务要符合OGC标准，而OGC标准中规定了对于每种服务类型的请求参数和返回值的统一格式，我观察后发现对于每种服务的请求参数中都含有两项：ServiceType以及REUQEST；其中ServiceType代表要访问某种类型的服务（可枚举，共八种），REUQEST代表的是要使用服务中的哪种功能(可枚举，比如GetCapabilities、GetTile、GetMap)；之后我就为这八种服务实现了一个模板化的Controller，根据请求中的ServiceType参数从BeanFactory(实际使用的是ApplicationContext)注入相应类型的Service对象实现，然后根据REQUEST参数反射调用Service对象中的方法获取返回值返回给客户端。这样就做到了八种服务共用Controller层代码，只需为八种服务每个服务设计相应的Service层接口即可。**因为我阅读分析过SpringMVC中DispatcherServlet分发请求到Controller层的具体实现，它的本质上也是模板方法(doDispatch)+BeanFactory(根据用户请求的URL对应获取到对应的Controller Bean)+反射方法调用(反射调用与用户请求url匹配的Controller中的方法，使用的是Method.invoke实现)实现的请求分发。我的这种实现方式借鉴了这个思想，不过我的业务场景和具体实现方式比起DispatcherServlet来说就要简单的太多了。**

也就是模板方法(模板化Controller层调用)+BeanFactory(根据ServiceType对Service层对象依赖注入)+反射(根据方法名称反射调用Service的相关方法)的应用模式。

#### 关联问题一、仔细讲一下DispatcherServlet进行请求分发(或者说SpringMVC初始化及请求转发实现原理)的实现原理？

1. Spring MVC的初始化过程是从DispatcherServlet的init方法执行开始的，首先读取配置文件(或者按照注解的配置)进行初始化，以通过ContextLoaderListener建立起来的WebApplicatioContext作为父上下文，这个过程中比较重要的一步就是初始化我们所有的Controller层的Bean放在IOC容器中以备使用。
2. DispatcherServlet的init方法第二个重要的步骤就是初始化HandlerMapping类型的Bean，以RequestMappingHandler为例，在通过getBean方法实例化一个RequestMappingHandler类型的对象的时候，**先扫描我们所有自定义的Intercepetor类**型的对象，注册到RequestMappingHandler中以备使用。**之后会扫描所有使用@Controller注解标注的Controller类**，获取类中所有使用RequestMapping注解标注的方法，之后在RequestMappingHandler中建立起一个Map，Map的key值为RequestMapping注解中具体的Url，Map的value值为Controller所在的具体路径(包名+类名)+对应的方法名。
3. 请求分发到具体的Controller是在DispatcherServlet的service方法中实现的，简要来说就是通过用户请求的url在RequestMappingHandler中进行匹配，之后反射调用对应Controller层中的相应方法来实现的。在调用具体业务方法之前，还会检测是否有interceptor需要调用。

#### 关联问题二、BeanFactory与ApplicationContext的异同？

ApplicationContext继承自BeanFactory，在BeanFactory的基础上又添加了一些新的功能。ApplicationContext·中对于Bean的依赖注入(即对于getBean()方法的调用)是委托给BeanFactory来实现的

ApplicationContext在BeanFactory基础之上增加的新功能有：

1. Spring事件处理机制。Spring事件处理机制符合标准的Observer设计模式，由ApplicationEventPublisher、ApplicationEvent、ApplicationListener三个组件组成。
2. 国际化功能
3. Spring IOC容器初始化的时候在AbstractApplicationContext.refresh()方法中对registerBeanPostProcessors的调用会默认注册一些BeanPostProcessors，**比如对于事务处理和AOP的支持都是通过BeanPostProcessor实现的，如果仅仅使用BeanFactory，那这些特性就不能使用了。除非在代码中**代码中使用BeanFactory通过手工调用addBeanPostProcessor()方法进行注册。
4. ApplicationContext之间可以具有父子关系，而BeanFactory之间就不能构建这种关系。比如Web应用通过ContextLoaderListener生成的WebApplicationContext为父上下文，每个DispatcherServlet都可以具有自己的上下文，该上下文独立于任何其它DispatcherServlet的上下文。

**另补充：BeanFactory中Bean的生命周期：**

1. 实例化Bean对象（调用构造方法） 2、设置对象属性（调用set方法）
2. 检查Aware接口并设置相关依赖 4、BeanPostProcessors前置处理
3. 检查是否是InitializingBean以决定是否调用afterProperties方法
4. 检查是否有自定义的init-method
5. BeanPostProcessors后置处理 8、注册必要的Destruction相关回调接口
6. Bean使用中 9、Bean销毁 10、是否实现DisposableBean接口

11、是否配置有自定义Destroy方法

#### 关联问题三、反射的底层实现原理及项目中对反射(method.invoke)的性能优化？

每个Java方法有且只有一个Method对象作为root，它相当于根对象，对用户不可见。当我们创建Method对象时，我们代码中获得的Method对象都相当于它的副本（或引用）。也就是说每次调用Class.getDeclareMethod（）方法都会新创建一个Method对象。

root对象持有一个MethodAccessor对象，所以所有获取到的Method对象都共享这一个MethodAccessor对象。Method.invoke方法底层实际是使用MethodAccessor.invoke方法来实现的。

MethodAccessor的具体实现类有：

* sun.reflect.DelegatingMethodAccessorImpl
* sun.reflect.MethodAccessorImpl
* sun.reflect.NativeMethodAccessorImpl

由于我们获取到的同一个Java方法的所有Method对象底层对应的都是同一个MethodAccessor对象，所以MethodAccessor.invoke方法一定是线程安全的。由此可以推断得知Method.invoke方法也是线程安全的。

既然Method.invoke方法是线程安全的，在我的业务场景下，我就在Method对象创建后缓存在一个ConcurrentHashMap中(key值为serviceType+REQUEST,value为对应的Method对象)，这就省去了Method.getDeclaredMethod()方法中频繁查找符合我们要求的ROOT Method及创建Root Method对象副本的开销。

参考文章：

https://blog.csdn.net/vikeyyyy/article/details/79083097

<https://mp.weixin.qq.com/s/5H6UHcP6kvR2X5hTj_SBjA>?

https://wj-126mail.iteye.com/blog/1012942

### 六、讲一下Spring MVC Interceptor的实现原理？

参考问题五关联问题一中的第二点。DispatcherServlet中init方法执行时的第二个步骤。

### 七、具体讲一下项目中SpringAOP的应用场景以及Spring AOP机制的实现原理？

**具体应用场景：**使用@Aspect注解配合@Before、@After注解实现，为**不同模块 配置 不同的切面**完成业务管控日志的记录（**AOP的核心思想即为基础(源程序即我们这里的不同的模块)+切面(通知程序)+配置(编织)**），存入数据库。

**AOP机制实现原理：AOP核心思想为源程序+切面+编织，其中最重要的一点就是编织，即如何把我们的切面程序编织进源程序当中实现拦截通知的目的。Spring AOP是通过动态代理来实现编织过程的，具体过程可总结为以下几步：**

1. **通过BeanPostProcessor拦截创建好的Bean，判断是否需要对其进行拦截通知。**
2. **如果需要的话则为其创建动态代理对象，以JDK动态代理为例，创建的动态代理对象中持有的InvocationHandler对象可以是一个**JdkDynamicAopProxy。
3. 通过JdkDynamicAopProxy对象可以获取我们定义的拦截器链(通过其成员变量ProxyFactory获取，动态代理对象也是通过ProxyFactory对象来创建的)，之后根据是否满足通知程序的切点来决定是否要对此次方法调用进行拦截通知。

