# 自我介绍

12年毕业，14年的时候开始从事这一行。入行第一个接手的项目就是移动公司的软件应用商城，就是给用户下安卓应用的平台，当时做的是主页展示这一块，每到节假日会做一些营销活到，比如说玩个有些送流量这些。在这个公司做了两年多，然后跳槽外包到顺丰，进的IBU研发部，做一个代理商管理系统，这就是一个卖货系统，顺丰走不了的国际件都就是往国外发的货物基本上都通过这个系统向对应国家的货运公司下单，比如美国的UPS，英国的皇家邮政，下完单之后再打印他们的标签，再把货给他们走。后面就进来上一家公司，主要是给线下的商铺做贷款的。

# 项目总结

## 慧眼风控系统总结：

**项目描述：**

线下贷前审批系统，项目包含app端，pc端及后端，主要包含用户管理，系统管理，订单管理，日常管理，贷款营销，贷款调查，贷款审核，贷款审批等模块

系统管理：主要包含设备轨迹管理。

订单管理：可以查看订单流程，可查询和操作订单状态。

日常管理：包括月度目标划分，工作日清等。工作分配一般就是贷款调查分配，工作日清包括工作成果和工作总结录入。

贷款营销：包括客户申请登记，征信查询，调查预约及任务分配。

贷款调查：分为贷款问卷，调查详情，贷款评分，审批报告，多媒体信息采集模块

贷款问卷：根据不同行业为用户提供标准的调查模型，调查分为借款人及企业信息、担保人信息、行业信息三大部分组成，调查以问卷的方式进行标准化作业。借款主体问卷包括个人姓名住址，名下是否有车辆或房产，家里有多少人，家庭收入大约多少之类的，企业信息一般就是企业信息。行业信息比如建材这一行业，可以问从事这一行的时间，淡旺季的月销售额，库存，销售模式，进货量等情况。

调查详情：调查详情是对调查问卷的整理和还原，整理后的信息以可编辑表格的形式呈现，用户可通过整理后表格对客户调查信息进行核对和修改。

贷款评分：根据调查问卷、调查详情、贷款申请等信息使用系统中的分析模型进行解析，获得“贷款综合评分”、“综合偏离度”、“建议最高额度”三项结果,并对风险点、偏离度超过限制的数据项进行提示，为贷款审批提供参考依据。比如说个人这一方面也有评分，男的评分比女的高，已婚的评分比未婚的高，评分规则可以修改，并且每一项占总评分的比例也可以修改。

审批报告：系统根据贷款详情自动生成电子版调查报告（pdf版）

**主要负责**订单管理，贷款调查模块（贷款问卷，调查详情，贷款评分，审批报告）。

**项目主要流程**：订单录入，然后调查预约，调查预约后会有专门的客户经理上门进行调查，调查以问卷的方式进行，问卷内容是根据不同行业为用户提供标准的调查模型，它主要分为三个模块：借款人及企业信息、担保人信息、行业信息。借款主体问卷包括个人姓名住址，名下是否有车辆或房产，家里有多少人，家庭收入大约多少之类的，企业信息一般就是企业信息。行业信息比如建材这一行业，可以问从事这一行的时间，淡旺季的月销售额，库存，销售模式，进货量等情况。问卷信息会持久化到数据库，然后在问卷提交的时候做一些整理到调查详情里面。调查详情是对调查问卷的整理和还原，整理后的信息以可编辑表格的形式呈现，用户可通过整理后表格对客户调查信息进行核对和修改。一般做调查问卷时会从客户那里拿一些纸质的证明，比如银行流水，出货订单这些，这些都可以在调查详情里面录入。贷款详情之后就是贷款评分环节，根据调查问卷、调查详情、贷款申请等信息使用系统中的分析模型进行解析，获得“贷款综合评分”、“综合偏离度”、“建议最高额度”三项结果,并对风险点、偏离度超过限制的数据项进行提示，为贷款审批提供参考依据。比如说个人这一方面也有评分，男的评分比女的高，已婚的评分比未婚的高，评分规则可以修改，并且每一项占总评分的比例也可以修改。

最后就是整理成审批报告，审批人审批之后会到放款中心，这个系统的流程就结束了。

**贷款流程：**

一、用户注册

二、实名认证

1. 用户身份要素认证（姓名、身份证号，银行卡号，银行预留手机号）

2. 人脸识别（通过前置摄像头采集用户人脸信息，并调用第三方人脸识别认证系统获取人脸匹配分数）

3. 活体检测（活体检测同样是在人脸识别环节完成，一般要求用户完成“眨眼”“摇头”“点头”等必要动作）

三、额度申请

四、信用申请资料

1. 征信报告认证（但征信报告获取流程较为复杂，要求用户注册并登录征信中心，通过问题验证/短信快捷验证。24小时之内用户收到征信中心的身份验证码后，需返回借款平台输入，才能成功获取征信报告）

2. 工资卡流水认证（通过模拟用户登录工资卡网银账号，爬取用户的流水信息，并筛选出其中为工资收入的流水条目，以此获取用户工资信息。）

3. 信用卡认证（通过模拟用户登录信用卡网银信息，爬取信用卡消费与还款记录，从而获取用户信用卡有无逾期与透支情况。）

4. 社保、公积金认证（需模拟用户登录社保与公积金中心账号，平台爬取用户缴纳社保及公积金信息，对用户是否持续缴纳、缴纳额度等信息进行获取。）

5. 电商信息认证（通过获取用户个人的电商账号（京东、淘宝等）授权，来爬取用户的有效购物记录，可判断用户的购买能力，并获知用户的真实所在地等信息。)

6. 手机运营商信息（用户需要输入手机号及运营商服务密码，平台以此获取用户近期的通话详单记录。）

7. 微信信息（获得用户微信支付信息等。）

五、个人信息（用户需补充个人居住地址、工作单位与地址、家庭联系人、工作联系人、紧急联系人等，为人工审批及贷后催收提供足够的信息。）

## 代理商管理系统总结

代理商管理系统操作流程

从kafka获取订单数据（订单数据会重复推送），重复推送的数据会先查询缓存有没有条数据，有的话直接以这个订单id为key，设置分布式锁，查询数据库是否有这条数据，并且判断订单状态为未下单，邮编匹配（根据地址进行匹配），代理选择（根据国家选择代理，然后再根据地区选择合适的代理公司），更新数据库，释放锁。如果这个环节邮编未能匹配到或未选择到代理，之后还可以手动选择，下单（下单会根据选择的代理商调用对应的下单流程，点击下单后，先是获取分布式锁，获取失败直接返回失败，成功的话首先更新订单状态，然后调用邮编，地址校验接口，组装json数据的报文，进行下单，下单成功后保存账单及标签数据，修改订单状态）。

# Spring

## BeanFactory与ApplicationiContext

**Spring容器(Bean工厂)可简单分成两种**

BeanFactory：这是最基础、面向Spring的, 最常用的BeanFactory 实现是XmlBeanFactory 类。

ApplicationContext：这是在BeanFactory基础之上，面向使用Spring框架的开发者。提供了一系列的功能！

**BeanFactory 体系:**BeanFactory 是一个bean容器，它里面维护着一个BeanDefinition map ，BeanDefinition 用来描述 Spring 中的 Bean 对象。

BeandefinitionReader体系:BeanDefinitionReader 的作用是读取 Spring 的配置文件的内容，并将其转换成 Ioc 容器内部的数据结构：BeanDefinition。

**ApplicationContext体系**:它继承 BeanFactory其主要区别有：继承 MessageSource，提供国际化的标准访问策略。继承 ApplicationEventPublisher ，提供强大的事件机制。扩展 ResourceLoader，可以用来加载多个 Resource，可以灵活访问不同的资源。对 Web 应用的支持。

**ApplicationContext和BeanFactory不同之处在于**

ApplicationContext会利用Java反射机制自动识别出配置文件中定义的BeanPostProcessor、 InstantiationAwareBeanPostProcesso 和BeanFactoryPostProcessor后置器，并自动将它们注册到应用上下文中。而BeanFactory需要在代码中通过手工调用addBeanPostProcessor()方法进行注册

ApplicationContext在初始化应用上下文的时候就实例化所有单实例的Bean。而BeanFactory在初始化容器的时候并未实例化Bean，直到第一次访问某个Bean时才实例化目标Bean。

**使用Spring框架的好处是什么？**

**轻量**：Spring 是轻量的，基本的版本大约2MB

**控制反转**：Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们

**面向切面的编程(AOP)**：Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开

容器：Spring 包含并管理应用中对象的生命周期和配置

**MVC框架**：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品

**事务管理**：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）

**异常处理**：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常

**Spring由哪些模块组成？**

以下是Spring 框架的基本模块：Core module;Bean moduleContext module;Expression Language module;JDBC module;ORM module;OXM module;Java Messaging Service(JMS) module;Transaction module;Web module;Web-Servlet module;Web-Struts module;Web-Portlet module;

**ApplicationContext通常的实现是什么？**

FileSystemXmlApplicationContext ：此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。

WebXmlApplicationContext：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

## IOC 及 bean的生命周期

**Spring IOC容器是怎么实现对象的创建和依赖的：**

容器启动时ResourceLoader从存储介质中加载Spring配置信息，并使用Resource表示这个配置文件的资源；BeanDefinitionReader会读取Resource所指向的配置文件资源，然后解析配置文件。配置文件中每一个<bean>解析成一个BeanDefinition对象，并保存到BeanDefinitionRegistry中；

容器扫描BeanDefinitionRegistry中的BeanDefinition；通过反射进行Bean实例化的工作；使用BeanWrapper完成Bean属性的设置工作；

单例Bean缓存池：Spring 在DefaultSingletonBeanRegistry类中提供了一个用于缓存单实例 Bean的缓存器，它是一个用HashMap实现的缓存器，单实例的Bean以beanName为键保存在这个HashMap中。

**IOC最大的优点是：**最小的代价和最小的侵入性使松散耦合得以实现。

原文：Spring IOC知识点一网打尽！ https://mp.weixin.qq.com/s/TYftOpe6J1Y4EDNDxKKqXg

**有哪些不同类型的IOC（依赖注入）方式**

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

**哪种依赖注入方式你建议使用，构造器注入，还是 Setter方法注入？**

你两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

**解释Spring支持的几种bean的作用域**

Spring框架支持以下五种bean的作用域：

**singleton** : bean在每个Spring ioc 容器中只有一个实例。

**prototype**：一个bean的定义可以有多个实例。

**request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。

**session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

**global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

缺省的Spring bean 的作用域是Singleton。

**解释Spring框架中bean的生命周期**

1、Spring Ioc容器找到关于Bean的定义并实例化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BeanFactroy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、如果销毁Bean实例时如果bean实现了DisposableBean接口，则调用其destroy方法

**什么是Spring的内部bean？**

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

**你可以在Spring中注入一个null 和一个空字符串吗？**

可以。

## AOP

面向切面的编程，或AOP， 是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。

**Aspect 切面**

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。在Spring AOP中，切面通过带有@Aspect注解的类实现。

**在Spring AOP 中，关注点和横切关注的区别是什么？**

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

**连接点**

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

**通知**

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用五种类型的通知：

before：前置通知，在一个方法执行前被调用

after：在方法执行之后调用的通知，无论方法执行是否成功

after-returning：仅当方法成功完成后执行的通知

after-throwing：在方法抛出异常退出时执行的通知

around：在方法执行之前和之后调用的通知

**切点**

切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

**什么是引入？**

引入允许我们在已存在的类中增加新的方法和属性。

**什么是目标对象？**

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

**什么是代理？**

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

**什么是织入。什么是织入应用的不同点？**

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。

织入可以在编译时，加载时，或运行时完成。

**解释基于XML Schema方式的切面实现**

在这种情况下，切面由常规类以及基于XML的配置实现。

**解释基于注解的切面实现**

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

**Spring AOP实现方式**

Spring AOP编程底层通过的是动态代理技术，在调用的时候肯定用的是代理对象。那么Spring是怎么做的呢？我只需要写一个BPP，在postProcessBeforeInitialization或者postProcessAfterInitialization方法中，对对象进行判断，看他需不需要织入切面逻辑，如果需要，那我就根据这个对象，生成一个代理对象，然后返回这个代理对象，那么最终注入容器的，自然就是代理对象了。

Spring提供了BeanPostProcessor，就是让我们可以对有需要的对象进行“加工处理”啊！

BBP的全称叫做：BeanPostProcessor，一般我们俗称对象后处理器。简单来说，通过BeanPostProcessor可以对我们的对象进行“加工处理”。

## Spring事务

Spring支持两种类型的事务管理：

编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

Spring事务的传播属性

1) **REQUIRED** ，这个是默认的属性,如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

2) **MANDATORY** 支持当前事务，如果当前没有事务，就抛出异常。

3) **NEVER** 以非事务方式执行，如果当前存在事务，则抛出异常。

4) **NOT\_SUPPORTED** 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

5) **REQUIRES\_NEW** 新建事务，如果当前存在事务，把当前事务挂起。

6) **SUPPORTS** 支持当前事务，如果当前没有事务，就以非事务方式执行。

7) **NESTED** 支持当前事务，新增Savepoint点，与当前事务同步提交或回滚。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。

PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别

它们非常 类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA 事务管理器的支持。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

**Spring框架的事务管理有哪些优点？**

它为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。

它为编程式事务管理提供了一套简单的API而不是一些复杂的事务API如

它支持声明式事务管理。

它和Spring各种数据访问抽象层很好得集成。

**你更倾向用那种事务管理类型？**

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

**认识Spring事务几个重要的接口**

在编程式事务中有以下几个重要的了接口：

TransactionDefinition：定义了Spring兼容的事务属性(比如事务隔离级别、事务传播、事务超时、是否只读状态)

TransactionStatus：代表了事务的具体运行状态(获取事务运行状态的信息，也可以通过该接口间接回滚事务等操作)

PlatformTransactionManager：事务管理器接口(定义了一组行为，具体实现交由不同的持久化框架来完成---类比JDBC)

在声明式事务中，除了TransactionStatus和PlatformTransactionManager接口，还有几个重要的接口：

TransactionProxyFactoryBean：生成代理对象

TransactionInterceptor：实现对象的拦截

TransactionAttrubute：事务配置的数据

## Spring事务失效情况

Spring事务失效的5种大的原因:

1. 如使用mysql且引擎是MyISAM，则事务会不起作用，原因是MyISAM不支持事务，可以改成InnoDB

2. 如果使用了spring+mvc，则context:component-scan重复扫描问题可能会引起事务失败。

3. @Transactional 注解开启配置，必须放到listener里加载，如果放到DispatcherServlet的配置里，事务也是不起作用的。

4. @Transactional 注解只能应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错，事务也会失效。

5. Spring团队建议在具体的类（或类的方法）上使用 @Transactional 注解，而不要使用在类所要实现的任何接口上。在接口上使用 @Transactional 注解，只能当你设置了基于接口的代理时它才生效。因为注解是 不能继承的，这就意味着如果正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装。

需要注意的事项:

在业务代码中，有如下两种情况，比如：

throw new RuntimeException(“xxxxxxxxxxxx”); 事务回滚

throw new Exception(“xxxxxxxxxxxx”); 事务没有回滚

spring内部catch的是 RuntimeException, service抛出RuntimeException可以回滚。如果抛出Exception，就不回滚….

checked异常,即Exception try{}捕获的不会回滚，基于注解的事务：Transactional的异常控制，默认是Check Exception 不回滚，unCheck Exception回滚。如果配置了rollbackFor 和 noRollbackFor 且两个都是用同样的异常，那么遇到该异常，还是回滚。rollbackFor 和noRollbackFor 配置也许不会含盖所有异常，对于遗漏的按照Check Exception 不回滚，unCheck Exception回滚

**Spring中 @Transactional 注解的限制**

1. 同一个类中, 一个nan-transactional的方法去调用transactional的方法, 事务会失效

2. 在private方法上标注transactional, 事务无效

结论

@Transactional 加于private方法, 无效

@Transactional 加于未加入接口的public方法, 再通过普通接口方法调用, 无效

@Transactional 加于接口方法, 无论下面调用的是private或public方法, 都有效

@Transactional 加于接口方法后, 被本类普通接口方法直接调用, 无效

@Transactional 加于接口方法后, 被本类普通接口方法通过接口调用, 有效

@Transactional 加于接口方法后, 被它类的接口方法调用, 有效

@Transactional 加于接口方法后, 被它类的私有方法调用后, 有效

总结: Transactional是否生效, 仅取决于是否加载于接口方法, 并且是否通过接口方法调用(而不是本类调用)

## Spring中的设计者模式

**Spring 框架中用到了哪些设计模式：**

**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 AOP 功能的实现。

**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。

**工厂模式**

Spring使用工厂模式可以通过 BeanFactory 或 ApplicationContext 创建 bean 对象。

两者对比：

BeanFactory ：延迟注入(使用到某个 bean 的时候才会注入),相比于BeanFactory来说会占用更少的内存，程序启动速度更快。

ApplicationContext ：容器启动的时候，不管你用没用到，一次性创建所有 bean 。BeanFactory 仅提供了最基本的依赖注入支持，ApplicationContext 扩展了 BeanFactory ,除了有BeanFactory的功能还有额外更多功能，所以一般开发人员使用ApplicationContext会更多。

**单例设计模式**

Spring 中 bean 的默认作用域就是 singleton(单例)的。

**代理设计模式(AOP)**

分JDK Proxy代理与CGLIB代理

AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

**Spring AOP 和 AspectJ AOP 有什么区别?**

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)

**模板方法**

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。一般情况下，我们都是使用继承的方式来实现模板模式，但是 Spring 并没有使用这种方式，而是使用Callback 模式与模板方法模式配合，既达到了代码复用的效果，同时增加了灵活性。

public abstract class Template {

//这是我们的模板方法

public final void TemplateMethod(){

PrimitiveOperation1();

PrimitiveOperation2();

}

protected void PrimitiveOperation1(){

//当前类实现

}

//被子类实现的方法

protected abstract void PrimitiveOperation2();

}

public class TemplateImpl extends Template {

@Override

public void PrimitiveOperation2() {

//当前类实现

}

}

**观察者模式**

Spring 的事件流程总结:

定义一个事件: 实现一个继承自 ApplicationEvent，并且写相应的构造函数；

定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；

使用事件发布者发布消息: 可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息。

**适配器模式**

适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。

spring AOP中的适配器模式:我们知道 Spring AOP 的实现是基于代理模式，但是 Spring AOP 的增强或通知(Advice)使用到了适配器模式，与之相关的接口是AdvisorAdapter 。Advice 常用的类型有：BeforeAdvice（目标方法调用前,前置通知）、AfterAdvice（目标方法调用后,后置通知）、AfterReturningAdvice(目标方法执行结束后，return之前)等等。每个类型Advice（通知）都有对应的拦截器:MethodBeforeAdviceInterceptor、AfterReturningAdviceAdapter、AfterReturningAdviceInterceptor。Spring预定义的通知要通过对应的适配器，适配成 MethodInterceptor接口(方法拦截器)类型的对象（如：MethodBeforeAdviceInterceptor 负责适配 MethodBeforeAdvice）。

spring MVC中的适配器模式:在Spring MVC中，DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由HandlerAdapter 适配器处理。HandlerAdapter 作为期望接口，具体的适配器实现类用于对目标类进行适配，Controller 作为需要适配的类。

**装饰者模式**

Spring 中配置 DataSource 的时候，DataSource 可能是不同的数据库和数据源。我们能否根据客户的需求在少修改原有类的代码下动态切换不同的数据源？这个时候就要用到装饰者模式(这一点我自己还没太理解具体原理)。Spring 中用到的包装器模式在类名上含有 Wrapper或者 Decorator。这些类基本上都是动态地给一个对象添加一些额外的职责

## spring配置解析

**@Configuration有什么用？**

加入@Configuration 注解，表明这就是一个配置类。有一个myBean()的方法，返回一个MyBean()的实例，并用@Bean进行注释，表明这个方法是需要被Spring进行管理的bean。

**@Configuration和XML有什么区别？哪种好？**

哪种好说不上，但spring提倡用代码方式配置。

**Spring是如何基于来获取Bean的定义的？**

基于ComponentScan() 来获取Bean的定义，@Controller, @Service, @Repository, @Component四个注解标记的类都能够通过@ComponentScan 扫描到，

**@Autowired 、 @Inject、@Resource 之间有什么区别？**

@Autowired和@Inject基本是一样的，因为两者都是使用AutowiredAnnotationBeanPostProcessor来处理依赖注入。但是@Resource是个例外，它使用的是CommonAnnotationBeanPostProcessor来处理依赖注入。当然，两者都是BeanPostProcessor。

