### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# 体验一下面试官可能会对分布式系统发起的一串连环炮

我之前有一些同学，之前呢主要是做传统行业，外包项目，互联网公司，一直是那种小的公司，技术一直都搞的比较简单。共同的一个问题，就是都没怎么搞过分布式系统，现在互联网公司，一般都是做分布式的系统，大家都不是做底层的分布式系统，分布式存储系统，hadoop hdfs，分布式计算系统，hadoop mapreduce，spark，分布式流式计算系统，storm。

分布式业务系统，把原来用java开发的一个大块系统，给拆分成多个子系统，多个子系统之间互相调用，形成一个大系统的整体。假设原来你做了一个OA系统，里面包含了权限模块、员工模块、请假模块、财务模块，一个工程，里面包含了一堆模块，模块与模块之间会互相去调用，1台机器部署。

现在如果你把他这个系统给拆开，权限系统，员工系统，请假系统，财务系统，4个系统，4个工程，分别在4台机器上部署

一个请求过来，完成这个请求，这个员工系统，调用权限系统，调用请假系统，调用财务系统，4个系统分别完成了一部分的事情，最后4个系统都干完了以后，才认为是这个请求已经完成了。

老师，我就搞不懂，到底什么是分布式系统？

## 1.1 为什么要进行系统拆分？

（1）为什么要进行系统拆分？如何进行系统拆分？拆分后不用dubbo可以吗？dubbo和thrift有什么区别呢？

## 1.2 分布式服务框架

（1）说一下的dubbo的工作原理？注册中心挂了可以继续通信吗？

（2）dubbo支持哪些序列化协议？说一下hessian的数据结构？PB知道吗？为什么PB的效率是最高的？

（3）dubbo负载均衡策略和高可用策略都有哪些？动态代理策略呢？

（4）dubbo的spi思想是什么？

（5）如何基于dubbo进行服务治理、服务降级、失败重试以及超时重试？

（6）分布式服务接口的幂等性如何设计（比如不能重复扣款）？

（7）分布式服务接口请求的顺序性如何保证？

（8）如何自己设计一个类似dubbo的rpc框架？

但是这两年开始兴起和流行了spring cloud，但是我们这里就不讲了，spring cloud刚开始流行，还没有普及，目前普及的是dubbo，出去面试，大部分面试官都是问你dubbo的一些问题

## 1.3 分布式锁

（1）使用redis如何设计分布式锁？使用zk来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？

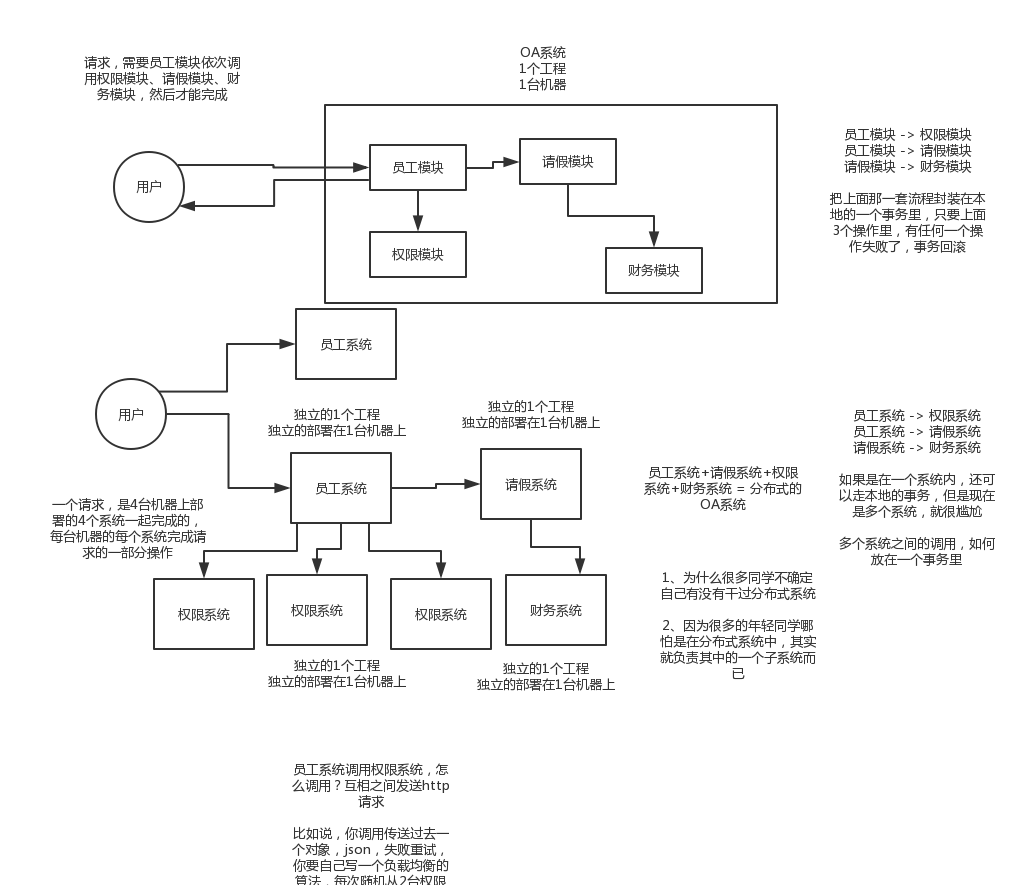
## 1.4 分布式事务

（1）分布式事务了解吗？你们如何解决分布式事务问题的？TCC如果出现网络连不通怎么办？XA的一致性如何保证？

## 1.5 分布式会话

（1）集群部署时的分布式session如何实现？

01\_什么是最简单的分布式系统



# 为什么要把系统拆分成分布式的？为啥要用dubbo？

为什么要进行系统拆分？

如何进行系统拆分？

拆分后不用dubbo可以吗？

从这个问题开始就进行分布式系统环节了，好多同学给我反馈说，现在出去分布式成标配了，没有哪个公司不问问你分布式的事儿。你要是不会分布式的东西，简直这简历没法看，没人会让你去面试。

其实为啥会这样呢？这就是因为整个大行业技术发展的原因

早些年，我印象中在2010年初的时候，整个IT行业，很少有人谈分布式，更不用说微服务，虽然很多BAT等大型公司，因为系统的复杂性，很早就是分布式架构，大量的服务，只不过微服务大多基于自己搞的一套框架来实现而已。

但是确实，那个年代，大家很重视ssh2，很多中小型公司几乎大部分都是玩儿struts2、spring、hibernate，稍晚一些，才进入了spring mvc、spring、mybatis的组合。那个时候整个行业的技术水平就是那样，当年oracle很火，oracle管理员很吃香，oracle性能优化啥的都是IT男的大杀招啊。连大数据都没人提，当年OCP、OCM等认证培训机构，火的不行。

但是确实随着时代的发展，慢慢的，很多公司开始接受分布式系统架构了，这里面尤为对行业有至关重要影响的，是阿里的dubbo，某种程度上而言，阿里在这里推动了行业技术的前进。

正是因为有阿里的dubbo，很多中小型公司才可以基于dubbo，来把系统拆分成很多的服务，每个人负责一个服务，大家的代码都没有冲突，服务可以自治，自己选用什么技术都可以，每次发布如果就改动一个服务那就上线一个服务好了，不用所有人一起联调，每次发布都是几十万行代码，甚至几百万行代码了。

直到今日，我很高兴的看到分布式系统都成行业面试标配了，任何一个普通的程序员都该掌握这个东西，其实这是行业的进步，也是所有IT码农的技术进步。所以既然分布式都成标配了，那么面试官当然会问了，因为很多公司现在都是分布式、微服务的架构，那面试官当然得考察考察你了。

如果有个同学看到这里说，我天，我不知道啥是分布式系统？我也不知道啥是dubbo？那你赶紧百度啊，搜个dubbo入门，去里面体验一下。

分布式系统，我用一句话给你解释一下，实在没时间多唠了，就是原来20万行代码的系统，现在拆分成20个小系统，每个小系统1万行代码。原本代码之间直接就是基于spring调用，现在拆分开来了，20个小系统部署在不同的机器上，得基于dubbo搞一个rpc调用，接口与接口之间通过网络通信来请求和响应。就这个意思。