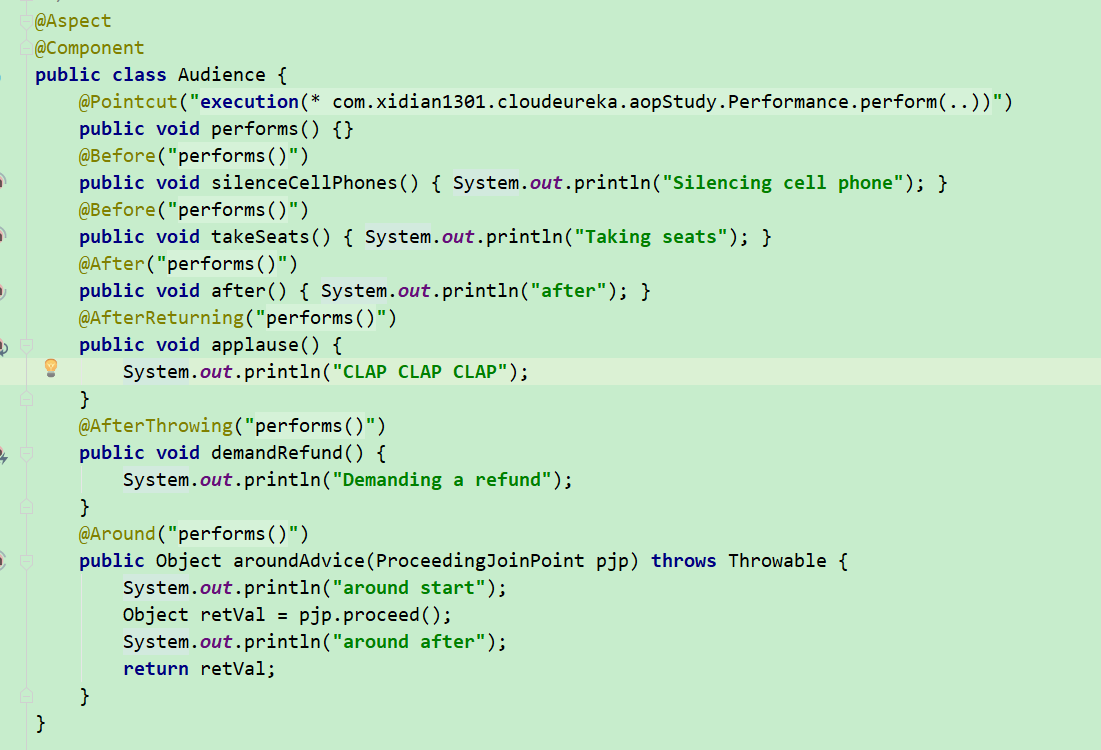
#### 关联问题一：Spring AOP中前置通知、后置通知、返回通知、异常通知、环绕通知几种类型的通知的异同？

几种通知的比较：

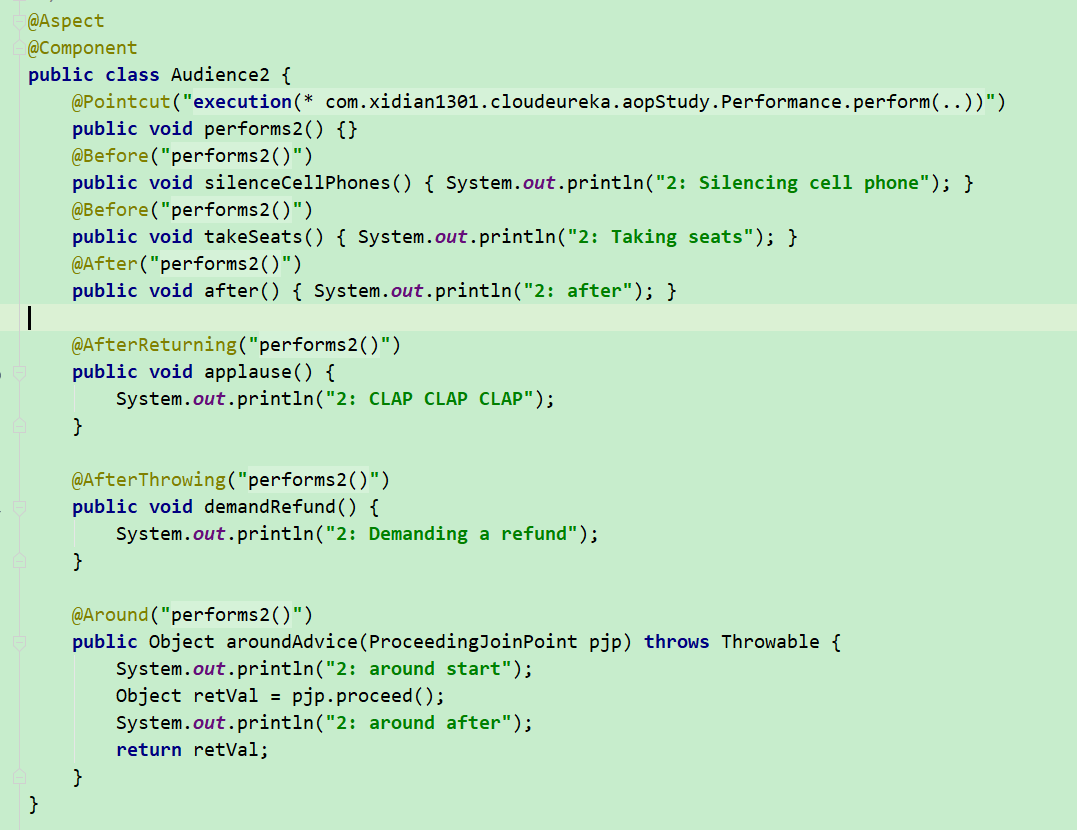


具体执行实例如下：

切面一代码：



切面二代码：



结果：



**@AfterThrowing注解**表示这个方法是用来作为异常通知，也就是在它签名中所标识的具体方法调用并出现异常之后才会进入这个方法

**环绕通知：**Around advice通常是用在下面这样的情况：

在多线程环境下，在joinpoint方法调用前后的处理中需要共享一些数据。如果使用Before advice和After advice也可以达到目的，但是就需要在aspect里面创建一个存储共享信息的field，而且这种做法并不是线程安全的。

**实例：**



#### 关联问题二：AOP中对一个切点有多个响应的通知函数，这些函数的执行顺序如何指定？

1、对于在同一个切面定义响应同一个切点的通知函数将会根据在类中的声明顺序执行。

2、如果在不同的切面中定义多个通知响应同一个切点，进入时则优先级高的切面类中的通知函数优先执行，退出时则最后执行，优先级由切面类实现的Ordered接口中getOrder方法返回值确定，返回值越小，优先级越高。

#### 关联问题三：Spring AOP实现动态代理有两种方式，JDK动态代理和CGlib动态代理，使用哪种方式比较好呢？

CGLIB创建代理对象时所花费的时间却比JDK多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

### 八、讲一下Spring事务控制机制的实现原理？

Spring事务控制系统的实现核心是AOP思想，通过TransactionProxyFactoryBean中获取我们使用的Service层的对象，这里获取的对象实际是使用ProxyFactory对象生成的代理对象，我们调用代理对象的方法时就会先使用TransactionInterceptor拦截器进行拦截（其中含有PlatformTransactionManager对象），通过这个拦截器中的环绕式拦截实现事务的开启与提交。