**@Value、@PropertySource 和 @Configuration？**

@Configuration 可以和@Value 和@PropertySource 一起使用读取外部配置文。通过@PropertySource引入的配置文件，使@Value 能够获取到属性值，在给myBean()方法指定了一个名称叫做myTestBean。

**Spring如何处理带@Configuration @Import的类？**

@Import的定义(来自于JavaDoc)：表明一个或者多个配置类需要导入，提供与Spring XML中相等的功能，允许导入@Configuration 、@ImportSelector、@ImportBeanDefinitionRegistar的实现，以及常规组件类似于AnnotationConfigApplicationContext。可能用于类级别或者是原注解。如果XML或者其他非@Configuration标记的Bean资源需要被导入的话，使用@ImportResource。

**@Profile有什么用？**

@Profile: 表示当一个或多个@Value 指定的配置文件处于可用状态时，组件符合注册条件，可以进行注册。

三种设置方式：

可以通过ConfigurableEnvironment.setActiveProfiles()以编程的方式激活

可以通过AbstractEnvironment.ACTIVEPROFILESPROPERTY\_NAME (spring.profiles.active )属性设置为 JVM属性

作为环境变量，或作为web.xml 应用程序的Servlet 上下文参数。也可以通过@ActiveProfiles 注解在集成测试中以声明方式激活配置文件。

作用域

作为类级别的注释在任意类或者直接与@Component 进行关联，包括@Configuration 类

作为原注解，可以自定义注解

作为方法的注解作用在任何方法

注意:

如果一个配置类使用了Profile 标签或者@Profile 作用在任何类中都必须进行启用才会生效，如果@Profile({"p1","!p2"}) 标识两个属性，那么p1 是启用状态 而p2 是非启用状态的。

**@Configuration 如何嵌套？**

@ImportResource: 这个注解提供了与@Import 功能相似作用，通常与@Configuration 一起使用，通过AnnotationConfigApplicationContext 进行启动，

**Spring如何对Bean进行延迟初始化？**

@Lazy : 表明一个bean 是否延迟加载，可以作用在方法上，表示这个方法被延迟加载；可以作用在@Component (或者由@Component 作为原注解) 注释的类上，表明这个类中所有的bean 都被延迟加载。如果没有@Lazy注释，或者@Lazy 被设置为false，那么该bean 就会急切渴望被加载；除了上面两种作用域，@Lazy 还可以作用在@Autowired和@Inject注释的属性上，在这种情况下，它将为该字段创建一个惰性代理，作为使用ObjectFactory或Provider的默认方法。

**Spring项目怎么进行单元测试？**

@RunWith 和 @ContextConfiguration

Junit4 测试类，用于注解在类上表示通过Junit4 进行测试，可以省略编写启动类代码，是ApplicationContext 等启动类的替换。一般用@RunWith 和 @Configuration 进行单元测试，这是软件开发过程中非常必要而且具有专业性的一部分

**@Configuration 使用上有哪些约束？**

必须以类的方式提供(即不是从工厂方法返回的实例)

@Configuration 注解的类必须是非final的

配置类必须是非本地的（即可能不在方法中声明）,native 标注的方法

任何嵌套的@Configuration 都必须是static 的。

@Bean 方法可能不会反过来创建更多配置类

# SpringMVC

## springMVC和struts2的区别有哪些?

（1）springmvc的入口是一个servlet即前端控制器（DispatchServlet），而struts2入口是一个filter过虑器（StrutsPrepareAndExecuteFilter）。

（2）springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

（3）Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

## SpringMVC工作流程描述

请求被Spring 前端控制Servelt DispatcherServlet捕获后，DispatcherServlet对请求URL进行解析，得到请求资源标识符，然后根据该URI，调用HandlerMapping获得对应的Handler

（包括Handler对象以及Handler对象对应的拦截器），最后以 HandlerExecutionChain对象的形式返回； DispatcherServlet 根据获得的Handler，获取一个合适的HandlerAdapter。

提取Request中的模型数据，填充Handler入参，HandlerAdapter会调用自己的handle方法，handle方法运用java的反射机制执行controller的具体方法。Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的 ViewResolver)返回给DispatcherServlet ； ViewResolver 结合Model和View，来渲染视图。将渲染结果返回给客户端。

## 其它

**Spring MVC的异常处理 ？**

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

**SpringMVC的控制器是不是单例模式,如果是,有什么问题,怎么解决？**

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

**SpringMVC里面拦截器是怎么写的？**

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

# SpringBoot

**什么是 Spring Boot？**

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

**Spring Boot解决哪些问题？**

Spring Boot使编码，配置，部署，监控变简单。

**Spring Boot 优点**

独立运行，简化配置，自动配置，无代码生成和XML配置，应用监控，上手容易

**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 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；

**Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？**

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

**开启 Spring Boot 特性有哪几种方式？**

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

**Spring Boot 自动配置原理是什么？**

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

**你如何理解 Spring Boot 中的 Starters？**

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。

**如何在 Spring Boot 启动的时候运行一些特定的代码？**

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个 run 方法

**Spring Boot 有哪几种读取配置的方式？**

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量

**SpringBoot 实现热部署有哪几种方式？**

主要有两种方式：Spring Loaded，Spring-boot-devtools

**你如何理解 Spring Boot 配置加载顺序？**

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1）properties文件；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

## SpringBoot启动流程

当执行SpringApplication类的静态run方法时

1、创建SpringApplication实例：

将传入的启动类"com.example.ms.DemoApplication"放入Set集合中

判断是否为Web环境：存在（javax.servlet.Servlet && org.springframework.web.context.ConfigurableWebApplicationContext ）类则是

创建并初始化ApplicationInitializer列表 （spring.factories）

创建并初始化ApplicationListener列表  （spring.factories）

初始化主类mainApplicatioClass    (DemoApplication)

2、启动run方法

创建计时器StopWatch

配置awt系统属性

获取SpringApplicationRunListeners

启动SpringApplicationRunListeners

创建ApplicationArguments

创建并初始化ConfigurableEnvironment

打印Banner

创建ConfigurableApplicationContext

准备ConfigurableApplicationContext

刷新ConfigurableApplicationContext

容器刷新后动作

SpringApplicationRunListeners发布finish事件

计时器停止计时

Spring Boot、Spring MVC 和 Spring 有什么区别？

Spring 是一个“引擎”;

Spring MVC 是基于Spring的一个 MVC 框架;

Spring Boot 是基于Spring4的条件注册的一套快速开发整合包。

# SpringCloud

## SpringCloud生态

**Spring Cloud Config**：配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion。

**Spring Cloud Bus**：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

**Eureka**：云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

**Hystrix**：熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

**Ribbon**：提供云端负载均衡，有多种负载均衡策略可供选择，可配合服务发现和断路器使用。

**Zuul**：Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。

**Feign**：Feign是一种声明式、模板化的HTTP客户端。

**Spring Cloud Task**：提供云端计划任务管理、任务调度。

**Spring Cloud Security**：基于spring security的安全工具包，为你的应用程序添加安全控制。

**Archaius**：配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

**Consul**：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

**Spring Cloud for Cloud Foundry**：通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

**Spring Cloud Sleuth**：日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。

**Spring Cloud Data Flow**：大数据操作工具，作为Spring XD的替代产品，它是一个混合计算模型，结合了流数据与批量数据的处理方式。

**Spring Cloud Zookeeper**：操作Zookeeper的工具包，用于使用zookeeper方式的服务发现和配置管理。

**Spring Cloud Stream**：数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

**Spring Cloud CLI**：基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

**Turbine**：Turbine是聚合服务器发送事件流数据的一个工具，用来监控集群下hystrix的metrics情况。

**Spring Cloud Connectors**：便于云端应用程序在各种PaaS平台连接到后端，如：数据库和消息代理服务。

**Spring Cloud Cluster**：提供Leadership选举，如：Zookeeper, Redis, Hazelcast, Consul等常见状态模式的抽象和实现。

**Spring Cloud Starters**：Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理。

## Spring Cloud微服务如何设计异常处理机制

面向外部提供的服务接口，服务端所有的异常处理我们都要进行相应地捕获，并在controller层映射成相应地错误码和错误信息，返回给调用方。

而面向内部的服务接口，通过FeignClient的方式进行服务调用，Feign内部会集成像Ribbon和Hystrix这样的框架来实现客户端服务调用的负载均衡和服务熔断功能。

**最佳实践设计**

首先，无论是内部还是外部的微服务，在服务端我们都应该设计一个全局异常处理类，用来统一封装系统在抛出异常时面向调用方的返回信息。而实现这样一个机制，我们可以利用Spring提供的注解@ControllerAdvice来实现异常的全局拦截和统一处理功能。从最佳实践上考虑，我们一般会为内部和外部接口分别设计一个统一面向调用方的异常对象。

FeignClient为我们提供了异常解码机制

## 原生Spring-Cloud-Config的配置中心的缺点

**Git的权限控制是个坑**：Git的权限管理是说控制用户能不能Push或者Delete分支，或者能不能Push代码，而不是能不能访问某个目录的文件。

**粒度问题**：你每次对一条配置发生crud的操作，其带来的影响是整个文件发生变动。当时我们最理想的存储介质就是数据库。

**Spring-Cloud-Config的刷新机制是个坑**：因为一个配置中心应该要能够做到，配置发生改动的时候，项目能够自动感知，自动更新配置才对。在Spring-Cloud-Config中，这套机制是借助一些代码仓库（SVN、Github等）提供的Webhook机制加上Spring-Cloud-Bus来实现的。

那么问题又来了！配置数据放在数据库中，数据库里没有Webhook这种东西啊，怎么做到实时刷新？

**长轮询，短轮询**

在页面上要实时显示后台的库存数量！比如库存减少了，用户不需要进行刷新，页面上的数字自己会变化。

如果采取短轮询就是在客户端(js)中不断访问后台，后台接到请求马上返回最新的库存数，然后刷新到这个页面当中。

长轮询是客户端(js)依然是不断的去请求。但是呢，服务端不是马上返回。而是等待库存数量变化了再返回。

怎么实现：我们在项目中采用Spring的DeferredResult来实现。

## 如何有效快速的监听出配置表的数据发生了变动？

因为我们用的是mysql。这里有一个Mysql的自定义函数叫mysql-udf-http。具有http\_get()、http\_post()、http\_put()、http\_delete()四个函数，可以在MySQL数据库中利用HTTP协议进行REST相关操作。

然后再和mysql的触发器结合起来用，可以实现在配置表发生变动的时候，主动通知我们的配置中心服务端。让服务端明白配置发生了变动！

# MySQL

## 谈谈我经历的三个公司的数据库操作

你们数据库的高可用架构是怎么样的？

卓望公司：使用oracle为主库，再使用工具将数据同步到mysql数据库中，运维会实时监控数据库是否宕机或其它异常。

顺丰科技：将订单号作为分库分表的依据，将该订单相关的数据都放到一个库中，然后基础数据冗余，有三个表。缺点是做统计的时候比较麻烦，做不了实时统计。冗余数据只写一个表，然后用etl同步工具同步。

瑞宝麟：贷前系统采用单一的oracle数据库，报表系统采用分表策略，每天一个表

自我感觉在数据库这一方面比较薄弱。听说最近有款分库分表的中间件叫ShardingSphere，还没来得及去看。

数据库主从复制的依据是：

Binlog日志

## 索引

**什么情况下可以建立索引**，如果我的数据访问量都很大那怎么办

在访问量过大或是访问频繁或访问大表中一小部分的时候需要建立索引，以便提高效率可以对该表进行分区，减少I/O操作

**索引的分类**

**单列索引**：一个索引只包含单个列，但一个表中可以有多个单列索引。 这里不要搞混淆了。

**普通索引**：MySQL中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

**唯一索引**：索引列中的值必须是唯一的，但是允许为空值，

**主键索引**：是一种特殊的唯一索引，不允许有空值。（主键约束，就是一个主键索引）

**组合索引**

**全文索引**：全文索引，只有在MyISAM引擎上才能使用，只能在CHAR,VARCHAR,TEXT类型字段上使用全文索引

整个表变成了一个索引，也就是所谓的「聚集索引」

**索引实现原理**

**[数据库](http://lib.csdn.net/base/14" \t "_blank" \o "MySQL知识库)索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

## 数据库锁

**mysql锁进行划分：**

按照锁的粒度划分：行锁、表锁、页锁

按照锁的使用方式划分：共享锁、排它锁（悲观锁的一种实现）

还有两种思想上的锁：悲观锁、乐观锁。

InnoDB中有几种行级锁类型：Record Lock、Gap Lock、Next-key Lock

Record Lock：在索引记录上加锁

Gap Lock：间隙锁

Next-key Lock：Record Lock+Gap Lock

**1.行锁**

​**行级锁是Mysql中锁定粒度最细的一种锁**，表示只针对当前操作的行进行加锁。其加锁粒度最小，但加锁的开销也最大。有可能会出现死锁的情况。行级锁按照使用方式分为共享锁和排他锁。

**共享锁用法（S锁 读锁）**：

若事务对数据对象加了读锁，则该事务只能读不能修改该数据，其它事务只能再对该对象加读锁，而不能加写锁。(select ... lock in share mode;)

**排它锁用法（X 锁 写锁）**：

若事务对数据对象加了写锁，改事务可以读也可以修改该对象，但其它事务不能对该数据加任何锁。(select ... for update)

排它锁，也称作独占锁，一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁。

**2.表锁**

​**表级锁是mysql锁中粒度最大的一种锁**，表示当前的操作对整张表加锁，资源开销比行锁少，不会出现死锁的情况，但是发生锁冲突的概率很大。被大部分的mysql引擎支持，MyISAM和InnoDB都支持表级锁，但是InnoDB默认的是行级锁。

共享锁用法： LOCK TABLE table\_name [ AS alias\_name ] READ

排它锁用法：LOCK TABLE table\_name [AS alias\_name][ LOW\_PRIORITY ] WRITE

解锁用法：unlock tables;

**3.页锁**

​**页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁**。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁

**4.1悲观锁**

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度(悲观)，因此，在整个数据处理过程中，将数据处于锁定状态。 悲观锁的实现，往往依靠数据库提供的锁机制

悲观锁的具体流程：

在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

**在mysql/InnoDB中使用悲观锁**：

首先我们得关闭mysql中的autocommit属性,begin;/begin work;/start transaction; (三者选一就可以)开始事务,commit;/commit work;提交事务

**4.2乐观锁**

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

实现数据版本有两种方式，第一种是使用版本号，第二种是使用时间戳。

**乐观锁的优点和不足：**

​ 乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

**MySQL/InnoDB中的行锁和表锁问题**

​首先我们知道InnoDB默认支持的是行锁，但这并不代表InnoDB不支持表锁。必须明白这一点在InnoDB中并不是在数据行上加锁，而是在对应的索引上加锁，这一点和oracle并不同，后者是在数据行上加锁的。这种实现的特点是：只有通过索引条件检索数据的时候加的是行锁，否则加表锁！

在不通过索引条件查询的时候，InnoDB使用的确实是表锁！（因为MySQL 的行锁是针对索引加的锁,不是针对记录加的锁不论 是使用主键索引、唯一索引或普通索引,InnoDB 都会使用行锁来对数据加锁。）

即便在条件中使用了索引字段,但是否使用索引来检索数据是由 MySQL 通过判断不同 执行计划的代价来决定的,如果 MySQL 认为全表扫 效率更高,比如对一些很小的表,它 就不会使用索引,这种情况下 InnoDB 将使用表锁,而不是行锁（比如全表更新和多表查询时）。因此,在分析锁冲突时, 别忘了检查 SQL 的执行计划,以确认是否真正使用了索引。

当使用索引检索数据时不同事务可以操作不同行数据

**表锁**

表锁的优势：开销小；加锁快；无死锁

表锁的劣势：锁粒度大，发生锁冲突的概率高，并发处理能力低

加锁的方式：自动加锁。查询操作（SELECT），会自动给涉及的所有表加读锁，更新操作（UPDATE、DELETE、INSERT），会自动给涉及的表加写锁。也可以显示加锁：

共享读锁：lock table tableName read;

独占写锁：lock table tableName write;

批量解锁：unlock tables;

MyISAM的表锁，读锁会阻塞写，不会阻塞读。而写锁则会把读写都阻塞。

**查看加锁情况**

show open tables; 1表示加锁，0表示未加锁。

分析表锁定

可以通过检查table\_locks\_waited 和 table\_locks\_immediate 状态变量分析系统上的表锁定：show status like ‘table\_locks%’

table\_locks\_immediate: 表示立即释放表锁数。

table\_locks\_waited: 表示需要等待的表锁数。此值越高则说明存在着越严重的表级锁争用情况。

此外，MyISAM的读写锁调度是写优先，这也是MyISAM不适合做写为主表的存储引擎。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永久阻塞。

**6.Record Lock、Gap Lock、Next-key Lock锁**

6.1.Record Lock

单条索引上加锁，record lock 永远锁的是索引，而非数据本身，如果innodb表中没有索引，那么会自动创建一个隐藏的聚集索引，锁住的就是这个聚集索引。所以说当一条sql没有走任何索引时，那么将会在每一条聚集索引后面加X锁，这个类似于表锁，但原理上和表锁应该是完全不同的。

**分析行锁定**

通过检查InnoDB\_row\_lock 状态变量分析系统上的行锁的争夺情况 show status like ‘innodb\_row\_lock%’

innodb\_row\_lock\_current\_waits: 当前正在等待锁定的数量

innodb\_row\_lock\_time: 从系统启动到现在锁定总时间长度；非常重要的参数

innodb\_row\_lock\_time\_avg: 每次等待所花平均时间；非常重要的参数，

innodb\_row\_lock\_time\_max: 从系统启动到现在等待最常的一次所花的时间；

innodb\_row\_lock\_waits: 系统启动后到现在总共等待的次数；非常重要的参数。直接决定优化的方向和策略。

**行锁优化**

1 尽可能让所有数据检索都通过索引来完成，避免无索引行或索引失效导致行锁升级为表锁。

2 尽可能避免间隙锁带来的性能下降，减少或使用合理的检索范围。

3 尽可能减少事务的粒度，比如控制事务大小，而从减少锁定资源量和时间长度，从而减少锁的竞争等，提供性能。

4 尽可能低级别事务隔离，隔离级别越高，并发的处理能力越低。

6.2.Gap Lock

间隙锁，是在索引的间隙之间加上锁，这是为什么Repeatable Read隔离级别下能防止幻读的主要原因。

幻读：事务A执行了一次读操作，此时事务B在事务A的影响区间内更新了一条数据，此时事务A在执行一次读操作时，会发现出现了不合理的数据。有关幻读的详细解释：https://blog.csdn.net/qq\_38238296/article/details/88363017

**6.2什么叫间隙锁**

​简单的说就是锁住了一个范围的数据

当我们用范围条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做”间隙(GAP)”。InnoDB也会对这个”间隙”加锁，这种锁机制就是所谓的间隙锁(Next-Key锁)。

危害(坑)：若执行的条件是范围过大，则InnoDB会将整个范围内所有的索引键值全部锁定，很容易对性能造成影响。

**幻读**，并不是说两次读取获取的结果集不同，幻读侧重的方面是某一次的 select 操作得到的结果所表征的数据状态无法支撑后续的业务操作。更为具体一些：select 某记录是否存在，不存在，准备插入此记录，但执行 insert 时发现此记录已存在，无法插入，此时就发生了幻读。

**6.2.2 为什么说gap锁是RR隔离级别下防止幻读的主要原因。**

快照读：简单的select操作，没有lock in share mode或for update

当前读：官方文档的术语叫locking read，也就是insert，update，delete,select…in share mode和select…for update，当前读会在所有扫描到的索引记录上加锁，不管它后面的where条件到底有没有命中对应的行记录。

首先了解到InnoDB索引的数据结构是B+树，其索引是有序性的，（具体原理可以看这篇文章：https://blog.csdn.net/qq\_38238296/article/details/88362635 ）如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。注意是当前读。

根据索引的有序性，我们可以从上面的例子推断出满足where条件的数据，只能插入在num=（1,3]U[3,4)两个区间里面，只要我们将这两个区间锁住，那么就不会发生幻读。

**6.3.Next-Key Lock**

这个锁机制其实就是前面两个锁相结合的机制，行锁就是采用这个锁的机制用来防止幻读！

**死锁的原理及分析**

7.1. MVCC

​MySQL InnoDB存储引擎，实现的是基于多版本并发控制协议—MVCC(Multi-Version Concurrency Control) MVCC最大的好处，读不加锁，读写不冲突。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。

7.2. 2PL：Two-Phase Locking

​传统RDBMS（关系数据库管理系统）加锁的一个原则，就是2PL (二阶段锁)：Two-Phase Locking。相对而言，2PL比较容易理解，说的是锁操作分为两个阶段：加锁阶段与解锁阶段，并且保证加锁阶段与解锁阶段不相交。下面，仍旧以MySQL为例，来简单看看2PL在MySQL中的实现。

7.3 为什么会发生死锁

​MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。而在InnoDB中，锁是逐步获得的，就造成了死锁的可能。（不过现在一般都是InnoDB引擎，关于MyISAM不做考虑）

在InnoDB中，行级锁并不是直接锁记录，而是锁索引。索引分为主键索引和非主键索引两种，如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。

当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。

避免死锁，这里只介绍常见的三种

如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

这篇文章关于mysql锁写的很有深度：http://hedengcheng.com/?p=771

## MYSQL事务

**事务的本质**

事务的核心是锁与并发

事务本质：独占某部分数据

事物单元之间的happen-before关系：读写 写读 读读 写写 只有这四种关系.