### （1）为什么要将系统进行拆分？

网上查查，答案极度零散和复杂，很琐碎，原因一大坨。但是我这里给大家直观的感受：

1）要是不拆分，一个大系统几十万行代码，20个人维护一份代码，简直是悲剧啊。代码经常改着改着就冲突了，各种代码冲突和合并要处理，非常耗费时间；经常我改动了我的代码，你调用了我，导致你的代码也得重新测试，麻烦的要死；然后每次发布都是几十万行代码的系统一起发布，大家得一起提心吊胆准备上线，几十万行代码的上线，可能每次上线都要做很多的检查，很多异常问题的处理，简直是又麻烦又痛苦；而且如果我现在打算把技术升级到最新的spring版本，还不行，因为这可能导致你的代码报错，我不敢随意乱改技术。

假设一个系统是20万行代码，其中小A在里面改了1000行代码，但是此时发布的时候是这个20万行代码的大系统一块儿发布。就意味着20万上代码在线上就可能出现各种变化，20个人，每个人都要紧张地等在电脑面前，上线之后，检查日志，看自己负责的那一块儿有没有什么问题。

小A就检查了自己负责的1万行代码对应的功能，确保ok就闪人了；结果不巧的是，小A上线的时候不小心修改了线上机器的某个配置，导致另外小B和小C负责的2万行代码对应的一些功能，出错了

几十个人负责维护一个几十万行代码的单块应用，每次上线，准备几个礼拜，上线 -> 部署 -> 检查自己负责的功能

最近从2013年到现在，5年的时间里，2013年以前，基本上都是BAT的天下；2013年开始，有几个小巨头开始快速的发展，上市，几百亿美金，估值都几百亿美金；2015年，出现了除了BAT以外，又有几个互联网行业的小巨头出现。

BAT工作，在市值几百亿美金的小巨头工作

有某一个小巨头，现在估值几百亿美金的小巨头，5年前刚开始搞的时候，核心的业务，几十个人，维护一个单块的应用

维护单块的应用，在从0到1的环节里，是很合适的，因为那个时候，是系统都没上线，没什么技术挑战，大家有条不紊的开发。ssh + mysql + tomcat，可能会部署几台机器吧。

结果不行了，后来系统上线了，业务快速发展，10万用户 -> 100万用户 -> 1000万用户 -> 上亿用户了。

2）拆分了以后，整个世界清爽了，几十万行代码的系统，拆分成20个服务，平均每个服务就1~2万行代码，每个服务部署到单独的机器上。20个工程，20个git代码仓库里，20个码农，每个人维护自己的那个服务就可以了，是自己独立的代码，跟别人没关系。再也没有代码冲突了，爽。每次就测试我自己的代码就可以了，爽。每次就发布我自己的一个小服务就可以了，爽。技术上想怎么升级就怎么升级，保持接口不变就可以了，爽。

所以简单来说，一句话总结，如果是那种代码量多达几十万行的中大型项目，团队里有几十个人，那么如果不拆分系统，开发效率极其低下，问题很多。但是拆分系统之后，每个人就负责自己的一小部分就好了，可以随便玩儿随便弄。分布式系统拆分之后，可以大幅度提升复杂系统大型团队的开发效率。

但是同时，也要提醒的一点是，系统拆分成分布式系统之后，大量的分布式系统面临的问题也是接踵而来，所以后面的问题都是在围绕分布式系统带来的复杂技术挑战在说。

### （2）如何进行系统拆分？

这个问题说大可以很大，可以扯到领域驱动模型设计上去，说小了也很小，我不太想给大家太过于学术的说法，因为你也不可能背这个答案，过去了直接说吧。还是说的简单一点，大家自己到时候知道怎么回答就行了。

系统拆分分布式系统，拆成多个服务，拆成微服务的架构，拆很多轮的。上来一个架构师第一轮就给拆好了，第一轮；团队继续扩大，拆好的某个服务，刚开始是1个人维护1万行代码，后来业务系统越来越复杂，这个服务是10万行代码，5个人；第二轮，1个服务 -> 5个服务，每个服务2万行代码，每人负责一个服务

如果是多人维护一个服务，<=3个人维护这个服务；最理想的情况下，几十个人，1个人负责1个或2~3个服务；某个服务工作量变大了，代码量越来越多，某个同学，负责一个服务，代码量变成了10万行了，他自己不堪重负，他现在一个人拆开，5个服务，1个人顶着，负责5个人，接着招人，2个人，给那个同学带着，3个人负责5个服务，其中2个人每个人负责2个服务，1个人负责1个服务

我个人建议，一个服务的代码不要太多，1万行左右，两三万撑死了吧

大部分的系统，是要进行多轮拆分的，第一次拆分，可能就是将以前的多个模块该拆分开来了，比如说将电商系统拆分成订单系统、商品系统、采购系统、仓储系统、用户系统，等等吧。

但是后面可能每个系统又变得越来越复杂了，比如说采购系统里面又分成了供应商管理系统、采购单管理系统，订单系统又拆分成了购物车系统、价格系统、订单管理系统。

扯深了实在很深，所以这里先给大家举个例子，你自己感受一下，核心意思就是根据情况，先拆分一轮，后面如果系统更复杂了，可以继续分拆。你根据自己负责系统的例子，来考虑一下就好了。

### （3）拆分后不用dubbo可以吗？

当然可以了，大不了最次，就是各个系统之间，直接基于spring mvc，就纯http接口互相通信呗，还能咋样。但是这个肯定是有问题的，因为http接口通信维护起来成本很高，你要考虑超时重试、负载均衡等等各种乱七八糟的问题，比如说你的订单系统调用商品系统，商品系统部署了5台机器，你怎么把请求均匀地甩给那5台机器？这不就是负载均衡？你要是都自己搞那是可以的，但是确实很痛苦。

所以dubbo说白了，是一种rpc框架，就是本地就是进行接口调用，但是dubbo会代理这个调用请求，跟远程机器网络通信，给你处理掉负载均衡了、服务实例上下线自动感知了、超时重试了，等等乱七八糟的问题。那你就不用自己做了，用dubbo就可以了。

# dubbo的工作原理是啥？注册中心挂了可以继续通信吗？

说一下的dubbo的工作原理？

注册中心挂了可以继续通信吗？

说说一次rpc请求的流程？

MQ、ES、Redis、Dubbo，上来先问你一些思考的问题，原理（kafka高可用架构原理、es分布式架构原理、redis线程模型原理、Dubbo工作原理），

生产环境里可能会碰到的一些问题（每种技术引入之后生产环境都可能会碰到一些问题），系统设计（设计MQ，设计搜索引擎，设计一个缓存，设计rpc框架）

当然比如说，hard面试官，死扣，结合项目死扣细节，百度（深入底层，基础性），阿里（结合项目死扣细节，扣很深的技术底层），小米（数据结构和算法）。

那既然开始聊分布式系统了，自然重点先聊聊dubbo了，毕竟dubbo是目前事实上大部分公司的分布式系统的rpc框架标准，基于dubbo也可以构建一整套的微服务架构。但是需要自己大量开发。

当然去年开始spring cloud非常火，现在大量的公司开始转向spring cloud了，spring cloud人家毕竟是微服务架构的全家桶式的这么一个东西。但是因为很多公司还在用dubbo，所以dubbo肯定会是目前面试的重点，何况人家dubbo现在重启开源社区维护了，未来应该也还是有一定市场和地位的。