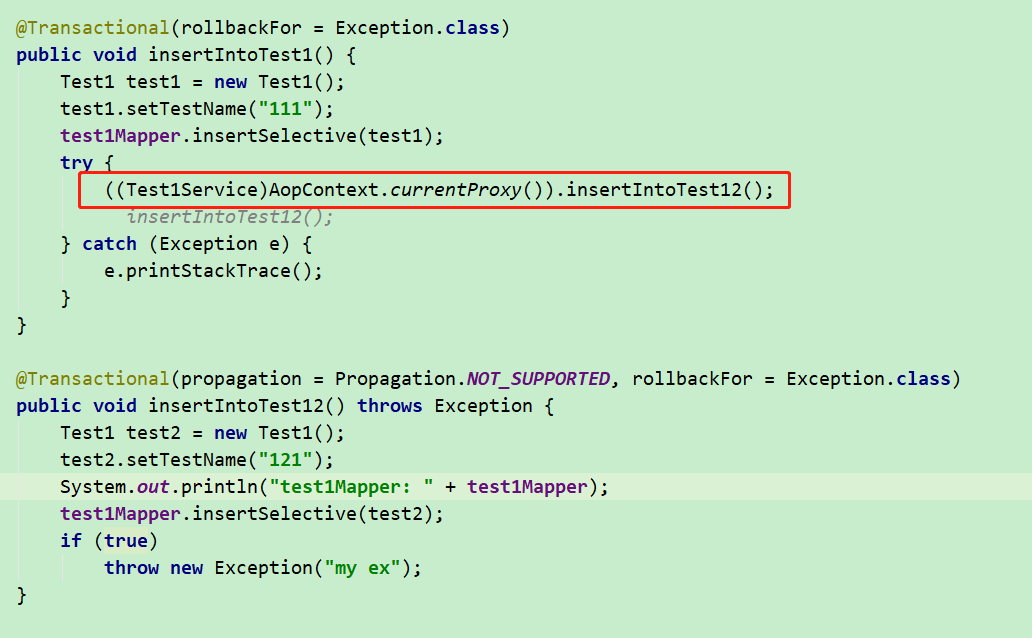
在使用Spring进行事务控制时我们获取到的数据库连接对象是通过TransactionSynchronizationManager中的ThreadLocal<Map<Object, Object>> resources

来拿到的，ThreadLocal<Map<Object, Object>> resources中map的key值为我们使用的数据源对象，Value为**与当前线程绑定**的数据库连接。我们拿到数据库连接对象时，这个对象上已经为我们开启了事务控制了，在我们业务方法执行完成后，这个对象上的commit或者rollback方法也会被自动调用。就是通过这样的过程完成了Spring框架的事务控制。

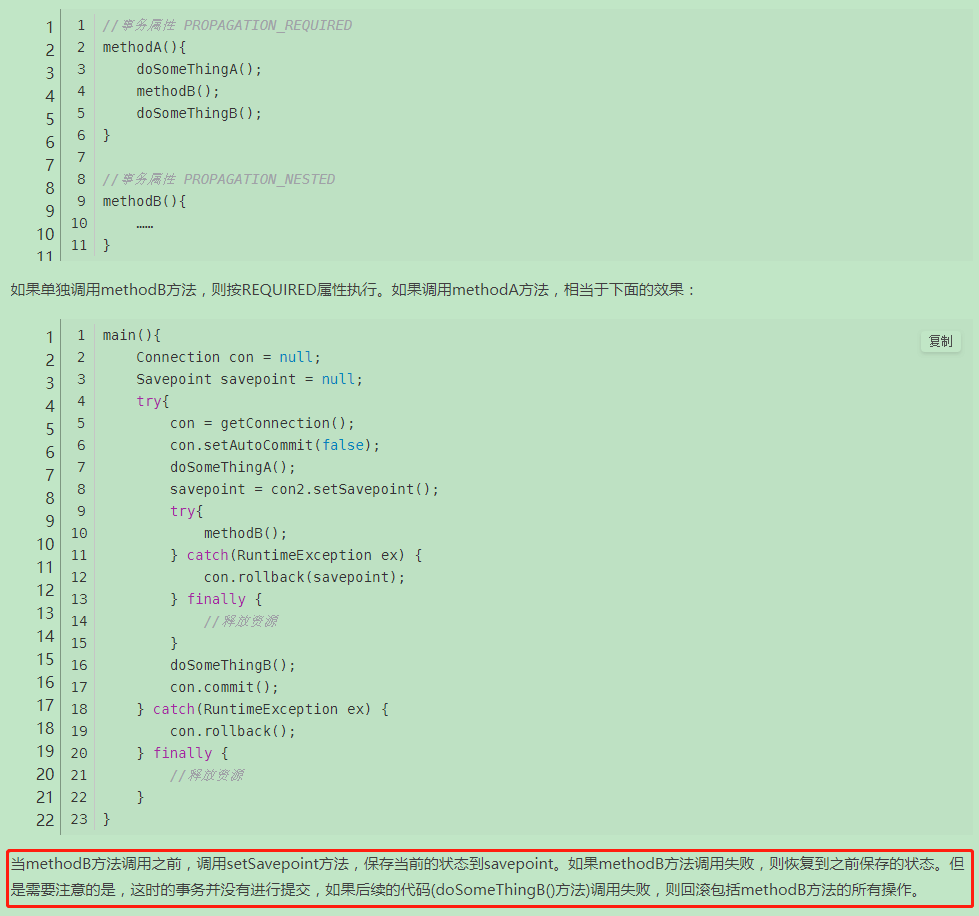
#### 关联问题一：Spring事务控制几种传播机制的异同及其实现原理？



**PROPAGATION\_REQUIRED:**默认的传播机制。当在一个Service方法A部调用Service的另一个方法B时，由于使用的是真正对象调用的B，没有使用Spring提供给我们的代理对象，B方法中执行的每个SQL语句都属于一个独立的时候，执行完之后即提交，即使方法B中抛出了异常，方法B中已经进行的数据库操作也不会回滚。想要解决这个问题，**需要先在SpringBootApplication上添加@EnableAspectJAutoProxy(exposeProxy = true)注解，之后使用AopContext.currentProxy()获取代理对象后调用B方法**才可以发挥事务传播机制的作用：



**PROPAGATION\_NESTED**:嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。



### 九、讲一下MyBatis与Spring的结合原理？

在单独使用MyBatis时是通过SqlSession.getMapper()方法来获取mapper接口的代理对象的，SqlSession对象不是线程安全的。

在MyBatis与Spring结合使用时，Mapper接口的代理对象是通过SqlSessionTemplate.getMapper()方法来生成的，最终获得的代理对象中持有的invocationHandler是一个MapperProxy类对象，其中引用的SqlSession对象是一个SqlSessionTemplate类的实例。

SqlSession对象在多线程使用时不是线程安全的，**在与Spring结合的过程中使用MyBatis时我们使用的单例mapper对象是可能在多线程情况下被操作的，那么线程安全问题是怎么解决的呢？**

SqlSessionTemplate在执行的过程中会通过其内部的代理对象为每个线程生成自己的SqlSession对象（每个SqlSession对象中持有一个数据库连接）并保存在TransactionSynchronizationManager中的ThreadLocal变量中，就是通过**动态代理的设计模式加上ThreadLocal方式完成了线程与SqlSession对象之间的对应**，保证了SqlSession使用时的线程安全性。