1、事务--排队法（一个队列）

2、事务--排他锁（多个队列）

3、事务--读写锁（对读读场景的优化）

4、事务--MVCC 本质来说就是copy on write

**事务**：就是作为单个逻辑单元执行的一组操作，要么全成功，要么全失败。

分布式事务用于在分布式系统中保证不同节点之间的数据一致性。

事务常说一系列操作作为一个整体要么都成功要么都失败，主要特性acid，事务的的实现主要依赖两个log redo-log,undo-log,每次事务都会记录数据修改前的数据undo-log，修改后的数据放入redo-log,提出成功则使用redo-log 更新到磁盘，失败则使用undo-log将数据恢复到事务之前的数据

Java事务的类型有三种：JDBC事务、容器事务、JTA(Java Transaction API)事务。

不用事务的编程框架来管理事务，直接使用资源管理器来控制事务。典型的就是java.sql.Connection 中的 setAutoCommit、commit、rollback方法。本地事务比较简单，基本原理就是数据库的事务原理

**本地事务的优点**

支持严格的ACID属性

可靠

高效

状态可以只在资源管理器中维护

应用编程模型简单

**本地事务的局限**

不具备分布式事务处理能力

隔离的最小单位由资源管理器决定，如数据库中的一条记录

**全局事务的定义**：全局事务就是一个标准的分布式事务。全局事务是由资源管理器管理和协调的事务。

全局事务是一个DTP模型的事务，所谓DTP模型指的是X/Open DTP(X/Open Distributed Transaction Processing Reference Model)，是X/Open 这个组织定义的一套分布式事务的标准，也就是了定义了规范和API接口，由这个厂商进行具体的实现。

原文：https://mp.weixin.qq.com/s/QToXYvPzIBW36vAPqp6Ciw

**说一下Innodb和MyISAM的区别**

MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。

InnoDB不支持FULLTEXT类型的索引

InnoDB 中不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含 where条件时，两种表的操作是一样的。

对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引。

DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除。

LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性(例如外键)的表不适用。

## MySQL中事物ACID的原理

**Mysql怎么保证一致性的**

ACID嘛，原子性(Atomicity/,ætə'mɪsɪtɪ/)、一致性(Consistency/kən'sɪst(ə)nsɪ/)、隔离性(Isolation/aɪsə'leɪʃ(ə)n/)、持久性(Durability/djuərə'biləti/)

ACID四大特性之中，C(一致性)是目的，A(原子性)、I(隔离性)、D(持久性)是手段，是为了保证一致性，数据库提供的手段。数据库必须要实现AID三大特性，才有可能实现一致性。

**Mysql怎么保证原子性的**

是利用Innodb的undo log，undo log名为回滚日志，是实现原子性的关键，当事务回滚时能够撤销所有已经成功执行的sql语句，他需要记录你要回滚的相应日志信息。undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

**Mysql怎么保证持久性的**

是利用Innodb的redo log。正如之前说的，Mysql是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。

怎么解决这个问题？简单啊，事务提交前直接把数据写入磁盘就行啊。但是这样太浪费资源了。于是，决定采用redo log解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

采用redo log的好处是进行刷盘比对数据页刷盘效率高，具体表现如下：redo log体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快。redo log是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

**Mysql怎么保证隔离性的**

利用的是锁和MVCC机制。MVCC,即多版本并发控制(Multi Version Concurrency Control),一个行记录数据有多个版本对快照数据，这些快照数据在undo log中。如果一个事务读取的行正在做DELELE或者UPDATE操作，读取操作不会等行上的锁释放，而是读取该行的快照版本。

**事务隔离级别：**

**SQL 标准定义了四个隔离级别**：

•**READ-UNCOMMITTED(读取未提交)**： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

•**READ-COMMITTED(读取已提交):** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

•**REPEATABLE-READ（可重读）**: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

•**SERIALIZABLE(可串行化)**: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

**并发事务带来的问题**

•**脏读（Dirty read）**: 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

•**丢失修改（Lost to modify）**: 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

•**不可重复读（Unrepeatableread）**: 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

•**幻读（Phantom read）**: 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复度和幻读区别：

不可重复读的重点是修改，幻读的重点在于新增或者删除。

例1（同样的条件, 你读取过的数据, 再次读取出来发现值不一样了 ）：事务1中的A先生读取自己的工资为 1000的操作还没完成，事务2中的B先生就修改了A的工资为2000，导 致A再读自己的工资时工资变为 2000；这就是不可重复读。

例2（同样的条件, 第1次和第2次读出来的记录数不一样 ）：假某工资单表中工资大于3000的有4人，事务1读取了所有工资大于3000的人，共查到4条记录，这时事务2 又插入了一条工资大于3000的记录，事务1再次读取时查到的记录就变为了5条，这样就导致了幻读。

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）。我们可以通过SELECT @@tx\_isolation;命令来查看

## 数据同步

**MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么？**

1.1 Row Level 行模式

日志中会记录每一行数据被修改的形式，然后在slave端再对相同的数据进行修改

优点：在row level模式下rowlevel的日志内容会非常清楚的记录下每一行数据修改的细节。不会出现某些特定的情况下的存储过程或function，以及trigger的调用和触发无法被正确复制的问题

缺点：会产生大量的日志内容。

1.2 Statement Level（默认）

每一条会修改数据的sql都会记录到master的bin-log中。slave在复制的时候sql进程会解析成和原来master端执行过的相同的sql来再次执行

优点：statement level减少了bin-log日志量，节约IO。

缺点：由于只记录语句，所以，在statement level下 已经发现了有不少情况会造成MySQL的复制出现问题，主要是修改数据的时候使用了某些定的函数或者功能的时候会出现。

1.3 Mixed 自动模式

在Mixed模式下，MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志格式，也就是在Statement和Row之间选择一种。如果sql语句确实就是update或者delete等修改数据的语句，那么还是会记录所有行的变更。

行模式和语句模式的区别

1.语句模式：

100万条记录

只需1条delete \* from test；就可以删除100万条记录

2.row模式

100万条记录

记录100万条删除命令

1.4 企业场景如何选择binlog模式

1、互联网公司，使用MySQL的功能相对少（存储过程、触发器、函数）

选择默认的语句模式，Statement Level（默认）

2、公司如果用到使用MySQL的特殊功能（存储过程、触发器、函数）

则选择Mixed模式

3、公司如果用到使用MySQL的特殊功能（存储过程、触发器、函数）又希望数据最大化一直，此时最好选择Row level模式

**数据库集群如何保证其每个数据库的数据一致性?**

1.半同步复制：简单的说就是: 主库发生增删改操作的时候,会等从库及时复制了并且通知了主库, 才会把这个操作叫做成功。专业的讲，半同步复制，是等待其中一个从库也接收到Binlog事务并成功写入Relay Log之后，才返回Commit操作成功给客户端。优点：保证数据一致性，优点：保证数据一致性。

2.数据库中间件：所有的读写请求都走中间件，然后写的请求路由到主库，读的请求路由到从库当有一个写请求过来时候，生成一个key A ，马上路由写到主库，然后立马有一个读请求过来。 如果时间是在1s内的，就对应的key继续路由到主库。如果在1s以后的，就路由到从库。说白了，中间件就是给个同步时间，给你同步，在同步时间内，所有的请求都落在主库

3 缓存记录写key法:将某个库上的某个key要发生写操作，记录在cache里，并设置“经验主从同步时间”的cache超时时间，例如1s。然后修改数据库。

用mysql过程中，有遇到什么问题么？

因为字段超长报错这些就不说了吧，比较棘手又常见的是慢查询问题

## MySQL Explain详解

概要描述：

id:选择标识符

select\_type:表示查询的类型。

table:输出结果集的表

partitions:匹配的分区

type:表示表的连接类型

possible\_keys:表示查询时，可能使用的索引

key:表示实际使用的索引

key\_len:索引字段的长度

ref:列与索引的比较

rows:扫描出的行数(估算的行数)

filtered:按表条件过滤的行百分比

Extra:执行情况的描述和说明

二、select\_type

示查询中每个select子句的类型

(1) SIMPLE(简单SELECT，不使用UNION或子查询等)

(2) PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的select被标记为PRIMARY)

(3) UNION(UNION中的第二个或后面的SELECT语句)

(4) DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)

(5) UNION RESULT(UNION的结果，union语句中第二个select开始后面所有select)

(6) SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)

(7) DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)

(8) DERIVED(派生表的SELECT, FROM子句的子查询)

(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

四、type

对表访问方式，表示MySQL在表中找到所需行的方式，又称“访问类型”。

常用的类型有： ALL、index、range、 ref、eq\_ref、const、system、NULL（从左到右，性能从差到好）

ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行

index: Full Index Scan，index与ALL区别为index类型只遍历索引树

range:只检索给定范围的行，使用一个索引来选择行

ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

eq\_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

const、system:当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量，system是const类型的特例，当查询的表只有一行的情况下，使用system

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

五、possible\_keys

指出MySQL能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用（该查询可以利用的索引，如果没有任何索引显示 null）

该列完全独立于EXPLAIN输出所示的表的次序。这意味着在possible\_keys中的某些键实际上不能按生成的表次序使用。

如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查WHERE子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询

六、Key

key列显示MySQL实际决定使用的键（索引），必然包含在possible\_keys中

如果没有选择索引，键是NULL。要想强制MySQL使用或忽视possible\_keys列中的索引，在查询中使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

七、key\_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的）

不损失精确性的情况下，长度越短越好

八、ref

列与索引的比较，表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

九、rows

估算出结果集行数，表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

## 数据仓库

**数据仓库都有什么特点呢**

**1.主题性**

不同于传统数据库对应于某一个或多个项目，数据仓库根据使用者实际需求，将不同数据源的数据在一个较高的抽象层次上做整合，所有数据都围绕某一主题来组织。

这里的主题怎么来理解呢？比如对于滴滴出行，“司机行为分析”就是一个主题，对于链家网，“成交分析”就是一个主题。

**2.集成性**

数据仓库中存储的数据是来源于多个数据源的集成，原始数据来自不同的数据源，存储方式各不相同。要整合成为最终的数据集合，需要从数据源经过一系列抽取、清洗、转换的过程。

**3.稳定性**

数据仓库中保存的数据是一系列历史快照，不允许被修改。用户只能通过分析工具进行查询和分析。

**4.时变性**

数据仓库会定期接收新的集成数据，反应出最新的数据变化。这和特点并不矛盾。

## 分库分表技术演进&最佳实践

**如何存储海量数据？目前比较普遍的方案有3个**

分区；

分库分表；

NoSQL/NewSQL；

不使用NoSQL/NewSQL的原因：不成熟

不使用分区的原因：很多的资源都受到单机的限制，例如连接数，网络吞吐等！虽然每个分区可以独立存储，但是分区表的总入口还是一个MySQL示例。从而导致它的并发能力非常一般，远远达不到互联网高并发的要求！

NoSQL比较具有代表性的是MongoDB，es。NewSQL比较具有代表性的是TiDB。

分库分表中间件

阿里的TDDL，DRDS和cobar，

开源社区的sharding-jdbc（3.x已经更名为sharding-sphere）；

民间组织的MyCAT；

360的Atlas；

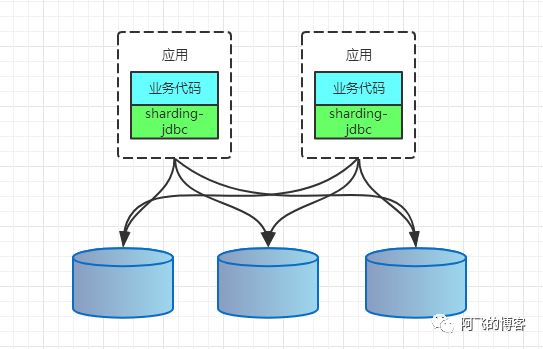
美团的zebra；

**~~分库分表中间件全部可以归结为两大类型**

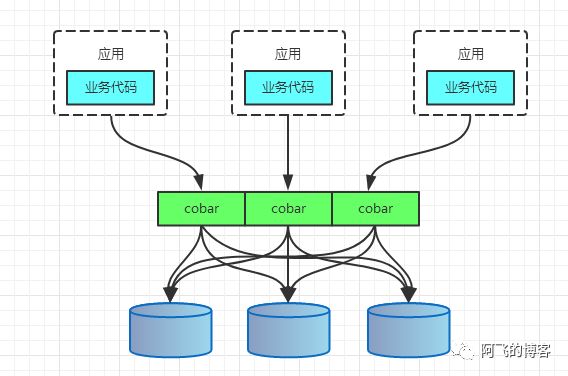
CLIENT模式；

PROXY模式；

CLIENT模式代表有阿里的TDDL，开源社区的sharding-jdbc（sharding-jdbc的3.x版本即sharding-sphere已经支持了proxy模式）。架构如下



PROXY模式代表有阿里的cobar，民间组织的MyCAT。架构如下：



但是，无论是CLIENT模式，还是PROXY模式。几个核心的步骤是一样的：SQL解析，重写，路由，执行，结果归并。

**实战案例**

分库分表第一步也是最重要的一步，即sharding column的选取，sharding column选择的好坏将直接决定整个分库分表方案最终是否成功。选择sharding column的方法最主要分析你的API流量，优先考虑流量大的API，将流量比较大的API对应的SQL提取出来，将这些SQL共同的条件作为sharding column。

这里列举分库分表的几种主要处理思路：

只选取一个sharding column进行分库分表 ；

多个sharding column多个分库分表；

sharding column分库分表 + es(es+HBase)；

## SQL优化考虑的步骤

1.数据库设计和表创建时就要考虑性能，简言之就是使用合适的数据类型,选择合适的索引，表字段避免null值出现，使用枚举或整数代替字符串类型；单表不要有太多字段；

索引：要根据查询有针对性的创建，考虑在WHERE和ORDERBY命令上涉及的列建立索引；应尽量避免在WHERE子句中对字段进行NULL值判断；字符字段只建前缀索引；字符字段最好不要做主键；不用外键，由程序保证约束；简单查询建单列索引，复杂查询建组合索引。使用组合索引时主意顺序和查询条件保持一致，同时删除不必要的单列索引；

2.SQL的编写需要注意优化

SQL优化方面主要应该做到这两点：SQL语句尽可能简单，大语句拆小语句，减少锁时间。查询尽量走索引。

尽量避免在WHERE子句中使用!=或<>操作符，避免模糊查询，避免select \*，OR改写成IN（OR的效率是n级别，IN的效率是log(n)级别，in的个数建议控制在200以内），对于连续数值，使用BETWEEN不用IN。尽量不用函数和触发器，在应用程序实现。使用连接查询（join）来代替子查询，但有些情况下使用可能使用子查询更快些。

还有一些就是条件顺序的优化：from后面的表关联，是自右向左解析的 ，这时就可以将记录最少的表放在最右边。而where条件的解析顺序是自下而上的。那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾,还有就是表之间的连接条件必须写在其它Where条件之前。

3.分区

如果做到上面点都无优化效果，可以考虑引入分区，MySQL在5.1版引入的分区是一种简单的水平拆分，用户需要在建表的时候加上分区参数，MySQL实现分区的方式也意味着索引也是按照分区的子表定义，没有全局索引。

用户的SQL语句是需要针对分区表做优化，SQL条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，可以通过EXPLAINPARTITIONS来查看某条SQL语句会落在那些分区上，从而进行SQL优化

分区的好处：

可以让单表存储更多的数据

分区表的数据更容易维护，可以通过清楚整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化、检查、修复等操作

部分查询能够从查询条件确定只落在少数分区上，速度会很快

分区表的数据还可以分布在不同的物理设备上，从而搞笑利用多个硬件设备

可以使用分区表赖避免某些特殊瓶颈，例如InnoDB单个索引的互斥访问、ext3文件系统的inode锁竞争

可以备份和恢复单个分区

分区的限制和缺点

一个表最多只能有1024个分区

如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来

分区表无法使用外键约束

NULL值会使分区过滤无效

所有分区必须使用相同的存储引擎

4.分表

分表分为垂直拆分和水平拆分，通常以某个字段做拆分项。

5.分库

把一个数据库分成多个，建议做个读写分离就行了，真正的做分库也会带来大量的开发成本

## 其它

**关系型数据库的优点**

**直观**：就是个表格；

**使用简单**：一般都只需要简单的SQL便可查出；

**对数据完整性的支持很好**；

**支持事务**；

**范式**：想要使用我们关系型数据库，必须得遵守一定的规则，这些规则就是“范式”。

第一范式是基本要求，即每个列都是不分割的数据项， 如果连这个都满足不了，还是洗洗睡吧。

第二范式要求实体属性要完全依赖主键，不能依赖部分主键。

第三范式就是一个表中不能包含其它表中已包含的非主关键字信息。不严谨地说就是这个表只包含其他表的ID。

做“数据的集成”

存在问题：高并发，大流量存在瓶颈。对分布式系统支持不好, 难于组成集群。

Java的常用数据库对象包括？

DatabaseMetaData ResultSetMetaData

触发器（Trigger） [ˈtrɪgə(r)]、表（Table）、视图（View）、存储过程（StoredProcedure）、索引（Index）、缺省值（Default）、图表（Diagram）、用户（User）、规则（Rule）等几类。

**sql的行转列(PIVOT)与列转行(UNPIVOT)**

**什么是ETL？**

ETL的英文全称是 Extract| ɪkˈstrækt |-Transform-Load 的缩写，用来描述将数据从来源迁移到目标的几个过程：

1.Extract，数据抽取，也就是把数据从数据源读出来。

2.Transform，数据转换，把原始数据转换成期望的格式和维度。如果用在数据仓库的场景下，Transform也包含数据清洗，清洗掉噪音数据。

3.Load 数据加载，把处理后的数据加载到目标处，比如数据仓库。

Hive是基于Hadoop的数据仓库工具，可以对存储在HDFS上的文件数据集进行查询和分析处理。

**SQL解析，SQL执行过程**

客户端把语句发给服务器端执行，服务器会先在数据库的高速缓存中去查找,是否存在相同语句的执行计划。如果在数据高速缓存中,则服务器进程就会直接执行这个 SQL 语句,省去后续的工作。如果没有就会走下一步，进行语句合法性检查(data dict cache)，语言含义检查(data dict cache)，当语法、语义都正确后,系统就会对我们需要查询的对象加锁。并且检查所连接的用户是否有这个数据访问的权限。当语句与语法都没有问题,权限也匹配的话,服务器进程还是不会直接对数据库文件进行查询。服务器进程会根据一定的规则,对这条语句进行优化。然后再是语义执行，提取数据并返回。

简要流程：查缓存->语句合法性检查->给查询对象加锁->权限检查->sql语句优化->提取数据

**说一下关系型数据库和非关系型数据库的区别**

非关系型数据库的优势：

性能：NOSQL是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过SQL层的解析，所以性能非常高

可扩展性：同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

使用场景：日志、埋点、论坛、博客等

关系型数据库的优势：

复杂查询：可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询

事务支持:使得对于安全性能很高的数据访问要求得以实现。

使用场景：所有有逻辑关系的数据存储

**删除重复记录**

1.通过建立临时表来实现

SQL>create table temp\_emp as (select distinct \* from employee)

SQL> truncate table employee; (清空employee表的数据）

SQL> insert into employee select \* from temp\_emp; (再将临时表里的内容插回来）

2.通过唯一rowid实现删除重复记录.在Oracle中，每一条记录都有一个rowid，rowid在整个数据库中是唯一的，rowid确定了每条记录是在Oracle中的哪一个数据文件、块、行上。在重复的记录中，可能所有列的内容都相同，但rowid不会相同，所以只要确定出重复记录中那些具有最大或最小rowid的就可以了，其余全部删除。

SQL>delete from employee e2 where rowid not in (

select max(e1.rowid) from employee e1 where

e1.emp\_id=e2.emp\_id and e1.emp\_name=e2.emp\_name and e1.salary=e2.salary);--这里用min(rowid)也可以。