既然聊dubbo，那肯定是先从dubbo原理开始聊了，你先说说dubbo支撑rpc分布式调用的架构师啥，然后说说一次rpc请求dubbo是怎么给你完成的，对吧。

### （1）dubbo工作原理

第一层：service层，接口层，给服务提供者和消费者来实现的

第二层：config层，配置层，主要是对dubbo进行各种配置的

第三层：proxy层，服务代理层，透明生成客户端的stub和服务单的skeleton

第四层：registry层，服务注册层，负责服务的注册与发现

第五层：cluster层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务

第六层：monitor层，监控层，对rpc接口的调用次数和调用时间进行监控

第七层：protocol层，远程调用层，封装rpc调用

第八层：exchange层，信息交换层，封装请求响应模式，同步转异步

第九层：transport层，网络传输层，抽象mina和netty为统一接口

第十层：serialize层，数据序列化层

工作流程：

1）第一步，provider向注册中心去注册

2）第二步，consumer从注册中心订阅服务，注册中心会通知consumer注册好的服务

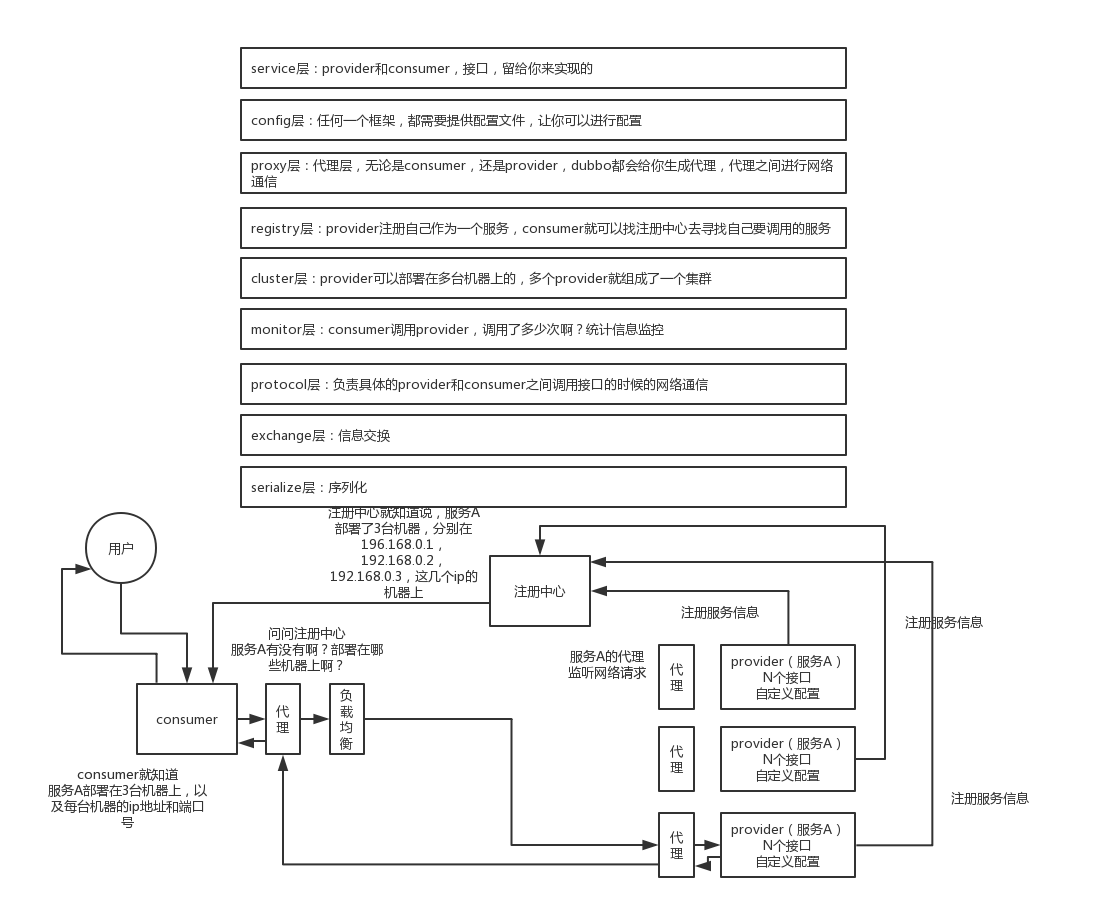
3）第三步，consumer调用provider

4）第四步，consumer和provider都异步的通知监控中心

### （2）注册中心挂了可以继续通信吗？

可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信

01\_dubbo的工作原理



# dubbo都支持哪些通信协议以及序列化协议？

dubbo支持哪些通信协议？

支持哪些序列化协议？

上一个问题，说说dubbo的基本工作原理，那是你必须知道的，至少知道dubbo分成哪些层，然后平时怎么发起rpc请求的，注册、发现、调用，这些是基本的。

接着就可以针对底层进行深入的问问了，比如第一步就可以先问问序列化协议这块，就是平时rpc的时候怎么走的？

### （1）dubbo支持不同的通信协议

1）dubbo协议

dubbo://192.168.0.1:20188

默认就是走dubbo协议的，单一长连接，NIO异步通信，基于hessian作为序列化协议

适用的场景就是：传输数据量很小（每次请求在100kb以内），但是并发量很高

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就100个连接。然后后面直接基于长连接NIO异步通信，可以支撑高并发请求。

否则如果上亿次请求每次都是短连接的话，服务提供者会扛不住。

而且因为走的是单一长连接，所以传输数据量太大的话，会导致并发能力降低。所以一般建议是传输数据量很小，支撑高并发访问。

2）rmi协议

走java二进制序列化，多个短连接，适合消费者和提供者数量差不多，适用于文件的传输，一般较少用

3）hessian协议

走hessian序列化协议，多个短连接，适用于提供者数量比消费者数量还多，适用于文件的传输，一般较少用

4）http协议

走json序列化

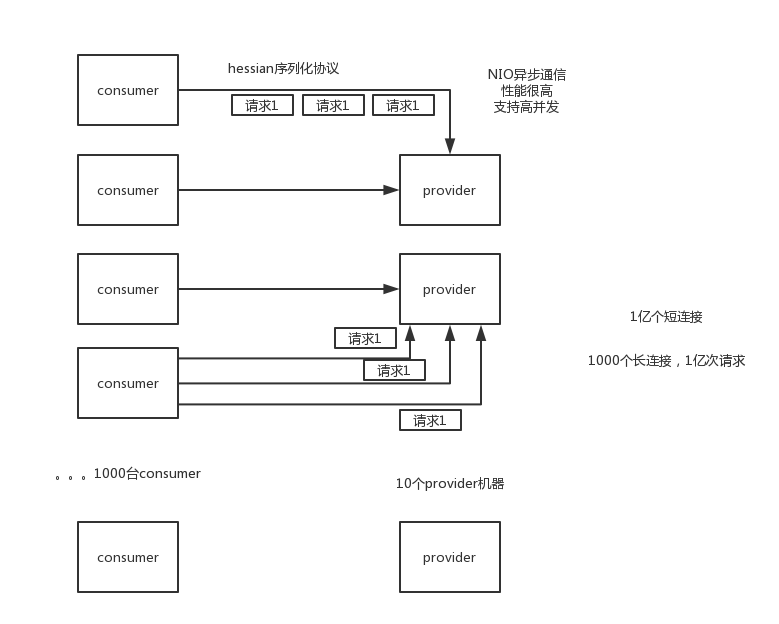
5）webservice

走SOAP文本序列化

### （2）dubbo支持的序列化协议

所以dubbo实际基于不同的通信协议，支持hessian、java二进制序列化、json、SOAP文本序列化多种序列化协议。但是hessian是其默认的序列化协议。

01\_dubbo的网络通信协议



# dubbo支持哪些负载均衡、高可用以及动态代理的策略？

dubbo负载均衡策略和集群容错策略都有哪些？

动态代理策略呢？

继续深问吧，这些都是用dubbo必须知道的一些东西，你得知道基本原理，知道序列化是什么协议，还得知道具体用dubbo的时候，如何负载均衡，如何高可用，如何动态代理。

说白了，就是看你对dubbo熟悉不熟悉：

（1）dubbo工作原理：服务注册，注册中心，消费者，代理通信，负载均衡

（2）网络通信、序列化：dubbo协议，长连接，NIO，hessian序列化协议

（3）负载均衡策略，集群容错策略，动态代理策略：dubbo跑起来的时候一些功能是如何运转的，怎么做负载均衡？怎么做集群容错？怎么生成动态代理？

（4）dubbo SPI机制：你了解不了解dubbo的SPI机制？如何基于SPI机制对dubbo进行扩展？

### （1）dubbo负载均衡策略

1）random loadbalance

默认情况下，dubbo是random load balance随机调用实现负载均衡，可以对provider不同实例设置不同的权重，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。

2）roundrobin loadbalance

还有roundrobin loadbalance，这个的话默认就是均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

跟运维同学申请机器，有的时候，我们运气，正好公司资源比较充足，刚刚有一批热气腾腾，刚刚做好的一批虚拟机新鲜出炉，配置都比较高。8核+16g，机器，2台。过了一段时间，我感觉2台机器有点不太够，我去找运维同学，哥儿们，你能不能再给我1台机器，4核+8G的机器。我还是得要。

3）leastactive loadbalance

这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求

4）consistanthash loadbalance

一致性Hash算法，相同参数的请求一定分发到一个provider上去，provider挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性hash策略。