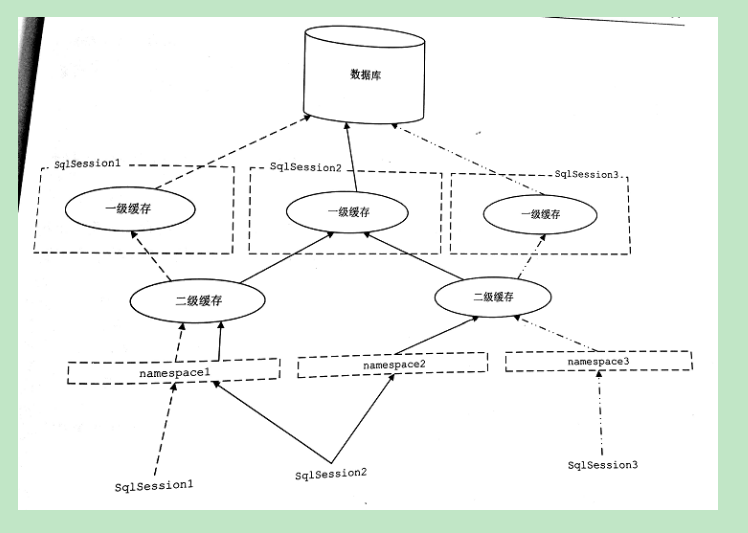
MyBatis使用时其事务控制是交由Spring框架来实现的，这是怎么做到的呢？

Spring与MyBatis结合之后是会将事务交由Spring进行处理的，这是通过SpringManagedTransaction来实现的，sqlSession在执行时使用的Connection对象是通过SpringManagedTransaction.getConnection()方法来实现的，SpringManagedTransaction.getConnection方法获取数据库连接的过程是先通过TransactionSynchronizationManager.getResource方法尝试获取，拿不到再创建。

Spring在Service层开启事务之后会把当前线程对应的开启了事务的Connection对象封装成ConnectionHolder对象存入TransactionSynchronizationManager中，所以我们通过TransactionSynchronizationManager.getResource方法获取到的就是这个已经开启了事务的Connection，就是通过这种方式实现了将事务交给Spring控制的，所以TransactionSynchronizationManager在Spring事务控制的过程中的地位也是很重要的，是MyBatis与Spring之间沟通的桥梁。

#### 关联问题：讲一下MyBatis的一级缓存与二级缓存的异同？如何使用呢？

MyBatis查找数据的过程是先在二级缓存（如果开启了的话），之后是一级缓存，最后是实际数据库，mapper.xml中对应的namespace唯一对应一个二级缓存对象，多个mapper.xml可以对应同一个二级缓存对象，具体如下图：



一级缓存（其实底层实现就是一个HashMap）的使用比较简单，由于每个SqlSession对象对应于一个线程，比如这个线程先进行了一次selectById的查找，之后把结果缓存起来，然后在进行selectById时发现一级缓存中有，则直接返回了，不再到数据库中进行查找。

#### 关联问题：讲一下MyBatis二级缓存的使用原则？

二级缓存的使用时注意事项

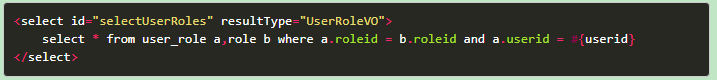
1. 只能在【只有单表操作】的表上使用缓存：不只是要保证这个表在整个系统中只有单表操作，而且和该表有关的全部操作必须全部在一个namespace下。
2. 在可以保证查询远远大于insert,update,delete操作的情况下使用缓存，这一点需要保证在1的前提下才可以！

**为什么避免使用二级缓存?**  
在符合【Cache使用时的注意事项】的要求时，并没有什么危害。其他情况就会有很多危害了。比如：

**1、针对一个表的某些操作不在他独立的namespace下进行。**

例如在UserMapper.xml中有大多数针对user表的操作。但是在一个XXXMapper.xml中，还有针对user单表的操作（UserMapper.xml与XXXMapper.xml使用的是不同的nameSpace）。 这会导致user在两个命名空间下的数据不一致。如果在UserMapper.xml中做了刷新缓存的操作，在XXXMapper.xml中缓存仍然有效，如果有针对user的单表查询，使用缓存的结果可能会不正确。

1. **涉及到多个表的查询操作可能会由于二级缓存的存在引发错误的查询结果。**



**像上面这种查询（查询一个用户的所有角色），**不管是写到RoleMapper.xml还是UserRoleMapper.xml，或者是一个独立的XxxMapper.xml中。如果使用了二级缓存，都会导致上面这个查询结果可能不正确。

比如将其写到RoleMapper中，第一次从数据库获取查询结果之后将结果存入了二级缓存，第二次获取直接返回的是从二级缓存中取出的结果。如果在第一次获取与第二次获取之间正好修改了这个用户的角色，第二次查询使用缓存的时候结果就是错的。

### 十、简述分布式Session的实现原理？

常见实现方案调研结果：

**1、Session复制：**由应用服务器提供支持。每台应用服务都保存会话Session，只要应用中出现了Session数据变化，就需要同步到所有机器上。机器越多，网络开销越大。

**2、Session集中管理：**再加一台服务，专门用来管理Session数据，每台应用服务器都从专门的Session管理服务中取会话Session数据，可以使用数据库保存Session。这种方式缺点在于相比于本机Session读写，带来了延时和不稳定性，还有就是Session集中服务如果出了问题，会影响整个系统。

项目需求：给用户端生成token(可以理解为sessionId)，使用 token作为Redis的Hset的Key值，value值(hset结构的value值是一个hashMap)相当于我们常用的session。设置这个token的默认超时时间为2小时，用户使用token访问时，redis中token对应的session的超时时间就进行更新，只有用户在2小时以内没有任何操作时，这个session才会失效。

**Token的生成算法采用Twitter的SnowFlake算法，保证分布式情况下分成的id的唯一性。**

参考文章：(精品)https://segmentfault.com/a/1190000011091273#articleHeader17

#### 关联问题一、为什么没有使用SpringSession，自己造轮子?

Spring Session虽然优点颇多，但是项目中目前需要功能的很简单，而对我来说做项目的时候引入一个新的框架尽量想把这个框架弄熟一些在运用进来，这就需要一些时间来学习，否则不熟的话问题就不太好解决，当时做的时候时间要求比较紧，所以目前自己基于Redis造了一个简单的轮子，模拟实现了分布式Session的功能，之后如果项目需求有变化的话，会考虑引入Spring Session。

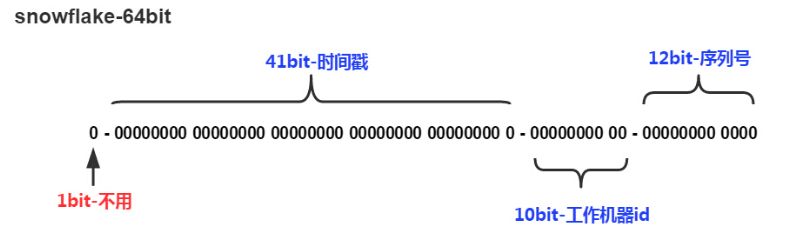
#### 关联问题二、详细介绍下SnowFlake算法？

**为什么不使用UUID？**

因为UUID虽然可以保证全局唯一，但是占用32位有些太长，而且是无序的，入库时性能较差。

**SnowFlake算法介绍：**

目标是在分布式系统中生成全局唯一且趋势递增的ID，并且只有64位(8字节)，占用空间较小。



**Twitter在1ms支持生成的唯一Id的数量约为420万。**

SnowFlake算法支持的时间约为140年(根据前42位时间戳能存储的总ms数计算得出)。比如应用开始使用的时间为2010年，可以支持到2150年。

#### 关联问题三、简要介绍下一致性Hash算法？

Memcached要想实现分布式只能在客户端来完成,目前比较流行的是通过一致性hash算法来实现.常规的方法是将server的hash值与server的总台数进行求余,即hash%N,这种方法的弊端是当增减服务器时,将会有较多的缓存需要被重新分配且会造成缓存分配不均匀的情况(有可能某一台服务器分配的很多,其它的却很少).