SQL>delete from employee e2 where rowid <(

select max(e1.rowid) from employee e1 where

e1.emp\_id=e2.emp\_id and e1.emp\_name=e2.emp\_name and

e1.salary=e2.salary);

3.也是通过rowid，但效率更高。

SQL>delete from employee where rowid not in (

select max(t1.rowid) from employee t1 group by

t1.emp\_id,t1.emp\_name,t1.salary);--这里用min(rowid)也可以。

EMP\_ID EMP\_NAME SALARY

mysql删除重复数据

DELETE FROM USER WHERE id NOT IN (SELECT a.id FROM (SELECT MAX(id) id FROM USER GROUP BY age) a)

# Mybatis

## MyBatis编程步骤是什么样的

① 创建SqlSessionFactory

② 通过SqlSessionFactory创建SqlSession

③ 通过sqlsession执行数据库操作

④ 调用session.commit()提交事务

⑤ 调用session.close()关闭会话

**Mybatis工作原理？**

通过SqlSessionFactoryBuilder从mybatis-config.xml配置文件中构建出SqlSessionFactory。

SqlSessionFactory开启一个SqlSession，通过SqlSession实例获得Mapper对象并且运行Mapper映射的Sql语句。

完成数据库的CRUD操作和事务提交，关闭SqlSession。

## Mybatis中的dao和xml关联关系

**什么是Mybatis的接口绑定，有什么好处？**

Mybatis实现了DAO接口与xml映射文件的绑定，自动为我们生成接口的具体实现，使用起来变得更加省事和方便。

**mybatis 的 dao 接口跟 xml 文件里面的sql 是如何建立关系的？**

Mybatis在初始化SqlSessionFactoryBean的时候，找到mapperLocations路径去解析里面所有的XML文件。Mybatis会把每个SQL标签封装成SqlSource对象（根据SQL语句的不同，又分为动态SQL和静态SQL。其中，静态SQL包含一段String类型的sql语句；而动态SQL则是由一个个SqlNode组成）。然后会将SqlSource与全限定类名+方法名组成的ID封装在一个MappedStatement对象中，并将它缓存到Configuration#mappedStatements中，每一个SQL标签就对应一个MappedStatement对象。

当使用@Autowired注入这个Dao接口的时候，spring会通过JDK动态代理，返回了一个Dao接口的代理对象，这个代理对象的处理器是MapperProxy对象。调用Dao接口方法的时候，实际调用到代理对象的invoke方法，在这里，实际上调用的就是SqlSession里面的东西了。SqlSession通过statement全限定类型+方法名拿到MappedStatement 对象，然后

MappedStatement调用执行器Executor去具体SQL并返回。

**对于有实现的dao接口，mapper还会用代理么？**

答案是肯定，只要你配置了MapperScan，它就会去扫描，然后生成代理。但是，如果你的dao接口有实现类，并且这个实现类也是一个Spring Bean，那就要看你在Autowired的时候，去注入哪一个了。如果没有显示注入哪个实现类，这种情况会在启动时就报错。

**如果有两个XML文件和这个Dao建立关系，岂不是冲突了？**

不管有几个XML和Dao建立关系，只要保证namespace+id唯一即可。

**使用MyBatis的mapper接口调用时有哪些要求？**

① Mapper接口方法名和mapper.xml中定义的每个sql的id相同

② Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同

③ Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

④ Mapper.xml文件中的namespace即是mapper接口的类路径。

**SqlMapConfig.xml中配置有哪些内容？**

SqlMapConfig.xml中配置的内容和顺序如下：

properties（属性）

settings（配置）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

mappers（映射器）

## 简单的说一下MyBatis的一级缓存和二级缓存？

Mybatic的缓存模块使用了装饰者模式的变体

Mybatis的一级、二级缓存: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。

## Hibernate与MyBatis对比

相同点：

Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启执行事务和SQL语句。其中SessionFactoryBuider，SessionFactory，Session的生命周期都是差不多的。Hibernate和MyBatis都支持JDBC和JTA事务处理。

不同点

（1）hibernate完全可以通过对象关系模型实现对数据库的操作，，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。而mybatis仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写sql来实现和管理。

（2）hibernate数据库移植性远大于mybatis

（3）hibernate拥有完整的日志系统，mybatis则欠缺一些

hibernate日志系统非常健全，涉及广泛，包括：sql记录、关系异常、优化警告、缓存提示、脏数据警告等；而mybatis则除了基本记录功能外，功能薄弱很多。

（4）mybatis相比hibernate需要关心很多细节，因为mybatis很多东西都要自己实现。

（5）sql直接优化上，mybatis要比hibernate方便很多

（6）缓存机制上，hibernate要比mybatis更好一些

MyBatis的二级缓存配置都是在每个具体的表-对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓存机制。并且Mybatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。

而Hibernate对查询对象有着良好的管理机制，用户无需关心SQL。所以在使用二级缓存时如果出现脏数据，系统会报出错误并提示。

## MyBatis框架的优/缺点：

Mybaits的优点：

（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50%以上的代码量，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）能够与Spring很好的集成；

（5）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。

Mybaits的缺点：

（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。

**MyBatis框架适用场合：**

（1）MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。

（2）对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao接口即Mapper接口。接口的全限名，就是映射文件中的namespace的值；接口的方法名，就是映射文件中Mapper的Statement的id值；接口方法内的参数，就是传递给sql的参数。

Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MapperStatement。在Mybatis中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个MapperStatement对象。

举例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到namespace为com.mybatis3.mappers.StudentDao下面 id 为 findStudentById 的 MapperStatement。

Mapper接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。Mapper 接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Mapper接口生成代理对象proxy，代理对象会拦截接口方法，转而执行MapperStatement所代表的sql，然后将sql执行结果返回。

## 其它

**$ 跟 # 的使用场景 ？**

#{}是预编译处理，${}是字符串替换。

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

**Mybatis在处理${}时，就是把${}替换成变量的值。**

使用#{}可以有效的防止SQL注入，提高系统安全性。

$一般用于传递变量（如传递数据库名称，和order by后面的字段）。#一般用于传值。

**什么情况用注解，什么情况用xml绑定？**

注解使用情况：Sql语句简单时

xml绑定使用情况：xml绑定 (@RequestMap用来绑定xml文件)

**Mybatis在核心处理类叫什么?**

SqlSession

**JDBC编程有哪些不足之处，MyBatis是如何解决这些问题的？**

① 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

② Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

③ 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决： Mybatis自动将java对象映射至sql语句。

④ 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象。

# Hibernate

## Hibernate的核心类和接口

Configuration ，SessionFactory，Session，Transaction，Query

Hibernate pojo的3种状态：瞬态 持久化 托管（离线）

一个PO有三种状态：

1、未被持久化的vO==pojo:此时就是一个内存对象VO，由JVM管理生命周期

2、已被持久化的PO，并且在Session生命周期内：此时映射数据库数据，由数据库管理生命周期

3、曾被持久化过，但现在和Session已经detached了，以VO的身份在运行这种和Session已经detached的PO还能够进入另一个Session，继续进行PO状态管理，此时它就成为PO的第二种状态了。这种PO实际上是跨了Session进行了状态维护的。在传统的JDO1.x中，PO只有前面两种状态，一个PO一旦脱离PM，就丧失了状态了，不再和数据库数据关联，成为一个纯粹的内存VO，它即使进入一个新的PM，也不能恢复它的状态了

## Hibernate的缓存分为

一级缓存(内置缓存)->session级别缓存,是Hibernate 的内置缓存 一级缓存在Hibernate中对应的即为session范围的缓存，也就是当session关闭时缓存即被清除，一级缓存在Hibernate中是不可配置的部分

二级缓存(应用级缓存)：

二级缓存也称进程级的缓存或SessionFactory级的缓存，二级缓存可以被所有的session共享

## Hibernate 延迟加载(lazy)

1、lazy的概念，指在需要数据的时候才发出sql

2、lazy策略只是在session打开期间才是有效的

hibernate 抓取策略 ( 查询抓取 子查询抓取 连接抓取 批量抓取)

连接抓取 fetch="join" 通过select语句使用外连接来加载其关联实体或集合 此时lazy会失效

查询抓取 fetch="select" 另外发送一条select语句抓取当前对象关联实体或集合

子查询抓取 fetch="subselect" 另外发送一条select语句抓取在前面查询到的所有实对象的关联集合 通过子查询in完成

批量抓取 batch-size设置

## Hibernate使用乐观锁

：只需在在class标签上 optimistic-lock="version"，等配置。

悲观锁的实现，通常依赖于数据库机制，在整个过程中将数据锁定，其它任何用户都不能读取或修改 利用LockMode.UPGRADE。p=(Person)session.load(Person.class, 6,LockMode.UPGRADE);

# Redis

## Redis数据类型以及底层数据结构：

**Redis有哪些数据类型？**

字符串String、字典Hash、列表List、集合Set、有序集合ZSet（SortedSet）。redis5.0新增了stream数据结构

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

**String：**最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

**hash:**这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。

**List：**使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。

**Set:**所以可以做全局去重的功能。就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

**sorted set：**sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作，可以用来做延时任务，可以做范围查找。

**Stream**:如果你了解MQ，那么可以把streams当做MQ。如果你还了解kafka，那么甚至可以把streams当做kafka另外，这个功能有点类似于redis以前的Pub/Sub，但是也有基本的不同。streams支持多个客户端（消费者）等待数据（Linux环境开多个窗口执行XREAD即可模拟），并且每个客户端得到的是完全相同的数据。Pub/Sub是发送忘记的方式，并且不存储任何数据；而streams模式下，所有消息被无限期追加在streams中，除非用于显示执行删除（XDEL）。streams的Consumer Groups也是Pub/Sub无法实现的控制方式。

**HyperLogLog** 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%，这样的精确度已经可以满足上面的 UV 统计需求了。

**Redis**可以执行发布/订阅模式(publish/subscribe), 该模式可以解耦消息的发送者和接收者,使程序具有更好的扩展性.

**Redis底层数据结构**

**简单动态字符串（simple dynamic string）SDS；链表；字典；跳跃表；整数集合；压缩列表；对象**

**字典**，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对的抽象数据结构。在字典中，一个键（key）可以和一个值（value）进行关联，字典中的每个键都是独一无二的。比如SET msg "hello world"创建这样的键值对（“msg”，“hello world”）在数据库中就是以字典的形式存储。Redis 字典所使用的哈希表由 dict.h/dictht 结构定义.Redis 中采用了连地址法（separate chaining）来解决键冲突。

**跳跃表**：跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

**整数集合**：整数集合是集合建的底层实现之一，当一个集合中只包含整数，且这个集合中的元素数量不多时，redis就会使用整数集合intset作为集合的底层实现。我们可以这样理解整数集合，他其实就是一个特殊的集合，里面存储的数据只能够是整数，并且数据量不能过大。

**压缩列表**：压缩列表是列表键和哈希键的底层实现之一。当一个列表键只把汗少量列表项，并且每个列表项要么就是小整数，要么就是长度比较短的字符串，那么Redis 就会使用压缩列表来做列表键的底层实现。

**知道动态字符串sds的优缺点么？**

**SDS与C字符串区别**

**C字符串：**

**缺点：**

获取字符串长度的复杂度为O(N)

API不安全，可能会造成缓存区溢出（对于一个字符串A后面拼接字符串B时，需要考虑A字符串的容量问题）

修改字符串长度N次必然需要执行N次内存重分配（即无论是缩小还是扩大C字符串的大小都需要进行内存分配）

只能保存文本数据（即C字符串是以'\0'结尾的，如果想保存'\0'这个特殊字符，则会导致数组提前被截取了）

**优点：**

可以使用所有<String.h>库中的函数

**SDS**

**优点：**

获取字符串长度的复杂度为O(1)

API是安全的，不会造成缓存溢出问题

修改字符串长度N次最多需要执行N次内存重新分配（由于SDS结构中有free字段的存在，因此缩小时，会把没有的char算到free中（惰性空间释放），而放大（空间预分配）时如果len长度<1M，则扩大后的len=原有len\*2，如果len长度>=1M，则扩大后的len=原有len+1M）

可以保存文本或者二进制数据（因为SDS获取字符串时是根据len长度算的，而不是根据'\0'结尾之前的字符串算）

可以使用一部分<String.h>库中函数（不需要重新实现）

**缺点：**

很多函数以值的形式返回新字符串，由于有时SDS要求创建一个占用更多空间的新字符串，所以大多数SDS的API调用像这样：s = sdscat(s,"Some more data");你可以看到s被用来作为sdscat的输入，但也被设为SDS API调用返回的值，因为我们不知道此调用是否会改变了我们传递的SDS字符串，还是会重新分配一个新的字符串。忘记将sdscat或者类似函数的返回值赋回到存有SDS字符串的变量的话，就会引起bug。

如果一个SDS字符串在你的程序中多个地方共享，当你修改字符串的时候，你必须修改所有的引用。但是，大多数时候，当你需要共享SDS字符串时，将字符串封装成一个结构体，并使用一个引用计数会更好，否则很容易导致内存泄露。

## Redis实现原理及作用

redis是一个key-value存储系统。它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。

**单线程的redis为什么这么快**

(一)纯内存操作

(二)单线程操作，避免了频繁的上下文切换

(三)采用了非阻塞I/O多路复用机制

## Redis分布式锁实现

使用set（key，value，30，NX）： 30秒过期，NX表示当这个key的值不存在时才存，与其相反的时XX，只有key存在时才存。

这种情况还要考虑锁超时的情况：

锁超时的情况下，另一个线程在删除这个Key的时候要对比一下是不是自己设置的key,不然会导致误删，因此，value要有唯一性。可以考虑value的格式为：唯一值+时间戳

假如某线程成功得到了锁，并且设置的超时时间是30秒。如果某些原因导致线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。随后，线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，线程A实际上删除的是线程B加的锁。可以将当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性

我们都是追求极致的程序员，所以这一块要用Lua脚本来实现：

String luaScript = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

redisClient.eval(luaScript , Collections.singletonList(key), Collections.singletonList(threadId));

3. 执行时间大于超时时间的问题，可以开启一个守护线程，在这个线程没执行完再续20秒

## redis的过期策略以及内存淘汰机制

redis采用的是定期删除+惰性删除策略。

定期删除，即默认每隔100ms随机抽取进行检查,是否有过期的key,有过期key则删除。(如果每隔100ms,全部key进行检查，redis岂不是卡死)。但是这种策略会导致很多key到时间没有删除。

于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。

在redis.conf中有一行配置

# maxmemory-policy volatile-lru

**该配置就是配内存淘汰策略的**

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。

2）allkeys-lru(Least recently used)：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。推荐使用，目前项目在用这种。

3）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。应该也没人用吧，你不删最少使用Key,去随机删。

4）volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。这种情况一般是把redis既当缓存，又做持久化存储的时候才用。

5）volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。

6）volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

ps：如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

**为什么不用定时删除策略?**

定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略.

## 如何解决redis的并发竞争key问题

回答:如下所示

(1)如果对这个key操作，不要求顺序

这种情况下，准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可，比较简单。

(2)如果对这个key操作，要求顺序

其他方法，比如利用队列，将set方法变成串行访问也可以。

Redis的并发竞争问题，主要是发生在并发写竞争。

## redis常见性能问题和解决方案

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

## 缓存问题

**缓存穿透**

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

解决方案:

(一)利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试

(二)采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

(三)提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器BloomFilter 类似于一个hase set 用来判断某个元素（key）是否存在于某个集合中），内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

(四)缓存一个空值，设置比较短的有效时间。

**缓存击穿**

在平常高并发的系统中，大量的请求同时查询一个 key 时，此时 这个key 正好失效了，就会导致大量的请求都打到数据库上面去。这种现象我们成为击穿。

如何解决：上面的现象是多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它。其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后做缓存。后面的线程进来发现已经有缓存了，就直接走缓存。

**缓存雪崩**

缓存的情况是说，当某一时刻发生大规模的缓存失效的情况。比如你的缓存服务宕机了，会有大量的请求进来直接打到DB上面。结果就是DB 称不住，挂掉。

解决办法：事前：使用集群缓存，保证缓存服务的高可用（如果是使用 Redis，可以使用 主从+哨兵 ，Redis Cluster 来避免 Redis 全盘崩溃的情况）事中：使用 ehcache 本地缓存 + Hystrix 限流&降级 ,避免 MySQL 被打死的情况发生。事后：开启 Redis 持久化机制，尽快恢复缓存集群

**热点数据集中失效问题怎么解决**

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案:

(一)给缓存的失效时间，加上一个随机值，避免集体失效。

(二)使用互斥锁，（在第一个请求去查询数据库的时候对他加一个互斥锁，其余的查询请求都会被阻塞住，直到锁被释放，从而保护数据库。）

(三)双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点

I 从缓存A读数据库，有则直接返回

II A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程。

III 更新线程同时更新缓存A和缓存B。

另一种解决方式是设置永不过期的数据，即通过新起一个线程的方式去定时将数据库中的数据更新到缓存中

**redis的热key问题**

热key，就是瞬间有几十万的请求去访问redis上某个固定的key，从而压垮缓存服务的情情况。

hot key和big key问题，大家一定要有所了解。

**怎么发现热key**

方法一:凭借业务经验，进行预估哪些是热key

方法二:在客户端进行收集

方法三:在Proxy层做收集

方法四:用redis自带命令

(1)monitor命令，该命令可以实时抓取出redis服务器接收到的命令，然后写代码统计出热key是啥。当然，也有现成的分析工具可以给你使用，比如redis-faina。但是该命令在高并发的条件下，有内存增暴增的隐患，还会降低redis的性能。

(2)hotkeys参数，redis 4.0.3提供了redis-cli的热点key发现功能，执行redis-cli时加上–hotkeys选项即可。但是该参数在执行的时候，如果key比较多，执行起来比较慢。

方法五:自己抓包评估

**如何解决**

(1)利用二级缓存:比如利用ehcache，或者一个HashMap都可以。在你发现热key以后，把热key加载到系统的JVM中。

(2)备份热key:不要让key走到同一台redis上不就行了。我们把这个key，在多个redis上都存一份不就好了。接下来，有热key请求进来的时候，我们就在有备份的redis上随机选取一台，进行访问取值，返回数据。

**业内方案**

在客户端进行收集热key：可以使用TMC方案

TMC 对原生jedis包的JedisPool和Jedis类做了改造，在JedisPool初始化过程中集成TMC“热点发现”+“本地缓存”功能Hermes-SDK包的初始化逻辑。也就说人家改写了jedis原生的jar包，加入了Hermes-SDK包

那Hermes-SDK包用来干嘛？

OK，就是做热点发现和本地缓存。从监控的角度看，该包对于Jedis-Client的每次key值访问请求，Hermes-SDK 都会通过其通信模块将key访问事件异步上报给Hermes服务端集群，以便其根据上报数据进行“热点探测”。

**分布式缓存系统面临的问题：**

缓存一致性问题

1：缓存系统与底层数据的一致性。这点在底层系统是“可读可写”时，写得尤为重要

2：有继承关系的缓存之间的一致性。为了尽量提高缓存命中率，缓存也是分层：全局缓存，二级缓存。他们是存在继承关系的。全局缓存可以有二级缓存来组成。

3：多个缓存副本之间的一致性。为了保证系统的高可用性，缓存系统背后往往会接两套存储系统（如memcache，redis等）

**如何保证缓存与数据库的双写一致性？**

分析:一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。答这个问题，先明白一个前提。就是如果对数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。另外，我们所做的方案其实从根本上来说，只能说降低不一致发生的概率，无法完全避免。因此，有强一致性要求的数据，不能放缓存。

首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统不是严格要求 “缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：读请求和写请求串行化，串到一个内存队列里去。

串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

**比较复杂的数据不一致问题分析**

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样了...