### （2）dubbo集群容错策略

1）failover cluster模式

失败自动切换，自动重试其他机器，默认就是这个，常见于读操作

1. failfast cluster模式

一次调用失败就立即失败，常见于写操作

3）failsafe cluster模式

出现异常时忽略掉，常用于不重要的接口调用，比如记录日志

4）failbackc cluster模式

失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种

5）forking cluster

并行调用多个provider，只要一个成功就立即返回

6）broadcacst cluster

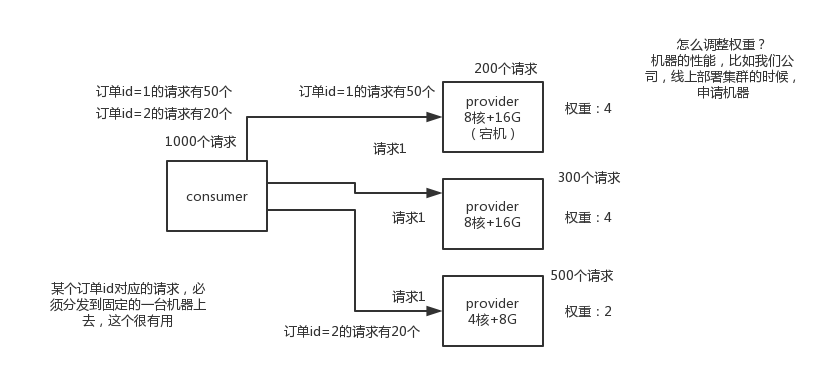
逐个调用所有的provider

### （3）dubbo动态代理策略

默认使用javassist动态字节码生成，创建代理类

但是可以通过spi扩展机制配置自己的动态代理策略

01\_dubbo负载均衡



# SPI是啥思想？dubbo的SPI机制是怎么玩儿的？

dubbo的spi思想是什么？

继续深入问呗，前面一些基础性的东西问完了，确定你应该都ok了解dubbo的一些基本东西，那么问个稍微难一点点的问题，就是spi，先问问你spi是啥？然后问问你dubbo的spi是怎么实现的？

其实就是看看你对dubbo的掌握如何

spi，简单来说，就是service provider interface，说白了是什么意思呢，比如你有个接口，现在这个接口有3个实现类，那么在系统运行的时候对这个接口到底选择哪个实现类呢？这就需要spi了，需要根据指定的配置或者是默认的配置，去找到对应的实现类加载进来，然后用这个实现类的实例对象。

接口A -> 实现A1，实现A2，实现A3

配置一下，接口A = 实现A2

在系统实际运行的时候，会加载你的配置，用实现A2实例化一个对象来提供服务

比如说你要通过jar包的方式给某个接口提供实现，然后你就在自己jar包的META-INF/services/目录下放一个跟接口同名的文件，里面指定接口的实现里是自己这个jar包里的某个类。ok了，别人用了一个接口，然后用了你的jar包，就会在运行的时候通过你的jar包的那个文件找到这个接口该用哪个实现类。

这是jdk提供的一个功能。

比如说你有个工程A，有个接口A，接口A在工程A里是没有实现类的 -> 系统在运行的时候，怎么给接口A选择一个实现类呢？

你就可以自己搞一个jar包，META-INF/services/，放上一个文件，文件名就是接口名，接口A，接口A的实现类=com.zhss.service.实现类A2。让工程A来依赖你的这个jar包，然后呢在系统运行的时候，工程A跑起来，对接口A，就会扫描自己依赖的所有的jar包，在每个jar里找找，有没有META-INF/services文件夹，如果有，在里面找找，有没有接口A这个名字的文件，如果有在里面找一下你指定的接口A的实现是你的jar包里的哪个类？

SPI机制，一般来说用在哪儿？插件扩展的场景，比如说你开发的是一个给别人使用的开源框架，如果你想让别人自己写个插件，插到你的开源框架里面来，扩展某个功能。

经典的思想体现，大家平时都在用，比如说jdbc

java定义了一套jdbc的接口，但是java是没有提供jdbc的实现类

但是实际上项目跑的时候，要使用jdbc接口的哪些实现类呢？一般来说，我们要根据自己使用的数据库，比如msyql，你就将mysql-jdbc-connector.jar，引入进来；oracle，你就将oracle-jdbc-connector.jar，引入进来。

在系统跑的时候，碰到你使用jdbc的接口，他会在底层使用你引入的那个jar中提供的实现类

但是dubbo也用了spi思想，不过没有用jdk的spi机制，是自己实现的一套spi机制。

Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

Protocol接口，dubbo要判断一下，在系统运行的时候，应该选用这个Protocol接口的哪个实现类来实例化对象来使用呢？

他会去找一个你配置的Protocol，他就会将你配置的Protocol实现类，加载到jvm中来，然后实例化对象，就用你的那个Protocol实现类就可以了

微内核，可插拔，大量的组件，Protocol负责rpc调用的东西，你可以实现自己的rpc调用组件，实现Protocol接口，给自己的一个实现类即可。

这行代码就是dubbo里大量使用的，就是对很多组件，都是保留一个接口和多个实现，然后在系统运行的时候动态根据配置去找到对应的实现类。如果你没配置，那就走默认的实现好了，没问题。

@SPI("dubbo")

public interface Protocol {

int getDefaultPort();

@Adaptive

<T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

@Adaptive

<T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

void destroy();

}

在dubbo自己的jar里，在/META\_INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol文件中：

dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol

http=com.alibaba.dubbo.rpc.protocol.http.HttpProtocol

hessian=com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol

所以说，这就看到了dubbo的spi机制默认是怎么玩儿的了，其实就是Protocol接口，@SPI(“dubbo”)说的是，通过SPI机制来提供实现类，实现类是通过dubbo作为默认key去配置文件里找到的，配置文件名称与接口全限定名一样的，通过dubbo作为key可以找到默认的实现了就是com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol。

dubbo的默认网络通信协议，就是dubbo协议，用的DubboProtocol

如果想要动态替换掉默认的实现类，需要使用@Adaptive接口，Protocol接口中，有两个方法加了@Adaptive注解，就是说那俩接口会被代理实现。

啥意思呢？

比如这个Protocol接口搞了俩@Adaptive注解标注了方法，在运行的时候会针对Protocol生成代理类，这个代理类的那俩方法里面会有代理代码，代理代码会在运行的时候动态根据url中的protocol来获取那个key，默认是dubbo，你也可以自己指定，你如果指定了别的key，那么就会获取别的实现类的实例了。

通过这个url中的参数不通，就可以控制动态使用不同的组件实现类

好吧，那下面来说说怎么来自己扩展dubbo中的组件

自己写个工程，要是那种可以打成jar包的，里面的src/main/resources目录下，搞一个META-INF/services，里面放个文件叫：com.alibaba.dubbo.rpc.Protocol，文件里搞一个my=com.zhss.MyProtocol。自己把jar弄到nexus私服里去。

然后自己搞一个dubbo provider工程，在这个工程里面依赖你自己搞的那个jar，然后在spring配置文件里给个配置：

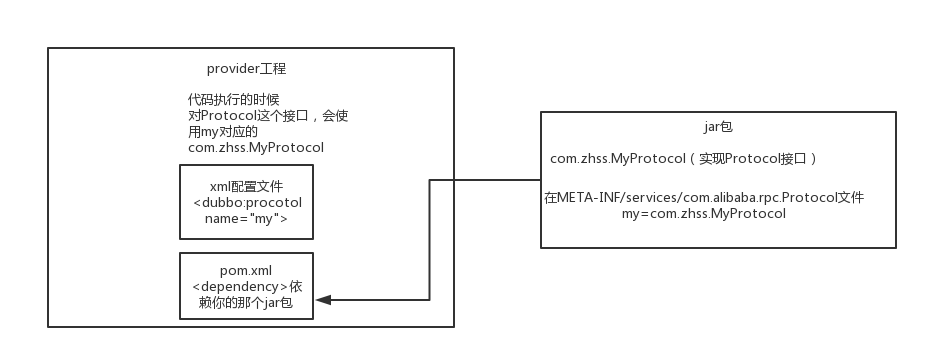
<dubbo:protocol name=”my” port=”20000” />