 　　今天分享一种叫做”ketama”的一致性hash算法,它通过虚拟节点的概念和不同的缓存分配规则有效的抑制了**缓存分布不均匀，并最大限度地减少服务器增减时缓存的重新分布**

**总结：一致性Hash算法致力于解决两个问题：**

**1、缓存分布不均匀 2、最大限度地减少服务器增减时缓存的重新分布**

**目的是为了解决增减服务器时会引起大量缓存失效，需要重新到数据库中查询存入缓存的弊端。**在Memcached上使用最多，在对数据库进行分表时也可参考使用这种算法，以尽量减少数据库分表需要扩张时带来的数据迁移问题。

还有个需要解决的问题是缓存分配不均匀的问题。

解决方案是将一个物理Memcached Server映射到圆环上的多个虚拟Memcached Server节点，比如每个物理Memcached Server节点映射到250个虚拟的Memcached Server节点，就像地铁站出口一样,出口越多,每个出口出现拥挤的情况就会越少，虚拟的Memcached Server节点越多，缓存数据就可以更均匀的分布到多个物理Memcached

Server中。

参考文章：

(精品)<https://www.cnblogs.com/RockLi/p/3530176.html>

<https://blog.csdn.net/xaccpJ2EE/article/details/82993120>

### 十一、为什么在项目中使用Eureka作为注册中心？

简单来说，由于我们项目中对于可用性的要求超过对于一致性的要求，所以选用Eureka。

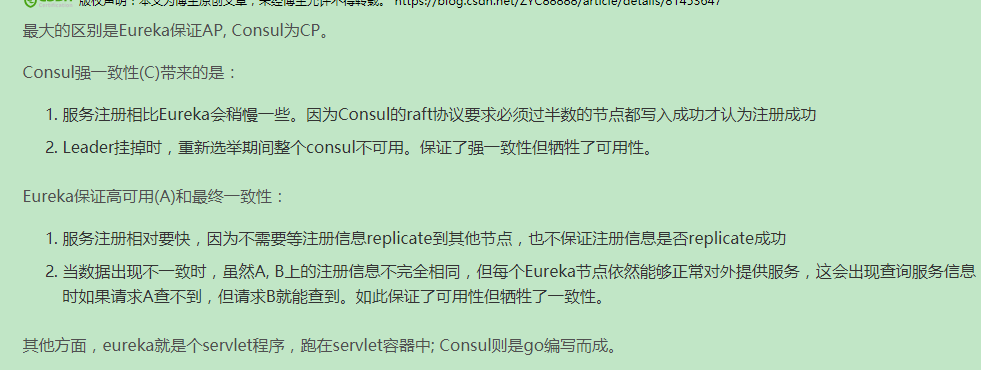
只对比了Consul、Zookeeper、Eureka (因为目前spring cloud对Alibaba nacos 、etcd支持貌似还不太好)

1. Eureka不追求强一致性，服务注册过程完成快
2. Consul与Zookeeper在Leader宕掉重新选举期间貌似不可用

Eureka，可用性（最重要的一点，CAP中支持AP，追求数据的最终一致性）；易用性（SpringCloud默认选用的注册中心，netflix贡献给开源社区，经过了生产环境的检测）；开源（java开发，便于我们更深入的理解其实现原理，更好的使用；consul为go开发）

Eureka中没有使用任何的数据强一致性算法保证不同集群间的Server的数据一致，仅通过数据拷贝的方式争取注册中心数据的最终一致性，虽然放弃数据强一致性但是换来了Server的可用性，降低了注册的代价，提高了集群运行的健壮性。在Eureka平台中，如果某台Eureka服务器宕机，客户端请求会自动切换到新的Eureka节点。没有类似于Zookeeper、Consul这种选举Leader的过程。

为什么使用Eureka不用Consul？



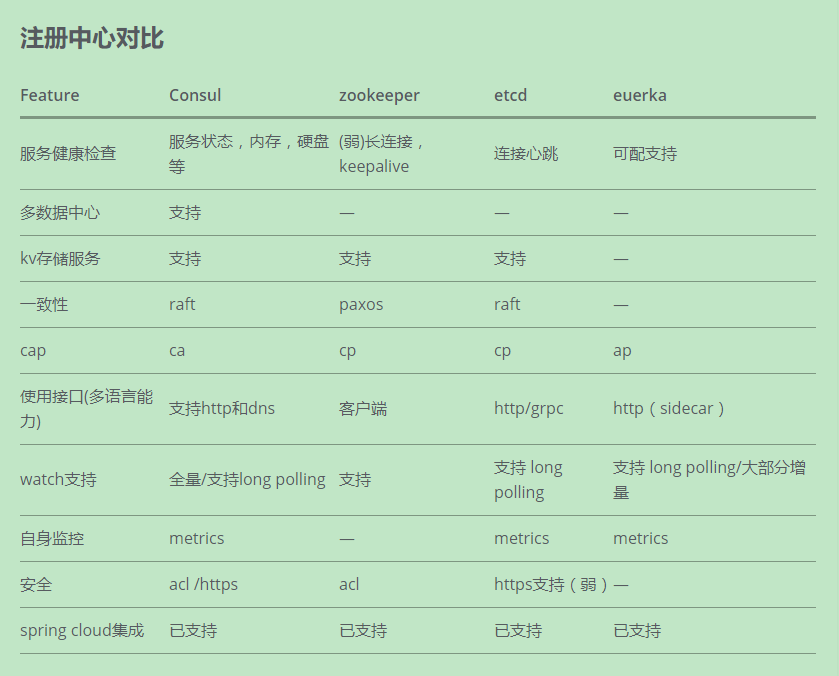
为什么使用Eureka不用Zookeeper？

Zookeeper本质是分布式协调服务，它的职责是保证数据（注：配置数据，状态数据）在其管辖下的所有服务之间保持同步、一致，并不是专为作为服务注册中心而设计的。

Zookeeper中支持的也是CP，其提供的是顺序一致性服务，**来自任意特定客户端的更新都会按其发送顺序被提交保持一致**。也就是说，如果一个客户端将Znode z的值更新为a，在之后的操作中，它又将z的值更新为b，则没有客户端能够在看到z的值是b之后再看到值a（如果没有其他对z的更新）。并且有更新操作实时性的保证，在特定的一段时间内，客户端看到的系统需要被保证是实时的（在十几秒的时间里）。在此时间段内，任何系统的改变将被客户端看到，或者被客户端侦测到。



所有注册中心对比：



参考文章：[分布式系统的Raft算法](https://www.jdon.com/artichect/raft.html)， [ZooKeeper和CAP理论及一致性原则](http://www.ymq.io/2018/05/18/zookeeper/)， [分布式中几种服务注册与发现组件的原理与比较](https://juejin.im/post/5bb77923f265da0af3348aa3%23heading-0)

#### 关联问题一、简述Eureka的实现原理？

Eureka中涉及到三个角色，服务注册中心、服务提供者、服务消费者。其核心功能包含服务注册、服务同步、服务续约、服务下线(有Eureka Client正常下线主动剔除的方式和EurekaServer通过定时任务的方式剔除一定时间内没有服务续约消息的两种方式)、获取服务、失效剔除、自我保护等。