解决思路：将操作都发送到串行化队列中执行。但是这个队列就是性能的瓶颈，这时就需要以某个字段为key，散列到分布式队列中进行处理。然而还有一个隐藏的问题：就是有些词访问频繁，有些不平凡，这样也可能会造成某个队列负载过大。

**Redis分布式锁最牛逼的实现**

普通实现：Redis分布式锁大部分人都会想到：setnx+lua，或者知道set key value px milliseconds nx。

这种实现方式有3大要点（也是面试概率非常高的地方）：

set命令要用set key value px milliseconds nx；

value要具有唯一性；

释放锁时要验证value值，不能误解锁；

事实上这类琐最大的缺点就是它加锁时只作用在一个Redis节点上，如果这个master节点由于某些原因发生了主从切换，那么就会出现锁丢失的情况：

在Redis的master节点上拿到了锁；

但是这个加锁的key还没有同步到slave节点；

master故障，发生故障转移，slave节点升级为master节点；

导致锁丢失。

正因为如此，Redis作者antirez基于分布式环境下提出了一种更高级的分布式锁的实现方式：Redlock。其基本思想是：假设有N个Redis master。这些节点完全互相独立，不存在主从复制或者其他集群协调机制。那么在这N个master都写上锁。

RLock lock1 = redissonClient1.getLock(resourceName);

RLock lock2 = redissonClient2.getLock(resourceName);

RLock lock3 = redissonClient3.getLock(resourceName);

RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3);

boolean isLock;

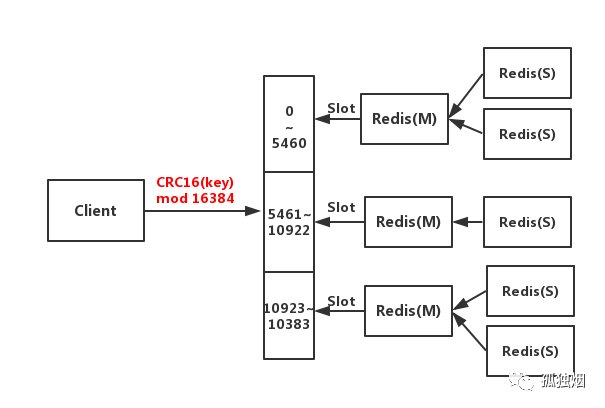
isLock = redLock.tryLock(500, 30000, TimeUnit.MILLISECONDS);

该实现类在org.redisson包下version 3.5

## ？Redis跳跃表的问题？

## ？redis里面的哈希表？

## Redis Cluster



工作原理如下

客户端与Redis节点直连,不需要中间Proxy层，直接连接任意一个Master节点

根据公式HASH\_SLOT=CRC16(key) mod 16384，计算出映射到哪个分片上，然后Redis会去相应的节点进行操作

具有如下优点:

(1)无需Sentinel哨兵监控，如果Master挂了，Redis Cluster内部自动将Slave切换Master

(2)可以进行水平扩容

(3)支持自动化迁移，当出现某个Slave宕机了，那么就只有Master了，这时候的高可用性就无法很好的保证了，万一Master也宕机了，咋办呢？ 针对这种情况，如果说其他Master有多余的Slave ，集群自动把多余的Slave迁移到没有Slave的Master 中。

缺点:

(1)批量操作是个坑

(2)资源隔离性较差，容易出现相互影响的情况。

**Redis Cluster面试问题**

**问题1:懂Redis事务么？**

正常版：Redis事务是一些列redis命令的集合, 先以 MULTI 开始一个事务，然后将多个命令入队到事务中，最后由 EXEC 命令触发事务，一并执行事务中的所有命令。

高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key是有可能分配在不同的Redis节点上的，在这种情况下Redis的事务机制是不生效的。其次，Redis事务不支持回滚操作，简直是鸡肋！所以基本不用！

**问题2:Redis的多数据库机制，了解多少？**

正常版：Redis支持多个数据库，并且每个数据库的数据是隔离的不能共享，单机下的redis可以支持16个数据库（db0 ~ db15）

高调版: 在Redis Cluster集群架构下只有一个数据库空间，即db0。因此，我们没有使用Redis的多数据库功能！

**问题3:Redis集群机制中，你觉得有什么不足的地方吗？**

正常版: 不知道

高调版: 假设我有一个key，对应的value是Hash类型的。如果Hash对象非常大，是不支持映射到不同节点的！只能映射到集群中的一个节点上！还有就是做批量操作比较麻烦！

**问题4:懂Redis的批量操作么？**

正常版: 懂一点。比如mset、mget操作等，blabla

高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key会划分到不同的slot中，因此直接使用mset或者mget等操作是行不通的。

Redis集群是没法执行批量操作命令的，如mget，pipeline等。这是因为redis将集群划分为16383个哈希槽，不同的key会划分到不同的槽中。但是，Jedis客户端提供了计算key的slot方法，已经slot和节点之间的映射关系，通过这两个数据，就可以计算出每个key所在的节点，然后使用pipeline获取数据。

**问题5:那在Redis集群模式下，如何进行批量操作？**

正常版:不知道

高调版:这个问题其实可以写一篇文章了，改天写。这里说一种有一个很简单的答法，足够面试用。即:

如果执行的key数量比较少，就不用mget了，就用串行get操作。如果真的需要执行的key很多，就使用Hashtag保证这些key映射到同一台Redis节点上。简单来说语法如下

对于key为{foo}.student1、{foo}.student2，{foo}student3，这类key一定是在同一个redis节点上。因为key中“{}”之间的字符串就是当前key的hash tags， 只有key中{ }中的部分才被用来做hash，因此计算出来的redis节点一定是同一个!

ps:如果你用的是Proxy分片集群架构，例如Codis这种，会将mget/mset的多个key拆分成多个命令发往不同得Redis实例，这里不多说。我推荐答的还是Redis Cluster。

**问题6:你们有对Redis做读写分离么？**

正常版:没有做，至于原因额。。。额。。。额。。没办法了，硬着头皮扯~

高调版:不做读写分离。我们用的是Redis Cluster的架构，是属于分片集群的架构。而Redis本身在内存上操作，不会涉及IO吞吐，即使读写分离也不会提升太多性能，Redis在生产上的主要问题是考虑容量，单机最多10-20G，key太多降低Redis性能.因此采用分片集群结构，已经能保证了我们的性能。其次，用上了读写分离后，还要考虑主从一致性，主从延迟等问题，徒增业务复杂度。

## Redis妙用

redis是键值对的数据库，常用的五种数据类型为字符串类型（string），散列类型（hash），列表类型（list），集合类型（set），有序集合类型（zset）

Redis用作缓存，主要两个用途：高性能，高并发，因为内存天然支持高并发

应用场景

**分布式锁（string）**

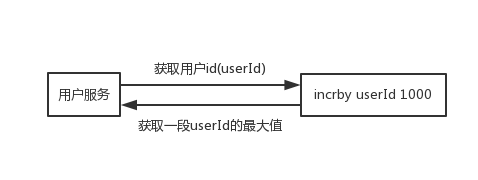
setnx key value，当key不存在时，将 key 的值设为 value ，返回1。若给定的 key 已经存在，则setnx不做任何动作，返回0。

当setnx返回1时，表示获取锁，做完操作以后del key，表示释放锁，如果setnx返回0表示获取锁失败，整体思路大概就是这样，细节还是比较多的，有时间单开一篇来讲解

**计数器（string）**

如知乎每个问题的被浏览器次数

**分布式全局唯一id（string）**



直接获取一段userId的最大值，缓存到本地慢慢累加，快到了userId的最大值时，再去获取一段，一个用户服务宕机了，也顶多一小段userId没有用到

**消息队列（list）**

在list里面一边进，一边出即可

**抽奖活动（set）**

# 参加抽奖活动:

sadd key {userId}

# 获取所有抽奖用户，大轮盘转起来:

smembers key

# 抽取count名中奖者，并从抽奖活动中移除:

spop key count

# 抽取count名中奖者，不从抽奖活动中移除:

srandmember key count

**实现点赞，签到，like等功能(set)**

# 1001用户给8001帖子点赞:

sadd like::8001 1001

# 取消点赞:

srem like::8001 1001

# 检查用户是否点过赞:

sismember like::8001 1001

# 获取点赞的用户列表:

smembers like::8001

# 获取点赞用户数:

scard like::8001

**实现关注模型，可能认识的人（set）**

seven关注的人

sevenSub -> {qing, mic, james}

青山关注的人

qingSub->{seven,jack,mic,james}

Mic关注的人

MicSub->{seven,james,qing,jack,tom}

# 返回sevenSub和qingSub的交集，即seven和青山的共同关注

sinter sevenSub qingSub -> {mic,james}

# 我关注的人也关注他,下面例子中我是seven

# qing在micSub中返回1，否则返回0

sismember micSub qing

sismember jamesSub qing

# 我可能认识的人,下面例子中我是seven

# 求qingSub和sevenSub的差集，并存在sevenMayKnow集合中

sdiffstore sevenMayKnow qingSub sevenSub -> {seven,jack}

**电商商品筛选（set）**

每个商品入库的时候即会建立他的静态标签列表如，品牌，尺寸，处理器，内存

# 将拯救者y700P-001和ThinkPad-T480这两个元素放到集合brand::lenovo

sadd brand::lenovo 拯救者y700P-001 ThinkPad-T480

sadd screenSize::15.6 拯救者y700P-001 机械革命Z2AIR

sadd processor::i7 拯救者y700P-001 机械革命X8TIPlus

# 获取品牌为联想，屏幕尺寸为15.6，并且处理器为i7的电脑品牌(sinter为获取集合的交集)

sinter brand::lenovo screenSize::15.6 processor::i7 -> 拯救者y700P-001

**排行版（zset）**

redis的zset天生是用来做排行榜的、好友列表, 去重, 历史记录等业务需求

# user1的用户分数为 10

zadd ranking 10 user1

zadd ranking 20 user2

取分数最高的3个用户

zrevrange ranking 0 2 withscores

## Redis开放性问题

**1亿个key，找出已知的前缀开头的10w个key**

**假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？**

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

**使用过Redis做异步队列么，你是怎么用的？**

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

如果对方追问pub/sub有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

如果对方追问redis如何实现延时队列？使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理

**如果有大量的key需要设置同一时间过期，一般需要注意什么？**

如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

## 其它

happen-before原则（前一个操作的执行结果必须对后一个操作可见）

**redis的单线程特性有什么优缺点？**

**Redis快的主要原因是**：

完全基于内存

数据结构简单，对数据操作也简单

使用多路 I/O 复用模型：这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。

**单进程单线程好处**

代码更清晰，处理逻辑更简单

不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

不存在多进程或者多线程导致的切换而消耗CPU

**单进程单线程弊端**

无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；

不适合大数据传输

**其他一些优秀的开源软件采用的模型**

多进程单线程模型：Nginx

单进程多线程模型：Memcached

**Redis分布式锁操作的原子性，Redis内部是如何实现的？**

因为redis是单线程的。

**Pipeline有什么好处，为什么要用pipeline？**

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

**Redis的同步机制了解么？**

Redis可以使用主从同步，从从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

**是否使用过Redis集群，集群的原理是什么？**

Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

**使用redis有什么缺点**

分析:大家用redis这么久，这个问题是必须要了解的，基本上使用redis都会碰到一些问题，常见的也就几个。

回答:主要是四个问题

(一)缓存和数据库双写一致性问题

(二)缓存雪崩问题

(三)缓存击穿问题

(四)缓存的并发竞争问题

**Redis的性能瓶颈在哪？**

一般在两个地方：

其一是机器内存大小，内存大小关系到Redis存储的数据量

其二是网络带宽，Redis客户端执行一条命令分为四个过程：发送命令、命令排队、命令执行、返回结果。而其中发送命令+返回结果这一过程被称为Round Trip Time（RTT，往返时间）

**为什么 Redis 单线程却能支撑高并发？**

redis中使用I/O多路复用技术来防止I/O阻塞

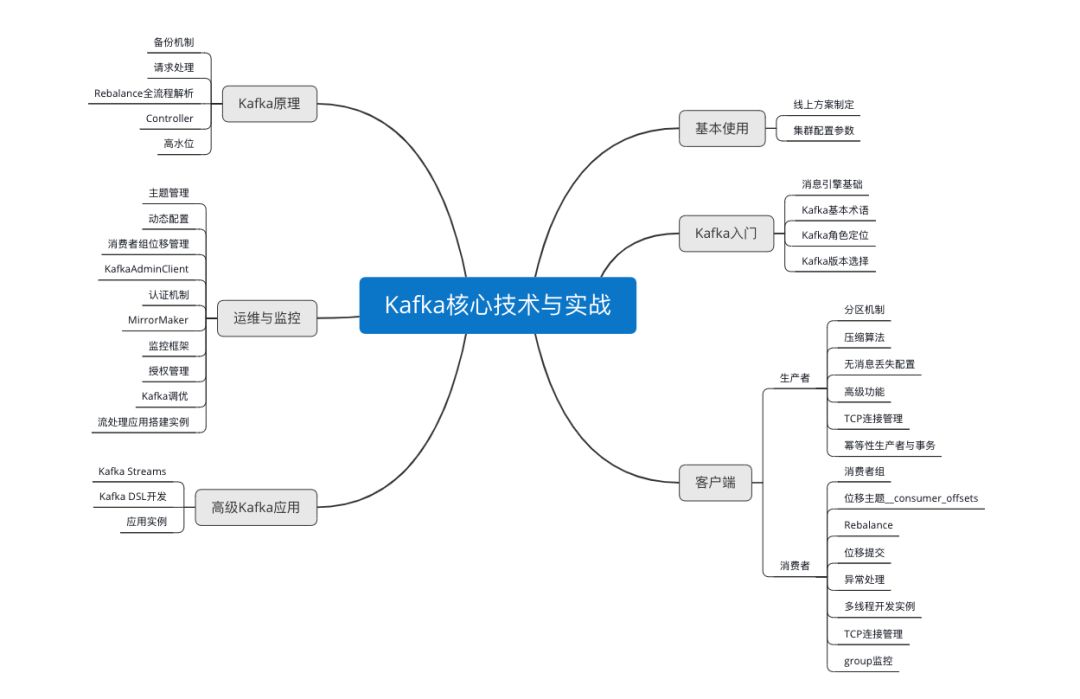
在 I/O 多路复用模型中，最重要的函数调用就是 select，该方法的能够同时监控多个文件描述符的可读可写情况，当其中的某些文件描述符可读或者可写时， select方法就会返回可读以及可写的文件描述符个数。与此同时也有其它的 I/O 多路复用函数 epoll/kqueue/evport

**Redis为什么设计成单线程？**

执行两个任务，在不涉及I/O操作的时候，单线程依次执行两个任务比双线程分别执行两个任务要快。Redis不涉及I/O操作，因此设计为单线程是效率最高的！

# Kafka

Kafka 核心技术：我们仅需要学习一套框架，就能在实际业务系统中实现消息队列应用、应用程序集成、分布式存储构建，甚至是流处理应用的开发与部署



极客时间新上线的《Kafka 核心技术与实战》

crc校验：消息完整性校验

ack：一个确认反馈机制

ISR：leader中记录的与其保持同步的Replica列表。

## kafka 的架构，包含了哪些角色？

**Broker**:一台 kafka 服务器就是一个 broker。一个集群由多个broker组成。

**Producer**:消息生产者。

**Consumer**:消息消费者。

**Topic**：Kafka的悄息通过 Topic 进行分类。 主题就好比数据库的表，或者文件系统里的文件夹。

**Partition**：一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个topic 可以分为多个 partition.消息以追加的方式写入partition，后以先先出的顺序读取。每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的id（offset）。kafka 只保证按一个 partition 中的顺序将消息发给 consumer，不保证一个 topic 的整体（多个partition间）的顺序；

**Consumer Group**（同一个消费者组中的消费者，一个分区只能被一个消费者消费）一个 Consumer Group 是多个 consumer 的组合, 作为一个整体存在.同一个消费者组内的消费者读取消息的时候, 不会读取同一个分区内的消息.但是组与组之间不受任何影响. 组内的所有消费者协调在一起来消费订阅主题(subscribed topics)的所有分区(partition)。当然，每个分区只能由同一个消费组内的一个consumer来消费。

**Offset**：偏移量.kafka 的存储文件都是按照 offset.kafka 来命名用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到2048.kafka 的文件即可。当然the first offset就是00000000000.kafka

**kafka ack 机制？集群中的ack 是怎么实现的？**

Kafka producer有三种ack机制：

0 （意味着producer不等待broker同步完成的确认）

1（意味着producer要等待leader成功收到数据并得到确认）

-1（意味着producer得到follwer确认）

**kafka 的最小工作单元？**

生产者、消费者、主题、偏移量

kafka 消息重复消费的问题？幂等怎么做的？

数据版本号，还有数据库唯一索引

产生重复消费的情况，比如说投递的时候重复了，消费的时候由于 offset 没处理好等等问题导致的话，我想可能会更好。

## Kafka存储策略

1. Kafka以topic来进行消息管理，每个topic包含多个partitions，每个partition对应一个逻辑log，由多个segment组成。

2. 每个segment中存储多条消息，消息id由其逻辑位置决定，该id可直接定位到消息存储位置，避免id到位置的额外映射。

3．每个part(patition)在内存中对应一个index，记录每个segment中的第一条消息偏移offset。

4. 发布者发到某个topic的消息会被均匀的分布到多个part上，broker收到发布消息往对应part的最后一个segment上添加消息。当某个segment上的消息数达到配置或发布时间的阀值，会被flush到磁盘。

5．无论消息是否被消费，只要在配置的时间段内，消息都将被留。

**Kafka数据持久化**

kafka使用文件存储消息(append only log),这就直接决定kafka在性能上严重依赖文件系统的本身特性.且无论任何OS下,对文件系统本身的优化是非常艰难的.文件缓存/直接内存映射等是常用的手段.因为kafka是对日志文件进行append操作,因此磁盘检索的开支是较小的;同时为了减少磁盘写入的次数,broker会将消息暂时buffer起来,当消息的个数(或尺寸)达到一定阀值时,再flush到磁盘,这样减少了磁盘IO调用的次数.对于kafka而言,较高性能的磁盘,将会带来更加直接的性能提升.

kafka是由follower周期性或者尝试去pull(拉)过来(其实这个过程与consumer消费过程非常相似)

**kafka不是完全同步，也不是完全异步，是一种ISR机制**

1. leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，而且是由leader动态维护   
2. 如果一个flower比一个leader落后太多，或者超过一定时间未发起数据复制请求，则leader将其重ISR中移除   
3. 当ISR中所有Replica都向Leader发送ACK时，leader才commit

## Kafka消息丢失怎么处理？

消息丢失的情况

1.在producer发送了kafka未接到的情况：

设置ack=1的时候，表示producer发送出去message，同步的把message存到对应topic的partition的leader上，然后producer就返回成功，partition leader异步的把message同步到其他partition replica上。

2.在kafka发送了consumer未接收或未保存成功的情况：

enable.auto.commit=false  关闭自动提交位移

3.在kafka中丢失：

这种情况丢数据的方式也有很多种，比如：kafka controller 切换的时候，会导致partition leader的切换（老的 kafka controller上面的partition leader会选举到其他的kafka broker上）,但是这样就会导致丢数据。或者kafka上消息积压太多，导致很多消息过期。

使用kafka(0.8.2.1)高级API消费消息时，有时会因各种原因，导致消息堆积。如果请求offset对应消息已过期，则会抛出下面异常：

Current offset 789380 for partition [test,3] out of range; reset offset to 799380"

抛出该异常的同时，会把该topic和分区下次请求的offset重置为一个新的值，此时就发生了消息丢失。

那么，如果减少数据丢失呢？

注意异常后重置offset的值。先看处理offset超出有效范围的代码。

通过代码可以清楚看到，这个重置的offet值和auto.offset.reset配置有关。auto.offset.reset值有earliest和latest。当消费客户端启动时

**earliest** 当分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费。   
**latest** 当分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据。

当消费客户端在消费过程中，如果有消息过期，earliest会将offset重置为kafka里最早消息，latest则会将offset重置为最新消息。因此，把auto.offset.reset设置为earliest， 可以在消费堆积的情况下，减少数据丢失。注意，auto.offset.reset默认是latest。

## Kafka的Leader选举机制？

Kakfa Broker集群受Zookeeper管理。所有的Kafka Broker节点一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。（这个过程叫Controller在ZooKeeper注册Watch）。这个Controller会监听其他的Kafka Broker的所有信息，如果这个kafka broker controller宕机了，在zookeeper上面的那个临时节点就会消失，此时所有的kafka broker又会一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。

## 其它

**Kfaka如何应对大量的长连接**

Kafka采用Reactor多路复用模型来应对大量客户端连接（kafka会与生产者消费者建立大量的长连接，及其耗内存）

简单来说，就是搞一个acceptor线程，基于底层操作系统的支持，实现连接请求监听。

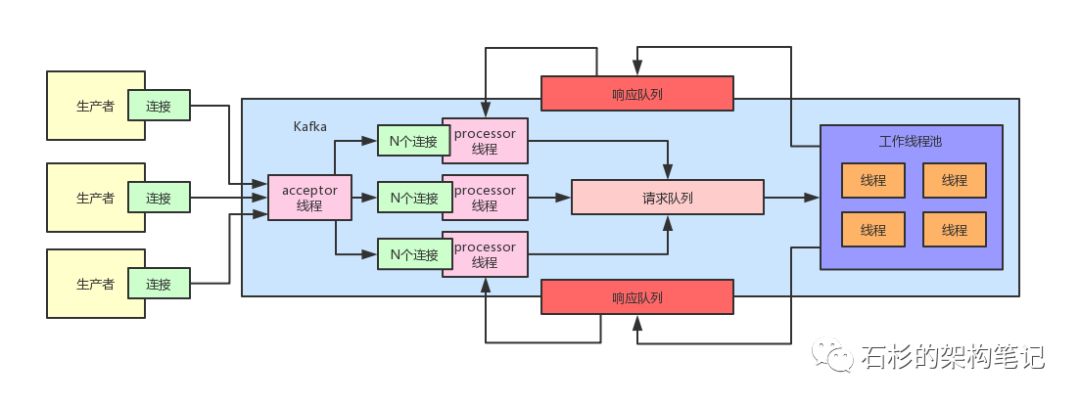
如果有某个设备发送了建立连接的请求过来，那么那个线程就把这个建立好的连接交给processor线程。

每个processor线程会被分配N多个连接，一个线程就可以负责维持N多个连接，他同样会基于底层操作系统的支持监听N多连接的请求。

如果某个连接发送了请求过来，那么这个processor线程就会把请求放到一个请求队列里去。

接着后台有一个线程池，这个线程池里有工作线程，会从请求队列里获取请求，处理请求，接着将请求对应的响应放到每个processor线程对应的一个响应队列里去。

最后，processor线程会把自己的响应队列里的响应发送回给客户端。



**为什么要使用kafka，有那种场景下需要使用jms**

由于来不及同步处理，请求往往会发生堵塞，比如说，大量的insert，update之类的请求同时到达Mysql，直接导致无数的行锁表锁，甚至最后请求会堆积过多，从而触发too many connections错误。通过使用消息队列，我们可以异步处理请求，从而缓解系统的压力。

kafka是一个发布订阅消息系统，由topic区分消息种类，每个topic中可以有多个partition，每个kafka集群有一个多个broker服务器组成，producer可以发布消息到kafka中，consumer可以消费kafka中的数据。kafka就是生产者和消费者中间的一个暂存区，可以保存一段时间的数据保证使用。

**kafka如何保证消息的顺序**

只有一个分区里的消息可以保证顺序，但是同一分区也可能乱序： 而且必须等上一条消息发送完再发送下一条消息。

假设a,b两条消息，a先发送后由于发送失败重试，这时顺序就会在b的消息后面，可以设置max.in.flight.requests.per.connection=1来避免 max.in.flight.requests.per.connection：

限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求，但吞吐量会下降

新启动的consumer默认从partition队列最头端最新的地方开始阻塞的读message

 Kafka producer 发送message不用维护message的offsite信息，因为这个时候，offsite就相当于一个自增id，producer就尽管发送message就好了。

Kafka的producer一般都是大批量的batch发送message

解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息?