这个时候provider启动的时候，就会加载到我们jar包里的my=com.zhss.MyProtocol这行配置里，接着会根据你的配置使用你定义好的MyProtocol了，这个就是简单说明一下，你通过上述方式，可以替换掉大量的dubbo内部的组件，就是扔个你自己的jar包，然后配置一下即可。

dubbo里面提供了大量的类似上面的扩展点，就是说，你如果要扩展一个东西，只要自己写个jar，让你的consumer或者是provider工程，依赖你的那个jar，在你的jar里指定目录下配置好接口名称对应的文件，里面通过key=实现类。

然后对对应的组件，用类似<dubbo:protocol>用你的哪个key对应的实现类来实现某个接口，你可以自己去扩展dubbo的各种功能，提供你自己的实现。

01\_dubbo的SPI原理



# 基于dubbo如何做服务治理、服务降级以及重试？

如何基于dubbo进行服务治理、服务降级、失败重试以及超时重试？

服务治理，这个问题如果问你，其实就是看看你有没有服务治理的思想，因为这个是做过复杂微服务的人肯定会遇到的一个问题。

服务降级，这个是涉及到复杂分布式系统中必备的一个话题，因为分布式系统互相来回调用，任何一个系统故障了，你不降级，直接就全盘崩溃？那就太坑爹了吧

失败重试，分布式系统中网络请求如此频繁，要是因为网络问题不小心失败了一次，是不是要重试？

超时重试，同上，如果不小心网络慢一点，超时了，如何重试？

（1）dubbo工作原理：服务注册，注册中心，消费者，代理通信，负载均衡

（2）网络通信、序列化：dubbo协议，长连接，NIO，hessian序列化协议

（3）负载均衡策略，集群容错策略，动态代理策略：dubbo跑起来的时候一些功能是如何运转的，怎么做负载均衡？怎么做集群容错？怎么生成动态代理？

（4）dubbo SPI机制：你了解不了解dubbo的SPI机制？如何基于SPI机制对dubbo进行扩展？

（5）dubbo的服务治理、降级、重试

### （1）服务治理

1）调用链路自动生成

一个大型的分布式系统，或者说是用现在流行的微服务架构来说吧，分布式系统由大量的服务组成。那么这些服务之间互相是如何调用的？调用链路是啥？说实话，几乎到后面没人搞的清楚了，因为服务实在太多了，可能几百个甚至几千个服务。

那就需要基于dubbo做的分布式系统中，对各个服务之间的调用自动记录下来，然后自动将各个服务之间的依赖关系和调用链路生成出来，做成一张图，显示出来，大家才可以看到对吧。

服务A -> 服务B -> 服务C

-> 服务E

-> 服务D

-> 服务F

-> 服务W

2）服务访问压力以及时长统计

需要自动统计各个接口和服务之间的调用次数以及访问延时，而且要分成两个级别。一个级别是接口粒度，就是每个服务的每个接口每天被调用多少次，TP50，TP90，TP99，三个档次的请求延时分别是多少；第二个级别是从源头入口开始，一个完整的请求链路经过几十个服务之后，完成一次请求，每天全链路走多少次，全链路请求延时的TP50，TP90，TP99，分别是多少。

这些东西都搞定了之后，后面才可以来看当前系统的压力主要在哪里，如何来扩容和优化啊

3）其他的

服务分层（避免循环依赖），调用链路失败监控和报警，服务鉴权，每个服务的可用性的监控（接口调用成功率？几个9？）99.99%，99.9%，99%

### （2）服务降级

比如说服务A调用服务B，结果服务B挂掉了，服务A重试几次调用服务B，还是不行，直接降级，走一个备用的逻辑，给用户返回响应

public interface HelloService {

void sayHello();

}

public class HelloServiceImpl implements HelloService {

public void sayHello() {

System.out.println("hello world......");

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

<dubbo:application name="dubbo-provider" />

<dubbo:registry address="zookeeper://127.0.0.1:2181" />

<dubbo:protocol name="dubbo" port="20880" />

<dubbo:service interface="com.zhss.service.HelloService" ref="helloServiceImpl" timeout="10000" />

<bean id="helloServiceImpl" class="com.zhss.service.HelloServiceImpl" />

</beans>

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

<dubbo:application name="dubbo-consumer" />

<dubbo:registry address="zookeeper://127.0.0.1:2181" />

<dubbo:reference id="fooService" interface="com.test.service.FooService" timeout="10000" check="false" mock="return null">

</dubbo:reference>

</beans>

现在就是mock，如果调用失败统一返回null

但是可以将mock修改为true，然后在跟接口同一个路径下实现一个Mock类，命名规则是接口名称加Mock后缀。然后在Mock类里实现自己的降级逻辑。

public class HelloServiceMock implements HelloService {

public void sayHello() {

// 降级逻辑

}

}

### （3）失败重试和超时重试

所谓失败重试，就是consumer调用provider要是失败了，比如抛异常了，此时应该是可以重试的，或者调用超时了也可以重试。

<dubbo:reference id="xxxx" interface="xx" check="true" async="false" retries="3" timeout="2000"/>

某个服务的接口，要耗费5s，你这边不能干等着，你这边配置了timeout之后，我等待2s，还没返回，我直接就撤了，不能干等你

如果是超时了，timeout就会设置超时时间；如果是调用失败了自动就会重试指定的次数

你就结合你们公司的具体的场景来说说你是怎么设置这些参数的，timeout，一般设置为200ms，我们认为不能超过200ms还没返回

retries，3次，设置retries，还一般是在读请求的时候，比如你要查询个数据，你可以设置个retries，如果第一次没读到，报错，重试指定的次数，尝试再次读取2次

# 分布式系统中接口的幂等性该如何保证？比如不能重复扣款？

分布式服务接口的幂等性如何设计（比如不能重复扣款）？

从这个问题开始，面试官就已经进入了实际的生产问题的面试了

一个分布式系统中的某个接口，要保证幂等性，该如何保证？这个事儿其实是你做分布式系统的时候必须要考虑的一个生产环境的技术问题。啥意思呢？

你看，假如你有个服务提供一个接口，结果这服务部署在了5台机器上，接着有个接口就是付款接口。然后人家用户在前端上操作的时候，不知道为啥，总之就是一个订单不小心发起了两次支付请求，然后这俩请求分散在了这个服务部署的不同的机器上，好了，结果一个订单扣款扣两次？尴尬了。。。

或者是订单系统调用支付系统进行支付，结果不小心因为网络超时了，然后订单系统走了前面我们看到的那个重试机制，咔嚓给你重试了一把，好，支付系统收到一个支付请求两次，而且因为负载均衡算法落在了不同的机器上，尴尬了。。。

所以你肯定得知道这事儿，否则你做出来的分布式系统恐怕容易埋坑

网络问题很常见，100次请求，都ok；1万次，可能1次是超时会重试；10万，10次；100万，100次；如果有100个请求重复了，你没处理，导致订单扣款2次，100个订单都扣错了；每天被100个用户投诉；一个月被3000个用户投诉

我们之前生产就遇到过，是往数据库里写入数据，重复的请求，就导致我们的数据经常会错，出现一些重复数据，就会导致一些问题

这个不是技术问题，这个没有通用的一个方法，这个是结合业务来看应该如何保证幂等性的，你的经验。

所谓幂等性，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确的，比如不能多扣款，不能多插入一条数据，不能将统计值多加了1。这就是幂等性，不给大家来学术性词语了。

其实保证幂等性主要是三点：

（1）对于每个请求必须有一个唯一的标识，举个例子：订单支付请求，肯定得包含订单id，一个订单id最多支付一次，对吧

（2）每次处理完请求之后，必须有一个记录标识这个请求处理过了，比如说常见的方案是在mysql中记录个状态啥的，比如支付之前记录一条这个订单的支付流水，而且支付流水采

（3）每次接收请求需要进行判断之前是否处理过的逻辑处理，比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，orderId已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用再扣款了。

（4）上面只是给大家举个例子，实际运作过程中，你要结合自己的业务来，比如说用redis用orderId作为唯一键。只有成功插入这个支付流水，才可以执行实际的支付扣款。