**服务同步：如果有多个Eureka Server，一个服务提供者向其中一个Eureka Server注册了，这个Eureka Server会向集群内的其它Eureka Server转发这个服务提供者的注册信息，从而实现实现Eureka Server之间的服务同步。**

以Eureka Client为例，使用Eureka Client的相关功能之前要在SpringBoot的启动类上加上@EnableDiscoveryClient注解，这个注解会引入一些自动化配置类，其完成的核心功能就是创建了一个DiscoveryClient的Bean，正是通过这个Bean完成了Eureka Client的一系列功能(服务注册、服务续约、可用服务列表获取等功能)。

#### 关联问题二、什么是Eureka的自我保护机制？其作用是什么？

当EurekaServer短时间内丢失了过多的实例连接时(比如由于网络故障或频繁的启动关闭客户端)，那么这个EurekaServer节点就会进入自我保护模式，一旦进入到该模式，EurekaServer就会保护服务注册表中的信息，不会删除任何服务注册表中的数据。

Eureka进入自我保护模式通常是因为多个服务实例与此Eureka之间的网络连接都出现了问题，此时问题可能出在网络连接，而不是在于注册到Eureka上的多个微服务，这是一种应对网络异常的安全保护措施。**\*\*它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。\*\***使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

### 十二、为什么Feign的调用方式为伪RPC调用？简述Feign的实现原理？

Feign对Ribbon与Hystrix进行了进一步的封装，使得服务之间的依赖调用非常优雅(伪RPC)。实际的请求调用是使用Ribbon实现的，底层使用的还是HTTP协议，所以称之为伪RPC调用。

Feign的实现原理也是动态代理，通过扫描@FeignClient注解标注的接口，通过spring的IOC方式注入到我们的应用中。之后生成接口对应的代理RequestTemplate模板对象，通过代理类实现对依赖服务的伪RPC调用以及上述Feign的核心功能。

#### 关联问题一、RPC调用的实现原理？

RPC调用的实现原理本质上讲是动态代理，通过代理对象帮我们调用对应的远程对象的相关方法，代理对象中屏蔽了网络通信的细节。

Java中RPC调用框架可用的有原生的RMI(使用的很少)，Dubbo、Motan、Thrift(可跨语言调用)等等。

参考文章：<https://colobu.com/2016/09/05/benchmarks-of-popular-rpc-frameworks/>

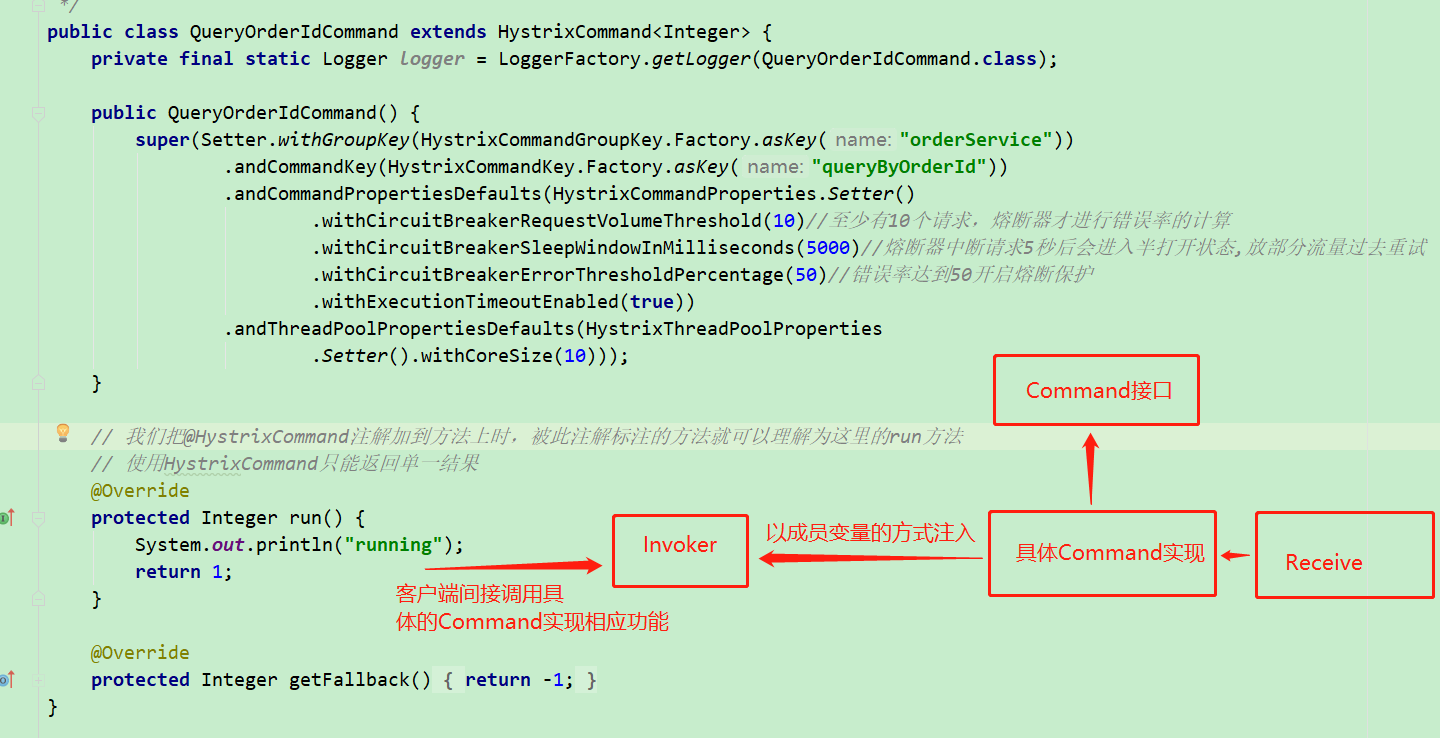
### 十三、什么是雪崩问题？Hystrix是如何解决雪崩问题的？

**雪崩问题：**在微服务架构中往往存在许多个服务单元，若一个服务单元出现故障，其它服务调用这个故障服务时就可能被阻塞，这样就很容易因为依赖关系引发故障的蔓延，最终导致整个系统的瘫痪，也就是常说的雪崩效应，为了解决这个问题，产生了断路器等一系列的服务保护机制。

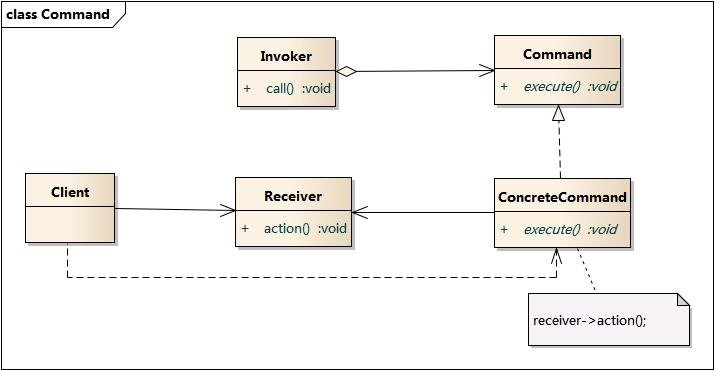
举例来说：服务B调用服务C，此时服务C上的tomcat并发线程数已经达到200,并且有90个请求在排队等待处理，这是服务B的请求就会阻塞在请求C上迟迟不能返回；这样就会导致服务B上的请求堆积的越来越多，结果服务B又不可用了，然后服务B又会导致调用它的其它服务不可用，引起雪崩。