在数据中，为了精确地获得Kafka的消息，你必须遵循两件事: 在数据消耗期间避免重复，在数据生产过程中避免重复。

这里有两种方法，可以在数据生成时准确地获得一个语义:

每个分区使用一个单独的写入器，每当你发现一个网络错误，检查该分区中的最后一条消息，以查看您的最后一次写入是否成功

在消息中包含一个主键(UUID或其他)，并在用户中进行反复制

在数据消耗期间避免重复：

offset偏移量

kafka为什么这么优秀(一) https://mp.weixin.qq.com/s/JPKYPazb6zJVwzVnncEdEg

kafka为什么这么优秀(二) https://mp.weixin.qq.com/s/jgd3e\_XnMw2Aluj9olJIFQ

## MQ

**使用MQ优缺点：**

**优点：**

但是比较核心的有 3 个：解耦、异步、削峰。

**缺点：**

增加系统的复杂性，降低系统的可用性。

数据一致性问题，如果多个服务对应多个库，多个服务不太可能永远都保持一致

**你们公司生产环境用的是什么消息中间件？**

Kafka,虽然未参与过选型，不过大概知道为什么。对比于ActiveMQ，RabbitMQ，RocketMQ，Kafka的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数据计算等场景来设计。

**为什么在你们系统架构中要引入消息中间件？**

复杂系统的解耦

复杂链路的异步调用（将调用链路中操作比较耗时的模块抽出来，通过MQ调用）

瞬时高峰的削峰处理

分布式事物

**如何保证消息中间件全链路数据100%不丢失**

第一个会导致数据丢失的地方，就是消费者获取到消息之后，没有来得及处理完毕，自己直接宕机了。这就要调整消费者的ack机制，通过在消费者服务中调整为手动ack机制，来确保消息一定是已经成功处理完了，才会发送ack通知给MQ集群。

第二个是集群接收到消息后整个集群直接宕机了。这个就要确认只要订单服务发送出去的消息确认成功了，此时MQ集群就一定已经将消息持久化到磁盘了。

其次，增加一个额外的系统，发送端成功发送消息给MQ后，同时发送这条消息的id到这个额外系统，服务端处理完这条消息后，将这条消息的状态更新为已处理。

建立死信MQ系统，确保原有MQ系统正常运行。

DLQ-死信队列(Dead Letter Queue)用来保存处理失败或者过期的消息。

RocketMQ对于失败次数超过16次的消息设置为死信消息，消息最终被放到DLQ死信队列中，需要人工进行干预处理。

当一个消息被redelivered超过maximumRedeliveries(缺省为6次，具体设置请参考后面的内容)次数时，会给broker发送一个"Poison ack"，这个消息被认为是a poison pill(毒丸)，这时broker会将这个消息发送到DLQ，以便后续处理。

**说说系统架构引入消息中间件有什么缺点？**

系统可用性降低（MQ挂掉之后的高可用保障方案）

系统稳定性降低（MQ系统故障导致丢数据，多数据等问题）

MQ中间件的这些问题的解决方案，包括：消息高可靠传递（0丢失）；消息幂等性传递（绝对不重复）；百万消息积压的线上故障处理；

分布式一致性问题（C现在处理自己本地数据库成功了，然后发送了一个消息给MQ，系统D也确实是消费到了，但是结果不幸的是，系统D操作自己本地数据库失败了。）

## MQ之间比较

ActiveMQ、RabbitMQ、RocketMQ、Kafka有什么优点和缺点

**ActiveMQ**

单机吞吐量：万级

topic数量都吞吐量的影响：

时效性：ms级

可用性：高，基于主从架构实现高可用性

消息可靠性：有较低的概率丢失数据

功能支持：MQ领域的功能极其完备

总结：

非常成熟，功能强大，在早些年业内大量的公司以及项目中都有应用

偶尔会有较低概率丢失消息

现在社区以及国内应用都越来越少，官方社区现在对ActiveMQ 5.x维护越来越少，几个月才发布一个版本

主要是基于解耦和异步来用的，较少在大规模吞吐的场景中使用

**RabbitMQ**

单机吞吐量：万级

topic数量都吞吐量的影响：

时效性：微秒级，延时低是一大特点。

可用性：高，基于主从架构实现高可用性

消息可靠性：

功能支持：基于erlang开发，所以并发能力很强，性能极其好，延时很低

总结：

erlang语言开发，性能极其好，延时很低；

吞吐量到万级，MQ功能比较完备

开源提供的管理界面非常棒，用起来很好用

社区相对比较活跃，几乎每个月都发布几个版本分

在国内一些互联网公司近几年用rabbitmq也比较多一些 但是问题也是显而易见的，RabbitMQ确实吞吐量会低一些，这是因为他做的实现机制比较重。

erlang开发，很难去看懂源码，基本职能依赖于开源社区的快速维护和修复bug。

rabbitmq集群动态扩展会很麻烦，不过这个我觉得还好。其实主要是erlang语言本身带来的问题。很难读源码，很难定制和掌控。

**RocketMQ**

单机吞吐量：十万级

topic数量都吞吐量的影响：topic可以达到几百，几千个的级别，吞吐量会有较小幅度的下降。可支持大量topic是一大优势。

时效性：ms级

可用性：非常高，分布式架构

消息可靠性：经过参数优化配置，消息可以做到0丢失

功能支持：MQ功能较为完善，还是分布式的，扩展性好

总结：

接口简单易用，可以做到大规模吞吐，性能也非常好，分布式扩展也很方便，社区维护还可以，可靠性和可用性都是ok的，还可以支撑大规模的topic数量，支持复杂MQ业务场景

而且一个很大的优势在于，源码是java，我们可以自己阅读源码，定制自己公司的MQ，可以掌控

社区活跃度相对较为一般，不过也还可以，文档相对来说简单一些，然后接口这块不是按照标准JMS规范走的有些系统要迁移需要修改大量代码

**Kafka**

单机吞吐量：十万级，最大的优点，就是吞吐量高。

topic数量都吞吐量的影响：topic从几十个到几百个的时候，吞吐量会大幅度下降。所以在同等机器下，kafka尽量保证topic数量不要过多。如果要支撑大规模topic，需要增加更多的机器资源

时效性：ms级

可用性：非常高，kafka是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用

消息可靠性：经过参数优化配置，消息可以做到0丢失

功能支持：功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用

总结：

kafka的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展

同时kafka最好是支撑较少的topic数量即可，保证其超高吞吐量

kafka唯一的一点劣势是有可能消息重复消费，那么对数据准确性会造成极其轻微的影响，在大数据领域中以及日志采集中，这点轻微影响可以忽略

**最后**

一般的业务系统要引入MQ，最早大家都用ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃

后来大家开始用RabbitMQ，但是确实erlang语言阻止了大量的java工程师去深入研究和掌控他，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司，会去用RocketMQ，确实很不错，但是要想好社区万一突然黄掉的风险

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用RabbitMQ是不错的选择；大型公司，基础架构研发实力较强，用RocketMQ是很好的选择

如果是大数据领域的实时计算、日志采集等场景，用Kafka是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范

**RabbitMQ**

结合erlang语言本身的并发优势，支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正是如此，使的它变的非常重量级，更适合于企业级的开发。

性能较好，但是不利于做二次开发和维护。

**ActiveMQ**

历史悠久的开源项目，是Apache下的一个子项目。已经在很多产品中得到应用，实现了JMS1.1规范，可以和spring-jms轻松融合，实现了多种协议，不够轻巧（源代码比RocketMQ多），支持持久化到数据库，对队列数较多的情况支持不好。

**Kafka**

Apache下的一个子项目，使用scala实现的一个高性能分布式Publish/Subscribe消息队列系统，具有以下特性：

快速持久化：通过磁盘顺序读写与零拷贝机制，可以在O(1)的系统开销下进行消息持久化；

高吞吐：在一台普通的服务器上既可以达到10W/s的吞吐速率；

高堆积：支持topic下消费者较长时间离线，消息堆积量大；

完全的分布式系统：Broker、Producer、Consumer都原生自动支持分布式，依赖zookeeper自动实现复杂均衡；

支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

# Netty NIO

## NIO和传统的IO有什么区别呢？

传统的IO，需要为每个连接创建一个线程，当并发的连接数量非常巨大时，线程所占用的栈内存和CPU线程切换的开销将非常巨大。使用NIO，不再需要为每个线程创建单独的线程，可以用一个含有限数量线程的线程池，甚至一个线程来为任意数量的连接服务。

IO是面向流的，一次可以读取一个或多个字节。NIO是缓冲区的，数据是先被 读/写到buffer中的，你可以控制读取什么位置的数据。

IO是阻塞式IO，当一条线程执行read()或者write()方法时，这条线程会一直阻塞直到读取到了一些数据或者要写出去的数据已经全部写出，在这期间这条线程不能做任何其他的事情。NIO在非阻塞式模式下，允许一条线程从channel中读取数据，通过返回值来判断buffer中是否有数据，如果没有数据，NIO不会阻塞。

Java NIO的selectors允许一条线程去监控多个channels的输入，你可以向一个selector上注册多个channel，然后调用selector的select()方法判断是否有新的连接进来或者已经在selector上注册时channel是否有数据进入。

NIO缺点：

需要额外做的工作是检查你需要的数据是否已经全部到了buffer中，你还需要保证当有更多的数据进入buffer中时，buffer中未处理的数据不会被覆盖

总结

NIO允许你用一个单独的线程或几个线程管理很多个channels（网络的或者文件的），代价是程序的处理和处理IO相比更加复杂

如果你需要同时管理成千上万的连接，但是每个连接只发送少量数据，例如一个聊天服务器，用NIO实现会更好一些，相似的，如果你需要保持很多个到其他电脑的连接，例如P2P网络，用一个单独的线程来管理所有出口连接是比较合适的

如果你只有少量的连接但是每个连接都占有很高的带宽，同时发送很多数据，传统的IO会更适合

## 那么NIO和IO各适用的场景是什么呢？

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，这时候用NIO处理数据可能是个很好的选择。

而如果只有少量的连接，而这些连接每次要发送大量的数据，这时候传统的IO更合适。使用哪种处理数据，需要在数据的响应等待时间和检查缓冲区数据的时间上作比较来权衡选择。

## ?netty的线程模型么？

Netty使用的是Reactor线程模型，Netty使用EventLoop来处理连接上的读写事件，而一个连接上的所有请求都保证在一个EventLoop中被处理，一个EventLoop中只有一个Thread，所以也就实现了一个连接上的所有事件只会在一个线程中被执行。一个EventLoopGroup包含多个EventLoop，可以把一个EventLoop当做是Reactor线程模型中的一个线程，而一个EventLoopGroup类似于一个ExecutorService，

## ？Netty半包等问题

# Dubbo

## 基础

Dubbo是基于远程服务调用的分布式框架，其核心部分包含：集群容错，多协议支持，软负载均衡，地址路由，动态配置等集群支持。

RPC (Remote Procedure Call)即远程过程调用，传统意义上的 RPC 是指长连接数据交互。

Dubbo默认使用netty作为通信框架，服务调用默认是阻塞的，可以异步调用，没有返回值的可以这么做。

Dubbo默认使用 zookeeper 注册中心，还有 Multicast注册中心, Redis注册中心, Simple注册中心。通过创建和监听znode节点来实现服务注册和发现。ZooKeeper的节点是通过像树一样的结构来进行维护的，并且每一个节点通过路径来标示以及访问。除此之外，每一个节点还拥有自身的一些信息，包括：数据、数据长度、创建时间、修改时间等等。

Dubbo默认使用 dubbo 协议,默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。 hessian是一个采用二进制格式传输的服务框架，相对传统soap web service，更轻量，更快速。

**为什么要用Dubbo**

内部使用了 Netty、Zookeeper，保证了高性能高可用性。使用 Dubbo 可以将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，可用于提高业务复用灵活扩展，使前端应用能更快速的响应多变的市场需求。

Dubbo 使用的是 RPC 通信，而 Spring Cloud 使用的是 HTTP RESTFul 方式。

**Dubbo底层通信是通过支持异步、事件驱动的NIO网络编程框架**

如：Netty、Mina、Grizzly，此框架是典型的Reactor模式使用，使得单个线程处理多个请求，且支持多请求并行执行，NIO接受请求处理流程是读取请求数据解码执行业务逻辑，编码，发送回应消息，Dubbo是对NIO框架的再次抽象封装，加入一些Dubbo需要的逻辑，通过抽象扩展Handler完成，如HeaderExchangerHandler完成请求-响应模式、同步转异步模式消息发送，AllChannelHandler通过线程池完成请求、响应、连接等并行执行，下面会详细介绍。

**Dubbo可以对结果进行缓存吗**

可以，Dubbo 提供了声明式缓存，用于加速热门数据的访问速度，以减少用户加缓存的工作量。

**Dubbo服务之间的调用是阻塞的吗**

默认是同步等待结果阻塞的，支持异步调用。

Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

**服务提供者能实现失效踢出是什么原理**

服务失效踢出基于 Zookeeper 的临时节点原理。

**Dubbo的管理控制台能做什么**

管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。

**Double服务框架其核心部分**：

1.远程通讯：提供对多种基于长连接的NIO框架抽象封装，包括多线程模型，序列化，以及“请求-响应”模式的信息交换方式

2.集群容错：提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持

3.自动发现，基于注册中心目录服务，是服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## ？Dubbo完整的一次调用链路介绍；

dubbo服务初始化以后，会将service接口生成代理对象，将接口的方法进行包装。每次调用都会调用到InvocationHandler的invoke()方法，此方法中会进行远程服务调用一些列复杂过程，诸如网络通信，编码，解码，序列化等，然后将结果返回。在InvokerInvocationHandler.invoker()方法中，最后调用invoker.invoke(new RpcInvocation(method,args))然后调用到FailfastClusterInvoker.invoke()，此对象持有接口服务的RegistryDirectory，里面含有远程提供者的具体信息，是在服务消费者初始化时，通过订阅zk相应节点得到的。此方法的调用过程比较复杂，会调用负载均衡算法，根据一定的策略，选取一个提供者，生成DubboInvoker对象,再进行DubboInvoke.invoke()的调用。进入HeaderExchangeClient.request()方法，HeaderExchangeClient里面封装了NettyClient，由此将请求交予netty框架发送。

请求到达服务端以后，netty服务端会先对数据进行解码，解码之后进入NettyHandler的messageReceived()方法，然后进入到DubboProtocol内部类ExchangeHandlerAdapter的reply()方法，此invoker为包装类型，包含filter执行链、提供端接口实现类的包装类，执行invoke()方法，先调用filter执行链，然后调用AbstractProxyInvoker实现类的doInvoke()方法，进行调用的包装类的invokeMethod()方法，包装类持有具体实现类，至此调用完毕，将结果封装成RpcResult返回,回到HeaderExchangeHandler.receive()方法，调用channel.send()写到客户端。

## dubbo的服务暴露过程

Dubbo会在Spring实例化完bean之后，在刷新容器最后一步发布ContextRefreshEvent事件的时候，通知实现了ApplicationListener的ServiceBean类进行回调onApplicationEvent 事件方法，dubbo会在这个方法中调用ServiceBean父类ServiceConfig的export方法，而该方法真正实现了服务的（异步或者非异步）发布。

由服务配置类 ServiceConfig 进行初始化工作及服务暴露入口，首先进去执行该类的export()方法。

export方法先判断是否需要延迟暴露（这里我们使用的是不延迟暴露），然后执行doExport方法。

doExport方法先执行一系列的检查方法，然后调用doExportUrls方法。检查方法会检测dubbo的配置是否在Spring配置文件中声明，没有的话读取properties文件初始化。

doExportUrls方法先调用loadRegistries获取所有的注册中心url，然后遍历调用doExportUrlsFor1Protocol方法。对于在标签中指定了registry属性的Bean，会在加载BeanDefinition的时候就加载了注册中心。

原文：<https://blog.csdn.net/yanpenglei/article/details/80261762>

## Dubbo配置

**Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做？**

在Provider上配置的Consumer端属性：actives，消费者端，最大并发调用限制，即当Consumer对一个服务的并发调用到上限后，新调用会Wait直到超时

Provider上配置的Provider端属性： executes，一个服务提供者并行执行请求上限，即当Provider对一个服务的并发调用到上限后，新调用会Wait（Consumer可能到超时）。在方法上配置（dubbo:method ）则并发限制针对方法，在接口上配置（dubbo:service），则并发限制针对服务。

**Dubbo启动的时候支持几种配置方式？**

根据 DUBBO 官方文档，配置 DUBBO 有 4 种方式，分别是：

XML 配置文件方式

properties 配置文件方式

annotation 配置方式

API 配置方式

**配置的优先级:**

1. 方法级配置别优于接口级别，即小Scope优先

2. Consumer端配置优于Provider配置，优于全局配置

3. Dubbo Hard Code的配置值（默认）

如：在dubbo的provider和consumer的配置文件中，如果都配置了timeout的超时时间，dubbo默认以consumer中配置的时间为准

**说说核心的配置有哪些？**

核心配置有：

1) dubbo:service/

2) dubbo:reference/

3) dubbo:protocol/

4) dubbo:registry/

5) dubbo:application/

6) dubbo:provider/

7) dubbo:consumer/va

8) dubbo:method/

**Dubbo 支持哪些协议，每种协议的应用场景，优缺点？**

dubbo： 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

rmi： 采用 JDK 标准的 rmi 协议实现，多个短连接，TCP 协议传输，同步传输，一般使用在大数据或文件传递方面。java 序列化存在安全漏洞；

webservice:基于 WebService 的远程调用协议，多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；

http： 基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；

hessian： 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache： 基于 memcached 实现的 RPC 协议

redis： 基于 redis 实现的 RPC 协议

thrift:

**Dubbo内置了哪几种服务容器**

Spring Container

Jetty Container

Log4j Container

**Dubbo里面有哪几种节点角色**

Provider,Consumer,Registry,Monitor,Container

**Dubbo有哪几种配置方式**

Spring 配置方式

Java API 配置方式

**Dubbo 核心的配置有哪些**



**在 Provider 上可以配置的 Consumer 端的属性有哪些**

1）timeout：方法调用超时

2）retries：失败重试次数，默认重试 2 次

3）loadbalance：负载均衡算法，默认随机

4）actives 消费者端，最大并发调用限制

**Dubbo有哪几种集群容错方案，默认是哪种**



**Dubbo有哪几种负载均衡策略，默认是哪种**



**当一个服务接口有多种实现时怎么做**

当一个接口有多种实现时，可以用 group 属性来分组，服务提供方和消费方都指定同一个 group 即可。

## Dubbo框架结构

服务层：

Dubbo最大的特点就是按照分层的方式来架构，使用这种方式可以实现各个层之间最大限度地松耦合。Dubbo的框架设计一共划分为10层，有服务接口层，配置层，服务代理层，集群层，监控层，服务注册层，远程调用层，信息交换层，网络传输层，数据序列化层。其中我觉得最主要的就是服务代理层，服务注册层与集群层，服务代理层会将根据你写的接口生成一个代理类，用于处理请求，而服务注册层则会将服务地址暴露给注册中心，请求过来后再由底层转发给各个代理实现类去处理，并响应。

服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。

服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。

服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。

集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。

监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。

远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。

信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。

网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。

数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

## Doubbo容错方案

**在使用过程中都遇到了些什么问题？ 如何解决的？**

比较常遇到就是因为服务端操作比较繁琐，造成超时的问题，之前做这个系统时因为有个对账模块因为数据量大偶尔会这样，这个要么优化代码逻辑，要么配置长一点的超时时间。

另一个就是因为传递的对象没有实现序列化接口而报错，印象比较深刻的就是list的sublist方法返回的list是未实现序列化的，这个要么换其它数据格式，要么转换成实现了序列化的集合。

其它的比如配置未生效，启动抱服务不可用，服务注册不上等等，一般都是环境或配置的问题，根据异常提示仔细检查一下就可以了。

**服务调用超时问题怎么解决**

dubbo在调用服务不成功时，默认是会重试两次的。这样在服务端的处理时间超过了设定的超时时间时，就会有重复请求，比如在发邮件时，可能就会发出多份重复邮件，执行注册请求时，就会插入多条重复的注册数据，那么怎么解决超时问题呢？如下

对于核心的服务中心，去除dubbo超时重试机制，并重新评估设置超时时间。 业务处理代码必须放在服务端，客户端只做参数验证和服务调用，不涉及业务流程处理 全局配置实例

**服务上线怎么兼容旧版本**

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

**Dubbo集群健壮性：**

监控中心宕掉不影响使用，只是丢失部分采样数据

数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务

注册中心对等集群，任意一台宕掉后，将自动切换到另一台

注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯

服务提供者无状态，任意一台宕掉后，不影响使用。

服务提供者全部宕掉后，服务消费者应用无法使用，并无限次重连等待服务提供者恢复

## 其它

**Hessian原理与协议简析：**

http的协议约定了数据传输的方式，hessian也无法改变太多：

1) hessian中client与server的交互，基于http-post方式。

2) hessian将辅助信息，封装在http header中，比如“授权token”等，我们可以基于http-header来封装关于“安全校验”“meta数据”等。hessian提供了简单的”校验”机制。

3) 对于hessian的交互核心数据，比如“调用的方法”和参数列表信息，将通过post请求的body体直接发送，格式为字节流。

4) 对于hessian的server端响应数据，将在response中通过字节流的方式直接输出。

**服务提供者能实现失效踢出是什么原理？**

答：服务失效踢出基于 zookeeper 的临时节点原理。

**服务上线怎么不影响旧版本？**

答：采用多版本开发，不影响旧版本。在配置中添加version来作为版本区分

**如何解决服务调用链过长的问题？**

答：可以结合 zipkin 实现分布式服务追踪。

**dubbo 在安全机制方面如何解决的**？

dubbo 通过 token 令牌防止用户绕过注册中心直连，然后在注册中心管理授权，dubbo 提供了黑白名单，控制服务所允许的调用方。

**集群容错怎么做？**

答：读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错。

**dubbo 和 dubbox 之间的区别？**

答：dubbox 是当当网基于 dubbo 上做了一些扩展，如加了服务可 restful 调用，更新了开源组件等。

**测试和生产公用一套zookeeper，怎么保证消费不冲突**

1.dubbo白名单（Filter过滤器）

首先实现com.alibaba.dubbo.rpc.Filter接口：并在实现的接口中根据服务的ip过滤掉不想用的服务。

其次在resources目录下添加纯文本文件META-INF/dubbo/com.alibaba.dubbo.rpc.Filter，内容如下： xxxFilter=com.xxx.AuthorityFilter

并且修改dubbo的provider配置文件，在dubbo:provider中添加配置的filter， 内容如下：<dubbo:provider filter="xxxFilter" />

2.服务分组

<!--服务-->

<dubbo:service group="feedback" interface="com.xxx.IndexService" />

<dubbo:service group="member" interface="com.xxx.IndexService" />

<!--引用-->

<dubbo:reference id="feedbackIndexService" group="feedback" interface="com.xxx.IndexService" />

<dubbo:reference id="memberIndexService" group="member" interface="com.xxx.IndexService" />

3. 多版本

<dubbo:service interface="com.foo.BarService" version="1.0.0" />

# Zookeeper

## ZooKeeper用途

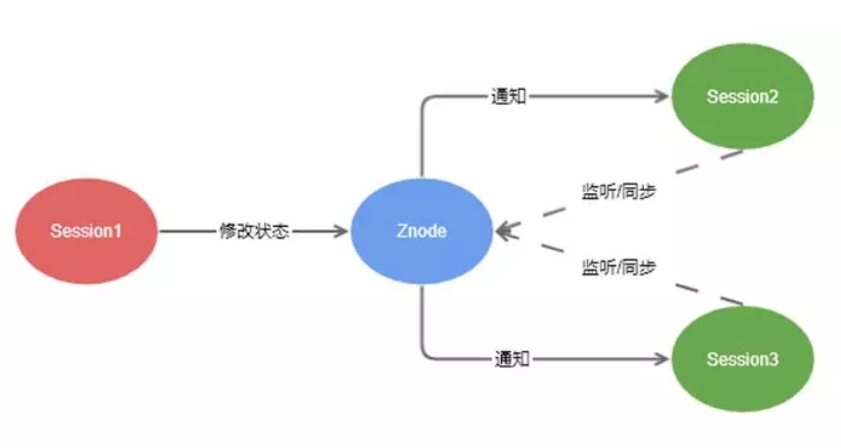
Zookeeper的用途包括统一配置管理、统一命名服务、分布式锁、集群管理。

**统一配置管理**：

我们可以将common.yml这份配置放在ZooKeeper的Znode节点中，系统A、B、C监听着这个Znode节点有无变更，如果变更了，及时响应。

**统一命名服务**：

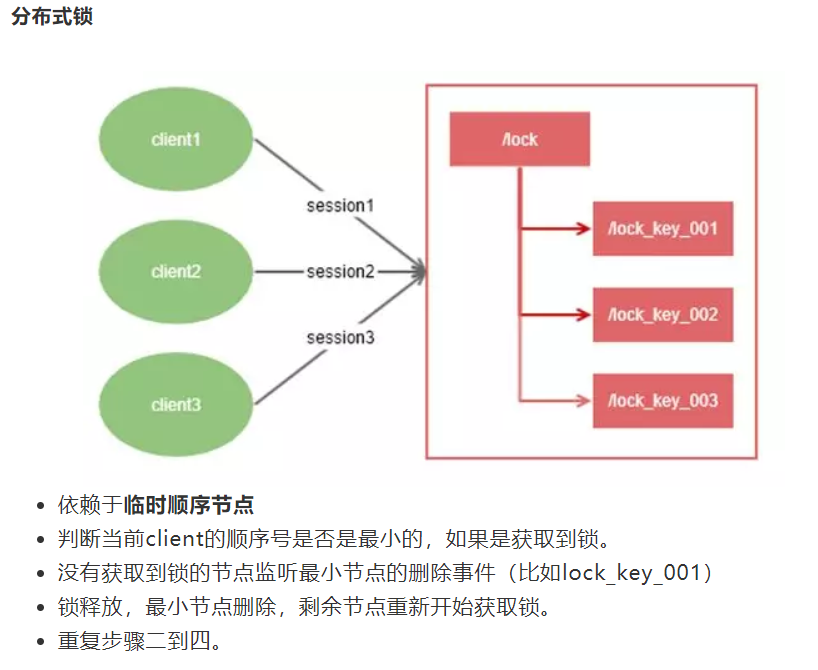
将几个服务的ip存于一个Znode下



**分布式锁**：

系统A、B、C都去访问/locks节点。访问的时候会创建带顺序号的临时/短暂(EPHEMERAL\_SEQUENTIAL)节点，比如，系统A创建了id\_000000节点，系统B创建了id\_000002节点，系统C创建了id\_000001节点。

接着，拿到/locks节点下的所有子节点(id\_000000,id\_000001,id\_000002)，判断自己创建的是不是最小的那个节点。如果是，则拿到锁（释放锁：执行完操作后，把创建的节点给删掉）。如果不是，则监听比自己要小1的节点变化。



**比较：**

从理解的难易程度角度（从低到高）

数据库 > 缓存（Redis） > Zookeeper

从实现的复杂性角度（从低到高）

Zookeeper >= 缓存（Redis） > 数据库

从性能角度（从高到低）

缓存（Redis） > Zookeeper >= 数据库

从可靠性角度（从高到低）

Zookeeper > 缓存（Redis） > 数据库

**集群管理**：

以我们三个系统A、B、C为例，在ZooKeeper中创建临时节点即可（/groupMember/A，/groupMember/B，/groupMember/C）, 只要系统A挂了，那/groupMember/A这个节点就会删除，通过监听groupMember下的子节点，系统B和C就能够感知到系统A已经挂了。(新增也是同理)

**实现方式**

ZooKeeper通过Znode的节点类型+监听机制来实现以上功能

ZooKeeper的数据结构，跟Unix文件系统非常类似，可以看做是一颗树，每个节点叫做ZNode。每一个节点可以通过路径来标识，Znode分为两种类型：

短暂/临时(Ephemeral)：当客户端和服务端断开连接后，所创建的Znode(节点)会自动删除

持久(Persistent)：当客户端和服务端断开连接后，所创建的Znode(节点)不会删除

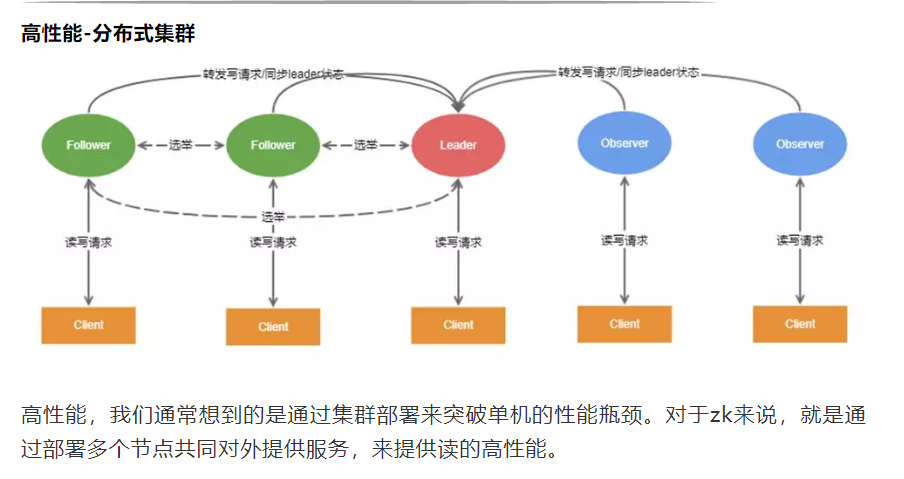
ZooKeeper还配合了监听器才能够做那么多事的。常见的监听场景有以下两项：

监听Znode节点的数据变化

监听子节点的增减变化

## 高性能分布式集群

**集群：**



Master/Slave模式。

在zookeeper中部署多台节点对外提供服务，客户端可以连接到任意一个节点。

每个节点的数据都是一样的。

节点根据角色分为Leader节点与Learner节点（包括Follower节点与Observer节点）。

集群中，只有一个Leader节点，完成所有的写请求处理。

每次写请求都会生成一个全局的唯一的64位整型的事务ID(可以理解为全局的数据的版本号)。

Learner节点可以有很多，每个Leaner可以独自处理读请求，转写请求到Leader节点。

当Leader节点挂掉后，会从Follower节点中通过选举方式选出一个Leader提供对外服务。

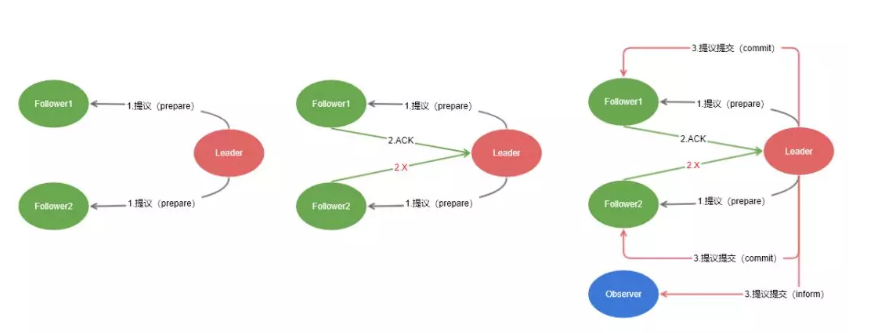
Follower节点与Observer节点区别在于不参与选举和提议的事务过半处理。

集群通常是按照奇数个节点进行部署（偶然太对容灾没啥影响，浪费机器）。

**数据一致性**

**（zab协议-原子广播协议）**

通过集群的部署，根据CAP原理，这样，可能导致同一个数据在不同节点上的数据不一致。zookeeper通过zab原子广播协议来保证数据在每一个节点上的一致性。原子广播协议（类似2PC提交协议）大概分为3个步骤。



Leader包装写请求，生成唯一zxid，发起提议，广播给所有Follower。

Follower收到提议后，写入本地事务日志，根据自身情况，是否同意该事务的提交。

Leader收到过半的Follower同意，自己先添加事务。然后对所有的Learner节点发送提交事务请求。

需要说明的是，zookeeper对数据一致性的要求是：

顺序一致性：严格按照事务发起的顺序执行写操作。

原子性：所有事务请求的结果在集群中的所有节点上的应用情况是一致的。

单一视图：客户端访问任何一个节点，看到的数据模型都是一致的。

实时性：保证在极小一段时间客户端最终可以从服务读取最新数据状态（如果要实时，需要客户端调用syn方法）。

**可用性-leader选举**

**（zab协议-崩溃恢复协议）**

在整个集群中，写请求都集中在一个Leader节点上，如果Leader节点挂了咋办呢？

当集群初始化或Follower无法联系上Leader节点的时候，每个Follower开始进入选举模式。选举步骤如下：

Follower节点第一次投票先投自己，然后将自己的选票广播给剩余的Follower节点。

Follower节点接收到其他的选票。

选票比较：比较自己的与接收的选票的投票更有。

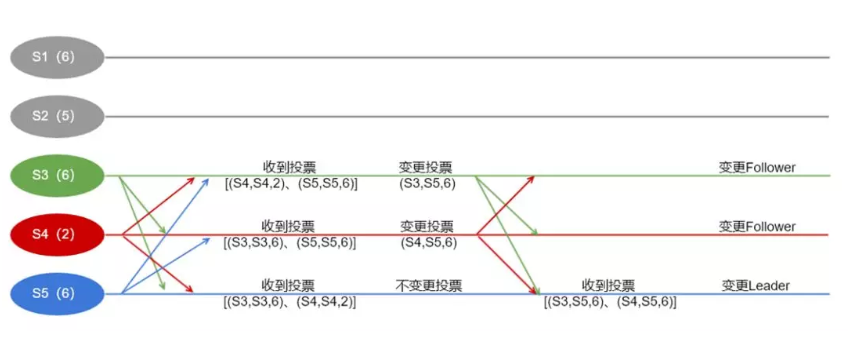
如果资金的选票不是最优选票，变更自己的选票，投最优选票的节点。

统计自己收到的选票，如果某个节点获得了过半的节点的投票。确认该节点为新的Leader节点。

确认Leader节点后，每个节点变更自己的角色。完成投票选举。

选举原则：谁的数据最新，谁就有优先被选为Leader的资格。

举个例子，假如现在zk集群有5个节点，然后挂掉了2个节点。剩余节点S3,S4,S6开始进行选举，他们的最大事务ID分别是6,2,6。定义投票结构为（投票的节点ID，被投节点ID，被投节点最大事务ID）。



初始状态，S3,S4,S5分别投自己，并带上自己的最大事务ID。

S3,S4,S5分别对自己收到的2票与自己的1票做比较。

S5发现自己的是最优投票，不变更投票，S3,S4发现S5的投票是最优解，更改投票。

S3,S4广播自己变更的投票。

最后大家都确认了S5是Leader，S5节点状态变更为Leader节点，S3,S4变更为Follower节点。

到这里，就是选举的主要过程。

**数据的持久化**

zookeeper所有数据都存在内存中。

zookeeper会定期将内存dump到磁盘中，形成数据快照。

zookeeper每次的事务请求，都会先接入到磁盘中，形成事务日志。

全量数据 = 数据快照 + 事务日志。

## Eureka比Zookeeper比较

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

3.1 Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

3.2 Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)

3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

4. 总结

Eureka作为单纯的服务注册中心来说要比zookeeper更加“专业”，因为注册服务更重要的是可用性，我们可以接受短期内达不到一致性的状况。不过Eureka目前1.X版本的实现是基于servlet的Java web应用，它的极限性能肯定会受到影响。期待正在开发之中的2.X版本能够从servlet中独立出来成为单独可部署执行的服务。

## Zookeeper选举原理

如何在zookeeper集群中选举出一个leader,zookeeper使用了三种算法,具体使用哪种算法,在配置文件中是可以配置的,对应的配置项是”electionAlg”,其中1对应的是LeaderElection算法,2对应的是AuthFastLeaderElection算法,3对应的是FastLeaderElection算法.默认使用FastLeaderElection算法.其他两种算法我没有研究过,就不多说了.

要理解这个算法,最好需要一些paxos算法的理论基础.

1) 数据恢复阶段

首先,每个在zookeeper服务器先读取当前保存在磁盘的数据,zookeeper中的每份数据,都有一个对应的id值,这个值是依次递增的,换言之,越新的数据,对应的ID值就越大.

2) 向其他节点发送投票值

在读取数据完毕之后,每个zookeeper服务器发送自己选举的leader（首次选自己）,这个协议中包含了以下几部分的数据:

a)所选举leader的id(就是配置文件中写好的每个服务器的id) ,在初始阶段,每台服务器的这个值都是自己服务器的id,也就是它们都选举自己为leader.

b) 服务器最大数据的id,这个值大的服务器,说明存放了更新的数据.

c)逻辑时钟的值,这个值从0开始递增,每次选举对应一个值,也就是说: 如果在同一次选举中,那么这个值应该是一致的 ; 逻辑时钟值越大,说明这一次选举leader的进程更新.

d) 本机在当前选举过程中的状态,有以下几种:LOOKING,FOLLOWING,OBSERVING,LEADING,顾名思义不必解释了吧.

服务器接受来自其他各个服务器的投票，并判断投票的有效性（包括检查是否是本轮投票、是否来自LOOKING状态的服务器）;

处理投票（服务器将自己的投票和收到的投票进行对比，先检查最大数据的id，大的服务器作为leader；如果ZXID相同，检查服务器id，服务器id大的作为leader；更新投票），将最终的投票重新发出去；

统计投票：每次投票后，服务器统计所有投票，判断是否有过半的服务器收到相同的投票信息；如果是，则选举出了新的leader，如果不是，重新开始投票；

改变服务器状态：follower将自己的状态改为FOLLOWING,leader将自己的状态改为LEADING。

以一个简单的例子来说明整个选举的过程.

假设有五台服务器组成的zookeeper集群,它们的id从1-5,同时它们都是最新启动的,也就是没有历史数据,在存放数据量这一点上,都是一样的.假设这些服务器依序启动,来看看会发生什么.

1) 服务器1启动,此时只有它一台服务器启动了,它发出去的报没有任何响应,所以它的选举状态一直是LOOKING状态

2) 服务器2启动,它与最开始启动的服务器1进行通信,互相交换自己的选举结果,由于两者都没有历史数据,所以id值较大的服务器2胜出,但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3),所以服务器1,2还是继续保持LOOKING状态.

3) 服务器3启动,根据前面的理论分析,服务器3成为服务器1,2,3中的老大,而与上面不同的是,此时有三台服务器选举了它,所以它成为了这次选举的leader.

4) 服务器4启动,根据前面的分析,理论上服务器4应该是服务器1,2,3,4中最大的,但是由于前面已经有半数以上的服务器选举了服务器3,所以它只能接收当小弟的命了.

5) 服务器5启动,同4一样,当小弟.