要求是支付一个订单，必须插入一条支付流水，order\_id建一个唯一键，unique key

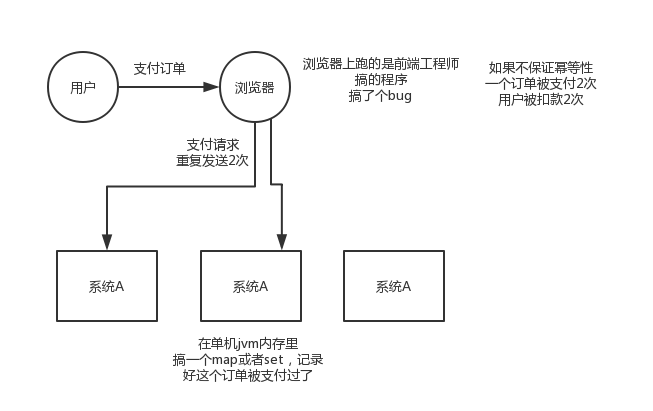
所以你在支付一个订单之前，先插入一条支付流水，order\_id就已经进去了

你就可以写一个标识到redis里面去，set order\_id payed，下一次重复请求过来了，先查redis的order\_id对应的value，如果是payed就说明已经支付过了，你就别重复支付了

然后呢，你再重复支付这个订单的时候，你写尝试插入一条支付流水，数据库给你报错了，说unique key冲突了，整个事务回滚就可以了

来保存一个是否处理过的标识也可以，服务的不同实例可以一起操作redis。

01\_分布式系统接口的幂等性问题



# 分布式系统中的接口调用如何保证顺序性？

分布式服务接口请求的顺序性如何保证？

其实分布式系统接口的调用顺序，也是个问题，一般来说是不用保证顺序的。但是有的时候可能确实是需要严格的顺序保证。给大家举个例子，你服务A调用服务B，先插入再删除。好，结果俩请求过去了，落在不同机器上，可能插入请求因为某些原因执行慢了一些，导致删除请求先执行了，此时因为没数据所以啥效果也没有；结果这个时候插入请求过来了，好，数据插入进去了，那就尴尬了。

本来应该是先插入 -> 再删除，这条数据应该没了，结果现在先删除 -> 再插入，数据还存在，最后你死都想不明白是怎么回事。

所以这都是分布式系统一些很常见的问题

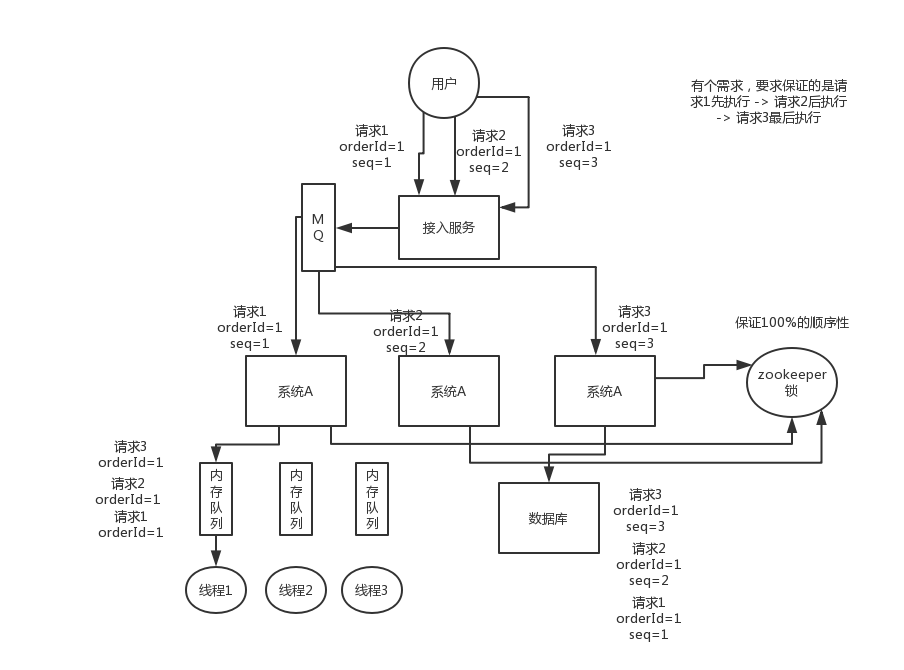
首先，一般来说，我个人给你的建议是，你们从业务逻辑上最好设计的这个系统不需要这种顺序性的保证，因为一旦引入顺序性保障，会导致系统复杂度上升，而且会带来效率低下，热点数据压力过大，等问题。

下面我给个我们用过的方案吧，简单来说，首先你得用dubbo的一致性hash负载均衡策略，将比如某一个订单id对应的请求都给分发到某个机器上去，接着就是在那个机器上因为可能还是多线程并发执行的，你可能得立即将某个订单id对应的请求扔一个内存队列里去，强制排队，这样来确保他们的顺序性。

但是这样引发的后续问题就很多，比如说要是某个订单对应的请求特别多，造成某台机器成热点怎么办？解决这些问题又要开启后续一连串的复杂技术方案。。。曾经这类问题弄的我们头疼不已，所以，还是建议什么呢？

最好是比如说刚才那种，一个订单的插入和删除操作，能不能合并成一个操作，就是一个删除，或者是什么，避免这种问题的产生。

01\_分布式系统接口调用顺序性



# 如何设计一个类似dubbo的rpc框架？架构上该如何考虑？

如何自己设计一个类似dubbo的rpc框架？

说实话，就这问题，其实就跟问你，如何自己设计一个MQ，一样的道理，就考两个：

（1）你有没有对某个rpc框架原理有非常深入的理解

（2）你能不能从整体上来思考一下，如何设计一个rpc框架，考考你的系统设计能力

其实一般问到你这问题，你起码不能认怂，因为既然咱们这个课程是短期的面试突击训练课程，那我不可能给你深入讲解什么kafka源码剖析，dubbo源码剖析，何况我就算讲了，你要真的消化理解和吸收，起码个把月以后了。

所以我给大家一个建议，遇到这类问题，起码从你了解的类似框架的原理入手，自己说说参照dubbo的原理，你来设计一下，举个例子，dubbo不是有那么多分层么？而且每个分层是干啥的，你大概是不是知道？那就按照这个思路大致说一下吧，起码你不能懵逼，要比那些上来就懵，啥也说不出来的人要好一些

举个例子，我给大家说个最简单的回答思路：

（1）上来你的服务就得去注册中心注册吧，你是不是得有个注册中心，保留各个服务的信心，可以用zookeeper来做，对吧

（2）然后你的消费者需要去注册中心拿对应的服务信息吧，对吧，而且每个服务可能会存在于多台机器上

（3）接着你就该发起一次请求了，咋发起？蒙圈了是吧。当然是基于动态代理了，你面向接口获取到一个动态代理，这个动态代理就是接口在本地的一个代理，然后这个代理会找到服务对应的机器地址

（4）然后找哪个机器发送请求？那肯定得有个负载均衡算法了，比如最简单的可以随机轮询是不是

（5）接着找到一台机器，就可以跟他发送请求了，第一个问题咋发送？你可以说用netty了，nio方式；第二个问题发送啥格式数据？你可以说用hessian序列化协议了，或者是别的，对吧。然后请求过去了。。

（6）服务器那边一样的，需要针对你自己的服务生成一个动态代理，监听某个网络端口了，然后代理你本地的服务代码。接收到请求的时候，就调用对应的服务代码，对吧。

这就是一个最最基本的rpc框架的思路，先不说你有多牛逼的技术功底，哪怕这个最简单的思路你先给出来行不行？好，突击课程，那就到这儿结束了，我这课程定位是帮你快速梳理一遍，扫清盲点，不是打通你任督二脉，给你九阳神功的。

### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# 说说zookeeper一般都有哪些使用场景？

### zk都有哪些使用场景？

zk，zookeeper，你们现在在聊的面试topic，是分布式系统，他其实已经跟你聊完了dubbo以及相关的一些问题，确认，你现在分布式服务框架，rpc框架，基本都有一些认知。可能开始要跟你聊分布式相关的其他问题了。

分布式锁这个东西，很常用的，你做java系统开发，分布式系统，可能会有一些场景会用到。最常用的分布式锁就是zookeeper来做分布式锁。

其实说实话，问这个问题，一般就是看看你是否了解zk，因为zk是分布式系统中很常见的一个基础系统。而且问的话常问的就是说zk的使用场景是什么？看你知道不知道一些基本的使用场景。但是其实zk挖深了自然是可以问的很深很深的。

友情提示，如果有某个同学连zookeeper是什么都不知道？听说过，连个基本原理都不知道，连个demo都没写过。我建议你先暂停这个课程，先百度一下，zookeeper入门，最好写helloworld级别的demo程序，体验一把。