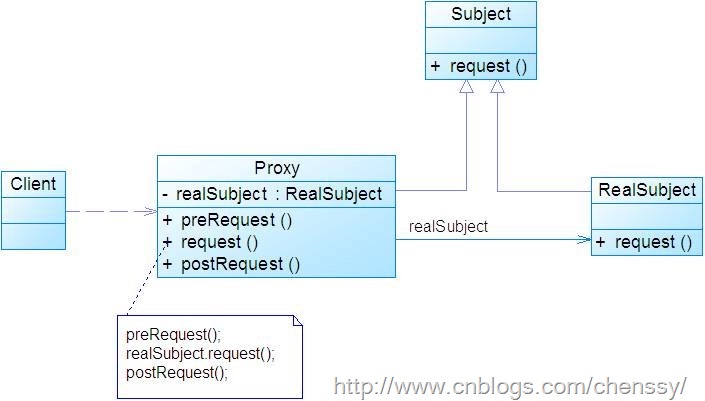
**Hystrix通过请求隔离和服务降级来解决雪崩问题。**

请求隔离的实现方式有信号量隔离（默认每个方法共享一个信号量，底层实现是AtomicInteger）和线程池隔离（默认每个类下的所有方法调用共享一个线程池，默认实现方式是ThreadPoolExecutor）两种实现方式，是通过命令模式来驱动执行的，把被@HystrixCommand注解标注的方法通过动态代理的方式封装为Command对应的Run方法来执行。线程池隔离方式调用线程可以随时离开，信号量隔离则不行。



命令模式的优势在于解耦行为请求者与行为实现者。





#### 关联问题一、Hystrix FallBack项目中怎么设计的？

为通过Feign进行调用的方法指定相应的fallback函数，返回相应的错误码实现。Feign中相应的Hystrix配置可以细化到每个url对应的接口实现。

参考：https://www.jianshu.com/p/82d9fece6bc0

### 十四、简述Spring Cloud核心组件的实现原理？

* Eureka：各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里
* Ribbon：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台
* Feign：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求
* Hystrix：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题
* Zuul：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

### 十五、Rancher的作用？

Rancher是一个开源的企业级容器管理平台。通过Rancher，企业再也不必自己使用一系列的开源软件去从头搭建容器服务平台。Rancher提供了在生产环境中使用的管理Docker和Kubernetes的全栈化容器部署与管理平台。

Rancher可以更方便的对Docker应用进行编排，在传统的容器编排工具之上进行了封装，提供了一个可视化管理界面，可以非常方便的配置Docker容器之间的网络划分之类的，仅需点击按钮即可启动新的服务实例，Rancher会自动将服务实例在某个物理机器上启动。其底层支持的容器编排工具有Docker Swarm、K8S、Cattle等。

# 比赛

天池第一届阿里云POLARDB数据性能大赛

## 项目简介：

实现一个简化、高效的 kv 存储引擎。初赛要求支持 open、Write、Read、close 接口。复赛在初赛题目基础上，还需要额外实现一个 Range 接口。程序评测共分为两个阶段：

1. 正确性评测

评测程序会并发写入特定数据（key 8B、value 4KB）同时进行任意次 kill -9 来模拟进程意外退出（参赛引擎需要保证进程意外退出时数据持久化不丢失），接着重新打开 DB，调用 Read、Range 接口来进行正确性校验，**后写入的key要求覆盖先写入的key。**

1. 性能评测

随机写入：64 个线程并发随机写入，每个线程使用 Write 各写 100 万次随机数据（key 8B、value 4KB）

随机读取：64 个线程并发随机读取，每个线程各使用 Read 读取 100 万次随机数据

顺序读取：64 个线程并发顺序读取，每个线程各使用 Range 有序（增序）**遍历全量数据 2 次**

语言限定：C++ & JAVA，一起排名 java内存占用限制：3G(评测程序占用500M，实际可用2.5G)；

题目概述：要求实现一个高效的KV存储引擎，实现对6400万对KV的高效查询。性能评测分为打开存储引擎，随机写入，随机读取，顺序读取，关闭存储引擎五个阶段。在Key值相同的条件下，后写入的KV要覆盖先写入的KV，写入阶段会进行任意次Kill -9模拟进程退出，引擎需要保证被kill时数据不丢失。

题目难点：在于如何在2.5G的内存限制条件下在每个阶段都打满磁盘IO。

实现过程：

## 解题过程：

初赛考点：

1、

## 实现亮点：

1. 设计了顺序写阶段的完全无锁化的存储架构，在顺序写阶段与随机读阶段基本打满磁盘IO。
2. Open阶段借鉴concurrentHashMap分段思想，将个LongLongHashMap中，减少锁冲突，使得Open阶段实现了并发索引构建，性能提升10s左右。
3. 使用MappedByteByffer进行索引的写入，实现WAL，并在使用完之后及时回收，防止内存泄漏；使用DIO进行数据的顺序写入与随机读取，绕过PageCache，减少多余数据拷贝。
4. 使用多线程快排+ForkJoin归并实现6400w key值3-4s左右排序。
5. 使用线程池复用填充缓存的线程，减少线程创建销毁开销。两次ThreadLocal进行对象复用，防止对象频繁创建回收。(比如read阶段保存每个线程读取数据使用的4kb的byte数组)
6. 自实现缓存使得一次IO操作可以满足64个读取线程的读取操作，并发缓存填充并且使用双buffer并行缓存填充与缓存读取操作(测试填充缓存的线程数为96个时为最佳，读取缓存的线程数固定为64个)提升性能(663到543，进步了120s)，其实还可以进一步做到RingBuffer(引用计数+wait-notify+buffer标志位即可实现)
7. 尽力使用好JUC包中的并发工具类提升并发操作性能，使用的类包含有CyclicBarrier、SynchronousQueue、ThreadPoolExecutor、AtomicInteger
8. Gc参数调优。两个方面：1、是在实现自定义缓存之后时间与预计的非常不符合，排查发现填充缓存时调用的一个工具函数(longToByte)中的一个对象没有复用，导致每次填充cache产生的朝生夕灭的对象就达到了96M，range 6400w数据需要填充977次cache，这样就会产生非常多的朝生夕灭的result对象，GC次数非常多，严重影响了程序性能。对其进行复用之后从2300s提升到了663s。2、调整GC参数，使用的是CMS+ParNewGC配合的方式，主要影响性能的参数是-XX:NewRatio，最终将其设置为-XX:NewRatio：3效果最好，此时内存当中主要有索引1G，两块cache256M，评测程序内存占用500M。归并排序的时候占用多余内存900M(后来发现其实可能只用一半内存就可以)，内存最大占用为2.5G左右，由于归并排序的高内存占用导致想把cache放在堆外的时候常常会发生内存溢出的错误。GC调优参考文章：http://xxfox.perfma.com/jvm/optimize