**对Paxos算法了解多少？**

Paxos是一种分布式一致性算法，

Paxos算法定义了三种角色：Acceptor、Proposer和Learner，其中Proposer是提议发起者，负责接收客户端请求，并将客户端的请求发送到Paxos集群中，以便决定这个值是否可以被批准；Acceptor是提议批准者，负责处理接收到的提议；Learner是学习者，只能学习到已经被批准的值，不能学习没有被批准的值。每一个进程都可以扮演其中任意一种角色。

Paxos完整算法核心过程包含prepare和accept两个阶段，可大致描述为：

1) prepare阶段：Proposer向Acceptor发起提议权申请请求（Prepare消息），Acceptor负责批准Proposer申请的提议权，如果同意就回复Promise消息

2) accept阶段：Proposer一旦获得提议权即可进一步提交变量取值（Accept消息），Acceptor负责批准Proposer提交的变量取值，如果接受就回复Accepted消息

## 其它

**Zookeeper在选举的过程中，还能对外提供服务么？**

在Zookeeper运行期间，如果Leader节点挂了，那么整个Zookeeper集群将暂停对外服务，进入新一轮Leader选举。

**ZooKeeper作为发现服务的问题**

在分布式系统领域有个著名的CAP定理（C-数据一致性；A-服务可用性；P-服务对网络分区故障的容错性)ZooKeeper是个CP的，即任何时刻对ZooKeeper的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性；但是它不能保证每次服务请求的可用性

作为一个分布式协同服务，ZooKeeper非常好，但是对于Service发现服务来说就不合适了；因为对于Service发现服务来说就算是返回了包含不实的信息的结果也比什么都不返回要好；

**zookeeper watch机制**

那么要实现Watch，就必须实现org.apache.zookeeper.Watcher接口，并且将实现类的对象传入到可以Watch的方法中。Zookeeper中所有读操作（getData()，getChildren()，exists()）都可以设置Watch选项。Watch事件具有one-time trigger（一次性触发）的特性，如果Watch监视的Znode有变化，那么就会通知设置该Watch的客户端。

**ZK为什么不提供一个永久性的Watcher注册机制**

不支持用持久Watcher的原因很简单，ZK无法保证性能。

使用watch需要注意的几点

Watches通知是一次性的，必须重复注册.

发生CONNECTIONLOSS之后，只要在session\_timeout之内再次连接上（即不发生SESSIONEXPIRED），那么这个连接注册的watches依然在。

节点数据的版本变化会触发NodeDataChanged，注意，这里特意说明了是版本变化。存在这样的情况，只要成功执行了setData()方法，无论内容是否和之前一致，都会触发NodeDataChanged。

对某个节点注册了watch，但是节点被删除了，那么注册在这个节点上的watches都会被移除。

同一个zk客户端对某一个节点注册相同的watch，只会收到一次通知。

Watcher对象只会保存在客户端，不会传递到服务端。

**创建的临时节点什么时候会被删除，是连接一断就删除吗？延时是多少？**

连接断了之后，ZK不会马上移除临时数据，只有当session过期之后，才会把这个会话建立的临时数据移除。因此，用户需要谨慎设置Session\_TimeOut

**是否可以拒绝单个IP对ZK的访问,操作**

ZK本身不提供这样的功能，它仅仅提供了对单个IP的连接数的限制。你可以通过修改iptables来实现对单个ip的限制；当然，你也可以通过这样的方式来解决。<https://issues.apache.org/jira/browse/ZOOKEEPER-1320>

**ZooKeeper集群中服务器之间是怎样通信的**

Leader服务器会和每一个Follower/Observer服务器都建立TCP连接，同时为每个F/O都创建一个叫做LearnerHandler的实体。LearnerHandler主要负责Leader和F/O之间的网络通讯，包括数据同步，请求转发和Proposal提议的投票等。Leader服务器保存了所有F/O的LearnerHandler。

**出现调用超时com.alibaba.dubbo.remoting.TimeoutException异常怎么办**

通常是业务处理太慢，可在服务提供方执行：jstack PID > jstack.log 分析线程都卡在哪个方法调用上，这里就是慢的原因。如果不能调优性能，请将timeout设大。

**出现java.util.concurrent.RejectedExecutionException或者Thread pool exhausted怎么办？**

RejectedExecutionException表示线程池已经达到最大值，并且没有空闲连，拒绝执行了一些任务。

Thread pool exhausted通常是min和max不一样大时，表示当前已创建的连接用完，进行了一次扩充，创建了新线程，但不影响运行。

原因可能是连接池不够用，请调整dubbo.properites中的：

// 设成一样大，减少线程池收缩开销

dubbo.service.min.thread.pool.size=200

dubbo.service.max.thread.pool.size=200

# Dump文件分析

# 分布式系统所有问题解决思路概括

## 分布式和Dubbo的问题

这是一个老生常谈的话题，也是这几年比较火的话题，说起分布式就一定和 Dubbo 有关系，但是不能仅仅就理解到 Dubbo。首先我们需要思考它解决的问题，为什么要引入 Dubbo这个概念。随着业务的发展、用户量的增长，系统数量增多，调用依赖关系也变得复杂，为了确保系统高可用、高并发的要求，系统的架构也从单体时代慢慢迁移至服务SOA时代，应运而生的 Dubbo 出现了，它作为 RPC 的出现使得我们搭建微服务项目变得简单，但是我们不仅仅要思考Dubbo带来的框架支撑。同时需要思考服务的幂等、分布式事务、服务之间的 Trace 定位、分布式日志、数据对账、重试机制等，与此同时考虑 MQ 对系统的解耦和压力的分担、数据库分布式部署和分库分表、限流、熔断等机制。所以最终总结是不仅仅要看 Dubbo 的使用、原理同时还要思考上下游和一些系统设计的问题，这块相对的知识点较多，可以针对上面抛出来的点各个击破。

## 分布式一致性session

分布式锁

分布式事务

分布式事务是指事务的参与者，支持事务的服务器，资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。

分库分表中间件及部署

## 分布式事物

分布式事物的存在，就是解决数据不一致的情况。分布式系统中有这么一个广为流传的理论：CAP 定理对于一个分布式计算系统来说，不可能同时满足以下三点：

•一致性（Consistency）

•可用性（Availability）

•分区容错性（Partition tolerance）

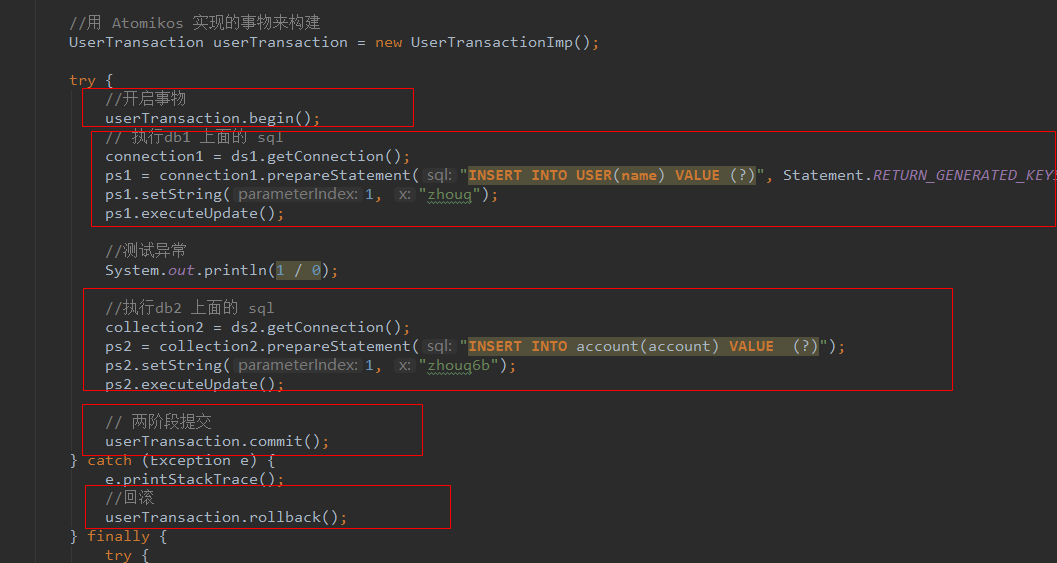
由于分区容错性（P）是必须保证的，那么我们分布式系统就更多是在一致性（CP） 和可用性（AP）上做平衡了，只能同时满足两个条件。其实，大家想想，ZK 是不是就是严格实现了 CP ，而 Eureka 则是保证了 AP。其实分布式事物强调的就是一致性。

**几种分布式事物解决方案**

**2PC**

在说 2PC 之前，我们先了解一下 XA规范 是个什么东西？

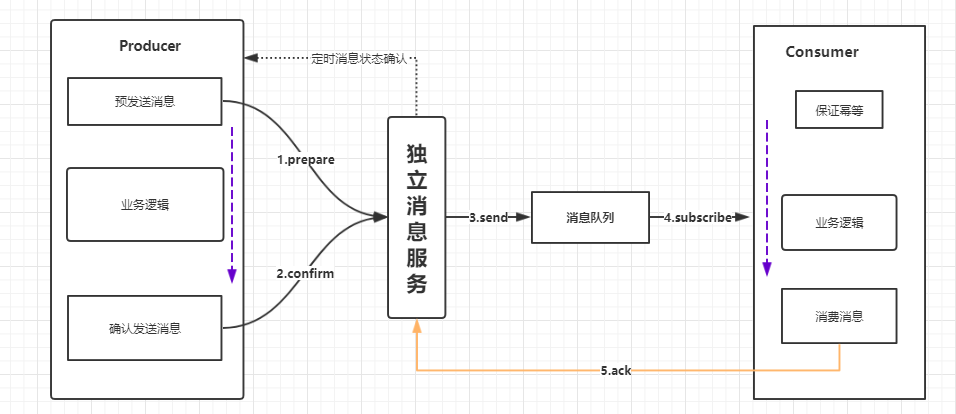
XA规范描述了全局的事务管理器与局部的资源管理器之间的接口。XA规范的目的是允许多个资源（如数据库，应用服务器，消息队列，等等）在同一事务中访问，这样可以使ACID属性跨越应用程序而保持有效。



XA 使用 两阶段提交（2PC） 来保证所有资源同时提交或回滚任何特定的事务。（2PC即两个事物分别预提交，再确认提交，如果报错则回滚。）

**可靠消息最终一致性方案来解决分布式事物的问题**

这个方案，就有消息服务中间件角色参与进来了。



后面的系统，一定要处理 幂等，重试，日志这几个东西。

**TCC 方案**

TCC 的全程分为三个阶段，分别是 Try、Confirm、Cancel：

1.Try阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定或者预留

2.Confirm阶段：这个阶段说的是在各个服务中执行实际的操作

3 .Cancel阶段：如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的回滚操作

还是以转账的例子为例大概思路就是这样的：

1.Try 阶段：先把A 银行账户先冻结 1 块，B银行账户中的资金给预加 1 块。

2.Confirm 阶段：执行实际的转账操作，A银行账户的资金扣减 1块，B 银行账户的资金增加 1 块。

3.Cancel 阶段：如果任何一个银行的操作执行失败，那么就需要回滚进行补偿，就是比如A银行账户如果已经扣减了，但是B银行账户资金增加失败了，那么就得把A银行账户资金给加回去。

## 分布式服务隔离

**按种类隔离**

按照服务种类隔离要从两个纬度来说:即服务提供方和服务调用方！假设我们一个系统有三个服务:订单服务，库存服务，支付服务！服务提供方这个角度而言，每一个服务乃至其对应的数据库，给一个服务器部署。服务调用方这个角度而言，如果不针对被调服务做服务隔离，一个被调服务出问题，就将导致调用方服务不可用！目前业内有信号量隔离和线程池隔离两种隔离方式。

线程池隔离方式，即给每个微服务都初始化出一个线程池，如下图所示，给订单服务和库存服务都初始化出一个线程池，不使用Tomcat线程池中的线程直接调用，而是用相应线程池中的线程去调用！如果此时库存服务不可用了呢？库存服务线程池会被迅速塞满，此时后面进来的新请求发现库存服务线程池满啦，于是乎就不去调库存服务，直接返回！实现方式可以了解一下Hytrix、Sentinel、以及Resilience4j

**按用户隔离**

租户和用户：

用户: 一个环境/系统的一个使用者即该环境/系统的一个用户。

租户:用户从某种粒度上被分到若干组内，每组成为一个租户。

组可以这么理解:用户根据一定的特征去做分组，比如是VIP的一组，不是VIP的一组。

那么所谓的用户隔离，就是按照不同的分组形成不同的服务实例。

有如下三种方式！

方式一:每个租户有独立的服务和独立的数据库

方式二:每个租户有共享的服务和独立的数据库

方式三:每个租户有共享的服务和共享的数据库

## 内存泄露排查思路

1、使用top命令，查看占用内存较高的进程ID。

发现PID为3331的进程占用内存 90.6%。而且是一个Java进程，基本断定是程序问题。

2、使用jmap查看内存情况，并分析是否存在内存泄露。

jmap -heap 3331：查看java 堆（heap）使用情况

jmap -histo 3331：查看堆内存(histogram)中的对象数量及大小

jmap -histo:live 3331：JVM会先触发gc，然后再统计信息

jmap -dump:format=b,file=heapDump 3331：将内存使用的详细情况输出到文件

得到堆dump文件后，可以进行对象分析。如果有大量对象在持续被引用，并没有被释放掉，那就产生了内存泄露，就要结合代码，把不用的对象释放掉。

## 设计一个消息中间件

让你从整体架构，核心流程，数据结构，等各个层面来考虑，你会如何完成这个设计？

首先第一个点，如果有人生产了消息，你作为一个消息中间件，应该如何存储这个数据？你会存储在内存里呢？还是存储在磁盘文件里呢？或者两者都同时共存？你的消费模型是什么样的？比如说一个queue里的数据，是会均匀分配给消费者的各个实例呢？还是会怎么做呢？

接着你应该考虑第二个大的问题，就是你的消息中间件肯定会遇到每天TB级海量数据高并发高吞吐写入的场景。你的数据是不是要分布式的存储？此外你还需要考虑，你的数据分片是不是要可以支撑扩容？

再者，数据宕机场景下的高可用架构。一般分布式系统实现高可用架构，都是采用多副本冗余机制。

还有就是支持数据不丢失的ack机制。

## 抢红包逻辑

方法1：将剩余的红包金额平均再乘二，取这个区间的值。

方法2：取一线段进行切割，依此给抢红包的人。按平均切割，然后将一边的数字随机立减到另一边，再打乱顺序。

## 怎样处理高并发的

高并发就是指网站在同一个时间可以同时处理多个用户请求。要处理高并发，最基本的就是：使用高性能服务器、高性能数据库、高效率编程语言、还有高性能的Web容器。其次就是(1)HTML静态化(2)图片服务器分离(3)数据库集群、库表散列(3)缓存机制(4)镜像(5)负载均衡

## 分布式集群下如何做到唯一序列号

1. 数据库自增长序列或字段:每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。

2. UUID

3. UUID的变种:

1）为了解决UUID不可读，可以使用UUID to Int64的方法。

2）为了解决UUID无序的问题，NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

4. Redis生成ID

5. Twitter的snowflake算法

6. 利用zookeeper生成唯一ID

分布式锁目前比较常用的几种方案：

基于数据库实现分布式锁

基于缓存（redis , memcached）实现分布式锁

基于zookeeper实现分布式锁

## 分布式系统优化

**分布式系统优化原则：**

减少跨机网络交互

尽可能带sharding key

分页优化

减少数据读写热点

切分颗粒度尽可能细

减少锁开销

尽可能规避分布式事务

尽一切可能利用单机资源

单机事务

单机join

好的存储模型，就是尽可能多多做到以下几点：

尽可能走内存

尽可能将一次要查询到到数据物理到放在一起

通过合理到数据冗余，减少走网络到次数

合理并行提升响应时间

读取数据瓶颈，可以通过加slave节点解决

写入瓶颈，用规则sharding和扩容来解决

全部信息索引：

http://weibo.com/1765738567/C72pRgl83

一致性选择

无主机方案

有主机方案

分布式存储

key-value多机扩展

CAP和分布式系统多一致性

## 服务治理

服务治理的核心是：服务性能KPI统计、告警、服务健康管理、灵活的弹性伸缩策略、故障自动迁移、服务限流和服务降级等多种治理手段，保障服务高质量运行

常见的服务治理：服务降级、服务流控、服务动态扩展、超时控制、优先级调度、负载均衡策略调整、分组调整、等

## 高可用系统常用解决手段

1. 系统拆分，读写分离
2. 降级：当故障发生后，怎么减小故障所造成的损失，让系统服务能力不直接降到0

2.1限流

2.2异步调用：弱依赖调用通过消息中间件的方式来实现。

2.3同步调用设置超时时间

2.4失败重试

# 服务器

**Nginx 并发模型**

Nginx 的并发模型是一个多进程并发模型，它的 Master 进程在绑定监听地址端口后 fork 出了多个 Slave 进程共同竞争处理这个服务端套接字接收到的很多客户端连接。

## Tomcat catalina的作用

首先catalina文件保存的是系统的配置信息，比如classpath ，jvm启动参数，以及java文件目录等信息。

而work下面的的catalina是tomcat编译你的项目后，产生的文件，也就是你的项目的class文件,当别人访问的时候，就是访问这里的文件

**Tomcat并发优化**

tomcat并发参数

**maxThreads**：最大的并发请求数，当cpu利用率高的时候，不宜增加线程的个数,当cpu利用率不高，大部分是io阻塞类的操作时，可以适当增加该值。**maxSpareThreads**：Tomcat连接器的最大空闲 socket 线程数

**acceptCount**：当处理任务的线程数达到最大时，接受排队的请求个数

**connectionTimeout**：网络连接超时，单位毫秒

**enableLookups**：若为false则不进行DNS查询，提高业务能力应设置为false

**disableUploadTimeout**：若为true则禁用上传超时

**tomcat内存参数**

 **-server**：一定要作为第一个参数，在多个CPU时性能佳

 -**Xms**：java Heap初始大小。 默认是物理内存的1/64。

 **-Xmx**：java heap最大值。建议均设为物理内存的一半。不可超过物理内存。

 **-XX:PermSize**：设定内存的永久保存区初始大小。缺省值为64M。

 **-XX:MaxPermSize**：设定内存的永久保存区最大 大小。缺省值为64M。

 **-Xmn**：young generation(年轻代)的heap大小。一般设置为Xmx的3、4分之一

## nginx常用配置

user nobody;#运行用户

worker\_processes 1;#启动进程,通常设置成和cpu的数量相等

#pid logs/nginx.pid;

#工作模式及连接数上限

events {

#epoll是多路复用IO(I/O Multiplexing)中的一种方式,

#仅用于linux2.6以上内核,可以大大提高nginx的性能

use epoll;

}

http {

#设定mime类型,类型由mime.type文件定义

include mime.types;

default\_type application/octet-stream;

#设定日志格式

log\_format

access\_log

#tcp\_nopush on;

#keepalive\_timeout 0;#连接超时时间

tcp\_nodelay on;

gzip on;#开启gzip压缩

gzip\_disable "MSIE [1-6].";

#设定请求缓冲

client\_header\_buffer\_size 128k;

large\_client\_header\_buffers 4 128k;

#设定虚拟主机配置

server {

listen 80;#侦听80端口

server\_name www.nginx.cn;#定义使用 www.nginx.cn访问

root html; #定义服务器的默认网站根目录位置

access\_log logs/nginx.access.log main;#设定本虚拟主机的访问日志

#默认请求

location / {

#定义首页索引文件的名称

index index.php index.html index.htm;

}

error\_page 500 502 503 504 /50x.html;# 定义错误提示页面

location = /50x.html {

}

}

}

## linux常用命令

Cd，pwd，help，ls，mkdir，cp，mv，rm，more，less，head，tail，chmod（改变文件或目录权限），chown（改变文件或目录的所有者），which，find，whereis，grep，tar，zip，ping，ifconfig，netstat，Vim，yum，top.

# 开放题

## 如何访问链表中间节点

对于这个问题，我们首先能够想到的就是先遍历一遍整个的链表，然后计算出链表的长度，进而遍历第二遍找出中间位置的数据。这种方式非常简单。

若题目要求只能遍历一次链表，那又当如何解决问题？

可以采取建立两个指针，一个指针一次遍历两个节点，另一个节点一次遍历一个节点，当快指针遍历到空节点时，慢指针指向的位置为链表的中间位置，这种解决问题的方法称为快慢指针方法。

## 有限内存的海量数据处理的题目

一般这类题目的解答无非是以下几种：

分治，hash映射，堆排序，双层桶划分，Bloom Filter，bitmap，数据库索引，mapreduce等。