大致来说，zk的使用场景如下，我就举几个简单的，大家能说几个就好了：

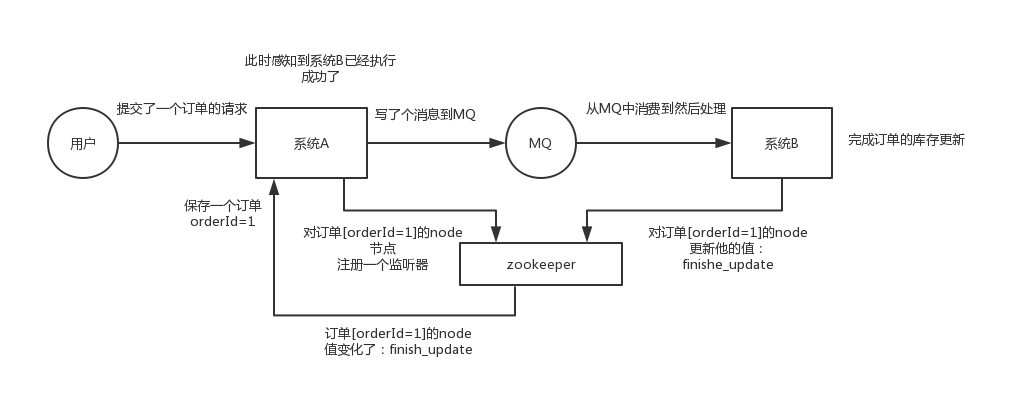
（1）分布式协调：这个其实是zk很经典的一个用法，简单来说，就好比，你A系统发送个请求到mq，然后B消息消费之后处理了。那A系统如何知道B系统的处理结果？用zk就可以实现分布式系统之间的协调工作。A系统发送请求之后可以在zk上对某个节点的值注册个监听器，一旦B系统处理完了就修改zk那个节点的值，A立马就可以收到通知，完美解决。

（2）分布式锁：对某一个数据连续发出两个修改操作，两台机器同时收到了请求，但是只能一台机器先执行另外一个机器再执行。那么此时就可以使用zk分布式锁，一个机器接收到了请求之后先获取zk上的一把分布式锁，就是可以去创建一个znode，接着执行操作；然后另外一个机器也尝试去创建那个znode，结果发现自己创建不了，因为被别人创建了。。。。那只能等着，等第一个机器执行完了自己再执行。

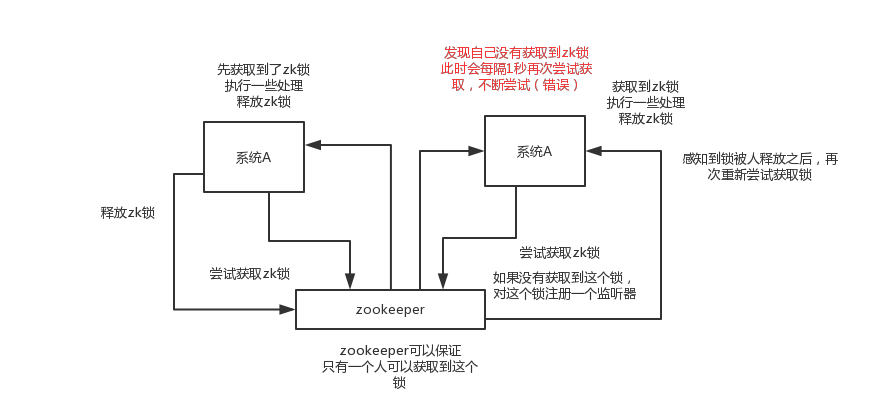
（3）元数据/配置信息管理：zk可以用作很多系统的配置信息的管理，比如kafka、storm等等很多分布式系统都会选用zk来做一些元数据、配置信息的管理，包括dubbo注册中心不也支持zk么

（4）HA高可用性：这个应该是很常见的，比如hadoop、hdfs、yarn等很多大数据系统，都选择基于zk来开发HA高可用机制，就是一个重要进程一般会做主备两个，主进程挂了立马通过zk感知到切换到备用进程

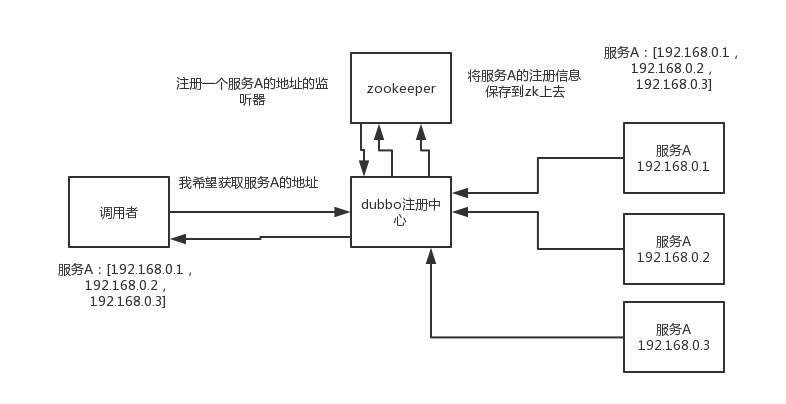
01\_zookeeper的分布式协调场景



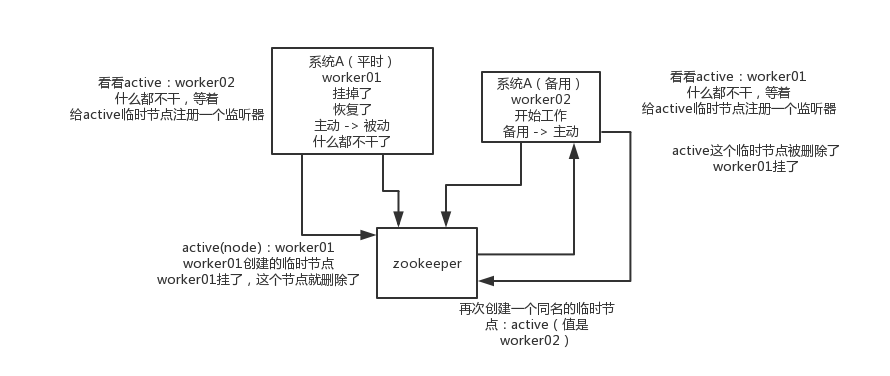
02\_zookeeper的分布式锁场景



03\_zookeeper的元数据\_配置管理场景



04\_zookeeper的HA高可用性场景



# 分布式锁是啥？对比下redis和zk两种分布式锁的优劣？

一般实现分布式锁都有哪些方式？

使用redis如何设计分布式锁？

使用zk来设计分布式锁可以吗？

这两种分布式锁的实现方式哪种效率比较高？

其实一般问问题，都是这么问的，先问问你zk，然后其实是要过度的zk关联的一些问题里去，比如分布式锁。因为在分布式系统开发中，分布式锁的使用场景还是很常见的。

### （1）redis分布式锁

官方叫做RedLock算法，是redis官方支持的分布式锁算法。

这个分布式锁有3个重要的考量点，互斥（只能有一个客户端获取锁），不能死锁，容错（大部分redis节点或者这个锁就可以加可以释放）

第一个最普通的实现方式，如果就是在redis里创建一个key算加锁

SET my:lock 随机值 NX PX 30000，这个命令就ok，这个的NX的意思就是只有key不存在的时候才会设置成功，PX 30000的意思是30秒后锁自动释放。别人创建的时候如果发现已经有了就不能加锁了。

释放锁就是删除key，但是一般可以用lua脚本删除，判断value一样才删除：

关于redis如何执行lua脚本，自行百度

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

为啥要用随机值呢？因为如果某个客户端获取到了锁，但是阻塞了很长时间才执行完，此时可能已经自动释放锁了，此时可能别的客户端已经获取到了这个锁，要是你这个时候直接删除key的话会有问题，所以得用随机值加上面的lua脚本来释放锁。

但是这样是肯定不行的。因为如果是普通的redis单实例，那就是单点故障。或者是redis普通主从，那redis主从异步复制，如果主节点挂了，key还没同步到从节点，此时从节点切换为主节点，别人就会拿到锁。