核心代码1k，算上外围代码（工具类、测试代码）大概4-5K行。

## 可能遇到的问题解答：

### 一、为什么初赛采用顺序写，随机读的方式？换句话说，为什么不使用B+树，LSM树这种结构呢？

***为什么不采用B+树的方案：***

B+树的特点：矮胖，非叶子结点不存储数据；在实际应用中的B树的阶数m都非常大（通常大于100），所以即使存储大量的数据，B树的高度仍然比较小。

B+树的两个优势：

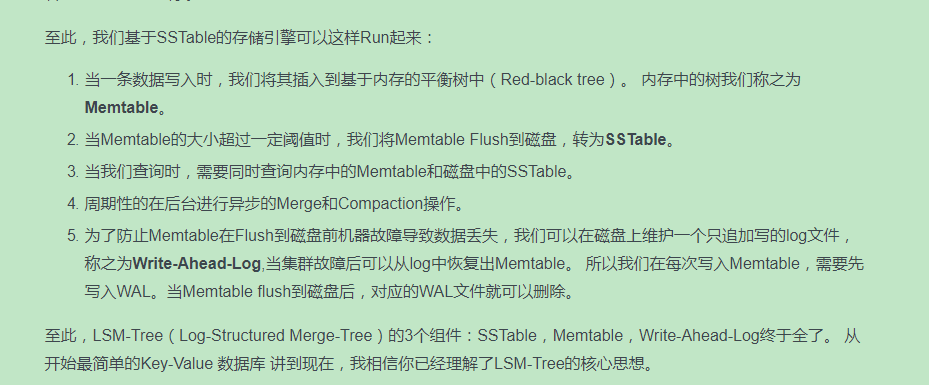
1. 在索引无法全部放入内存的情况下，以B+树的形式将索引存储在磁盘上，可以尽量降低索引高度，减少IO次数。（本题目调研后索引可以全部加载到内存，不合适）
2. 形成的B+树页内的数据有序，页间的数据有指针相连，便于进行范围查找。

本题目如果以B+树的形式组织key-value值(即非叶子节点存储key值，叶子结点存储key-value值)，缺点有以下几个：

一、随机写阶段写的同时要维护B+树索引，索引无法全部装入内存(将key值包装为java中的对象之后远大于8B，6400w个key超出内存能容纳的范围)且写为文件内的随机写，还会经常涉及到叶子结点的分裂等操作，引起数据在文件内的大量迁移。

二、实现复杂度高，实现了之后很可能不如顺序写随机读的方案。

***最终实现方案借鉴了LSM树（***LevelDb、RocksDb中使用***）的思想：***



借鉴LSM树的思想(它是比较通用的思想，根据比赛做了简化)，主要借鉴了两点：将随机写转化为顺序写，key-value存储分离；使用内存文件映射(mappedByteBuffer)存储索引，实现WAL，防止kill -9数据丢失。读取阶段直接随机读，将索引全部加载到内存(分析题目后发现索引通过HashMap或者二分查找的方式是可以完全加载到内存中的)，检索索引（hash查找或者二分查找）查找数据。

SSD的特性：顺序IO高吞吐，随机读并发性能出色。顺序写2.2G/s，顺序读2.5G/s，4K随机读 55w IOPS，大概2.1G/s，实测大概2.35G，4K随机写55w IOPS大概2.1G/s。

数据总量251G(数据250g，索引1g)，一共三个阶段，理论计算每个阶段耗时为：

顺序写：251/2.2=114s,4K随机读：6400w/55w=116.36，顺序读：251/2.5=100.4，理论极限耗时为:430秒左右，完赛后发现硬盘的速率会比官方给出的还要高一些。

我的各阶段时间：顺序写118s-没有打满。16k dio写可以更快，随机读108s-基本打满，顺序读234s

如何榨干IO达到最大的吞吐呢？

根据官方给出的数据并且和阿里云的说法总结在一起可以得到4K随机读写与4K顺序读写差别不大，并且经过我自己的测试可知，SSD 4K随机读与大块顺序读，并发读取是可能都是可能获得性能的提升的，具体并发数量则需要测试得出。

1. 为什么顺序写阶段采用现在的架构设计方案？

方案一、完全无锁化方案：每个线程分别写自己的数据文件与索引文件，每个索引文件中100w个key，每个数据文件中100w个value，这样可以做到完全无锁化；为了在这个方案的基础之上实现相同的key后写入的value要覆盖先写入的value，把索引文件中key对应的value 64位划分为3段，前6位存储数据文件号，中间38位存储写入时间戳，最后20位存储在文件中的写入顺序(实际存储位置为写入顺序\*4K)

方案二、有锁化方案：64个线程写同一个索引文件（直接映射一个大的MappedByteBuffer），64个线程向同一个索引文件中写数据；为了减少锁冲突，需要对数据进行均匀分片(ConcurrentHashMap分片方式或者高10位路由)，每个分片对应一个数据文件，比如将MappedByteBuffer划分为64个分片。对应64个数据文件，写索引中每个分片内的索引时，获取分片对应的锁即可，可减少锁冲突。

由于方案一完全没有锁，所以采用了方案一。最大的败笔！！！

3、Leveldb、rocksdb与redis、memcached等的区别？

Leveldb、rocksdb仅是存储引擎，redis、memcached是完备的数据库。

4、实现RingBuffer的关键：

一、引用计数，判断此buffer是否已经被64个线程访问过

二、buffer状态标识：填充完等待被range线程读取、填充未完成、正在被读取

假设4个buffer，2个线程并发填充，64个线程对一个buffer并发读取。可以令线程1填充1,3号buffer，线程2填充2,4号buffer；64个线程读取某个buffer时会阻塞要填充这个buffer的线程；填充某个buffer时也会阻塞读取线程；对于每个处于正在被读取状态的buffer，记录读取过的线程，防止重复读取。读取完成之后清空记录，将状态转变为填充未完成即可。

5、你觉得Java相比C++语言上的劣势体现在哪里？

1）不够贴近底层，隔着一层JVM，靠JVM解释字节码执行，虽然有JIT帮热点代码优化为native code执行，但终究不够直接。

2）有GC开销，只能尽量减少。

3）使用操作系统API不够方便，比如direct io原生JDK不支持，mmap释放内存不方便，线程bind CPU core不方便等。

# 西电活力工作室校招管家

## 项目简介：

工作室开发的一款微信小程序，致力于解决大学生求职过程中存在的校招信息收集、记录麻烦，校招进度难以共享的问题。

## 项目问题解决及实现：

1. 整合SpringBoot、Spring、SpringMVC、Mybatis、Quartz框架，实现了核心业务逻辑代码的编写。
2. 为了支持集群化部署，使用Redis实现了分布式Session，使用Quartz实现了分布式定时任务调度。
3. 使用Git管理代码，Apizza管理API接口，方便团队开发。

## 可能遇到的问题解答：

# IT技能

具备扎实的Java基础，理解OOP编程思想

熟练应用SpringMVC、Spring、Mybatis、Spring Boot等主流开源框架，并阅读过分析过部分源码

熟悉JVM相关知识，阅读分析过JDK集合包、并发包源码

熟悉常用数据结构和算法、基本的设计模式

熟悉MySQL，了解基本的Sql优化，对Redis有一定的了解

熟悉计算机网络相关知识，了解Linux操作系统

熟悉Maven、Git，具有一定的项目实战经验

了解Docker，能使用Rancher+Docker对微服务应用进行部署

## 可能遇到的问题解答：

# 获奖情况

2018年天池首届阿里云PolarDB数据库性能大赛初赛30/1653，复赛34/200

2018年全国移动终端开发大赛校级二等奖

2018年第四届互联网+创新创业大赛校赛银奖

2017-2018学年西安电子科技大学研究生一等奖学金

2018年西安电子科技大学优秀研究生干部、优秀研究生

2013-2017年间获得国家励志奖学金一次，校级一等奖学金两次

2015年西安电子科技大学程序设计竞赛二等奖

# 自我总结

善于学习，喜欢专研技术

为人诚恳，具有团队精神

勤奋务实，做事认真负责