第二个问题，RedLock算法

这个场景是假设有一个redis cluster，有5个redis master实例。然后执行如下步骤获取一把锁：

1）获取当前时间戳，单位是毫秒

2）跟上面类似，轮流尝试在每个master节点上创建锁，过期时间较短，一般就几十毫秒

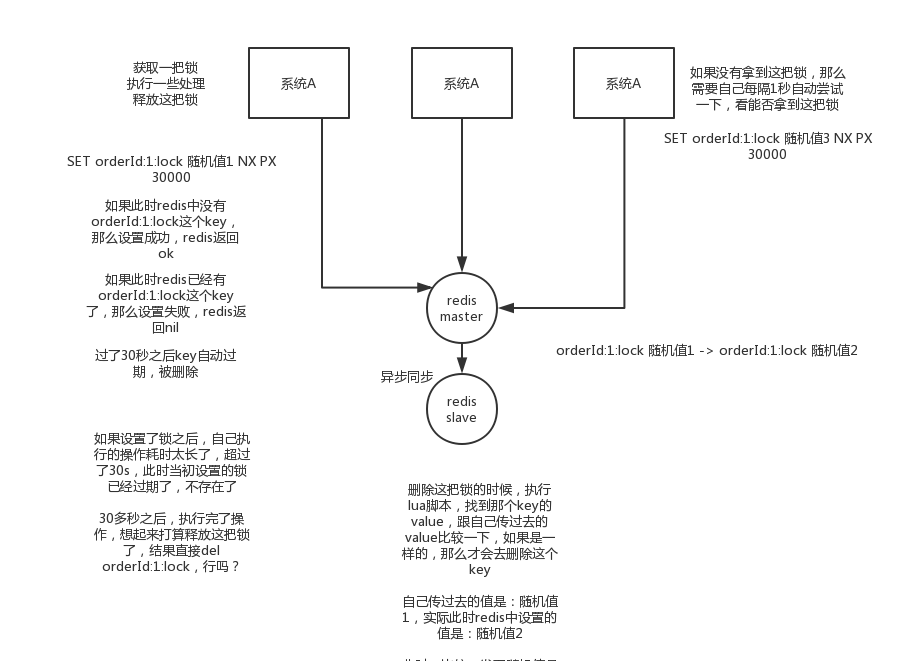
3）尝试在大多数节点上建立一个锁，比如5个节点就要求是3个节点（n / 2 +1）

4）客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

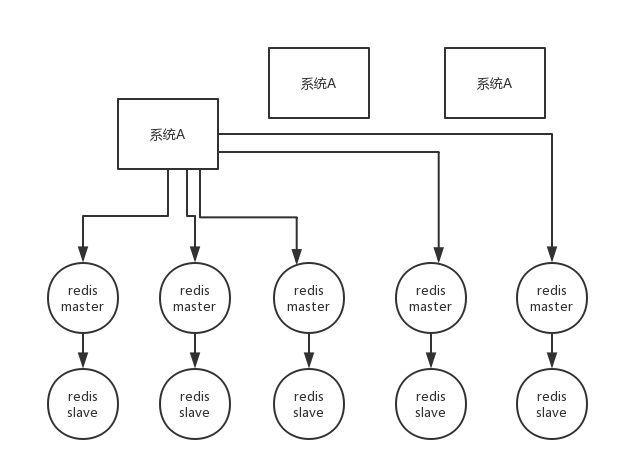
5）要是锁建立失败了，那么就依次删除这个锁

6）只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁

01\_redis最普通的分布式锁的实现原理



02\_RedLock算法



### （2）zk分布式锁

zk分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能注册个监听器监听这个锁。释放锁就是删除这个znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新枷锁。

/\*\*

\* ZooKeeperSession

\* @author Administrator

\*

\*/

public class ZooKeeperSession {

private static CountDownLatch connectedSemaphore = new CountDownLatch(1);

private ZooKeeper zookeeper;

private CountDownLatch latch;

public ZooKeeperSession() {

try {

this.zookeeper = new ZooKeeper(

"192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181",

50000,

new ZooKeeperWatcher());

try {

connectedSemaphore.await();

} catch(InterruptedException e) {

e.printStackTrace();

}

System.out.println("ZooKeeper session established......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 获取分布式锁

\* @param productId

\*/

public Boolean acquireDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch (Exception e) {

while(true) {

try {

Stat stat = zk.exists(path, true); // 相当于是给node注册一个监听器，去看看这个监听器是否存在

if(stat != null) {

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS);

this.latch = null;

}

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch(Exception e) {

continue;

}

}

// 很不优雅，我呢就是给大家来演示这么一个思路

// 比较通用的，我们公司里我们自己封装的基于zookeeper的分布式锁，我们基于zookeeper的临时顺序节点去实现的，比较优雅的

}

return true;

}

/\*\*

\* 释放掉一个分布式锁

\* @param productId

\*/

public void releaseDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.delete(path, -1);

System.out.println("release the lock for product[id=" + productId + "]......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 建立zk session的watcher

\* @author Administrator

\*

\*/

private class ZooKeeperWatcher implements Watcher {

public void process(WatchedEvent event) {

System.out.println("Receive watched event: " + event.getState());

if(KeeperState.SyncConnected == event.getState()) {

connectedSemaphore.countDown();

}

if(this.latch != null) {

this.latch.countDown();

}

}

}

/\*\*

\* 封装单例的静态内部类

\* @author Administrator

\*

\*/

private static class Singleton {

private static ZooKeeperSession instance;

static {

instance = new ZooKeeperSession();

}

public static ZooKeeperSession getInstance() {

return instance;

}

}

/\*\*

\* 获取单例

\* @return

\*/

public static ZooKeeperSession getInstance() {

return Singleton.getInstance();

}

/\*\*

\* 初始化单例的便捷方法

\*/

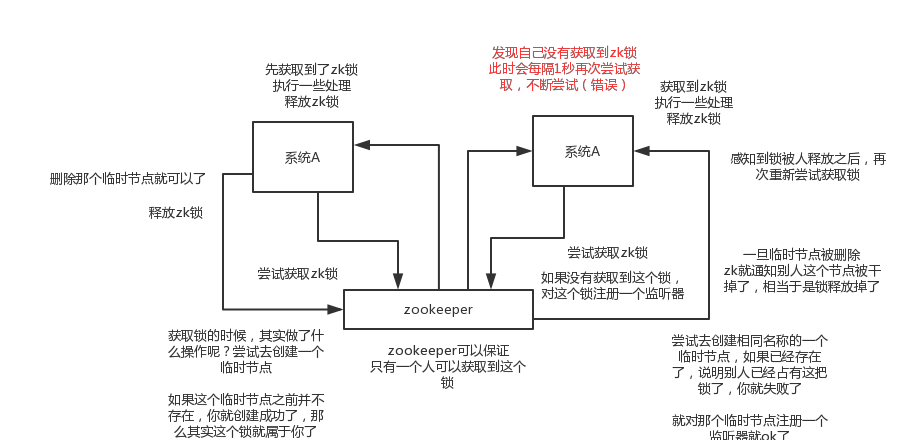
public static void init() {

getInstance();

}

}

03\_zookeeper的分布式锁原理



### （3）redis分布式锁和zk分布式锁的对比

redis分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能

zk分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小

另外一点就是，如果是redis获取锁的那个客户端bug了或者挂了，那么只能等待超时时间之后才能释放锁；而zk的话，因为创建的是临时znode，只要客户端挂了，znode就没了，此时就自动释放锁

redis分布式锁大家每发现好麻烦吗？遍历上锁，计算时间等等。。。zk的分布式锁语义清晰实现简单

所以先不分析太多的东西，就说这两点，我个人实践认为zk的分布式锁比redis的分布式锁牢靠、而且模型简单易用

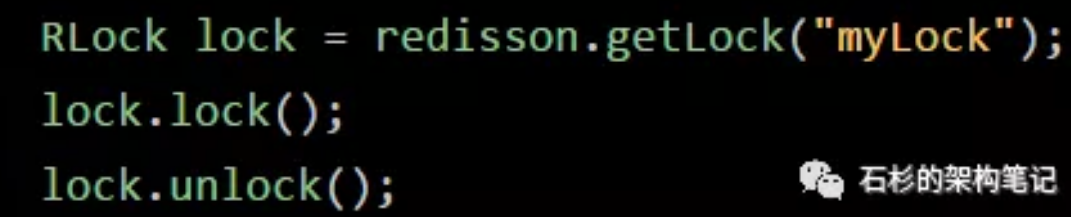
|  |
| --- |
| public class ZooKeeperDistributedLock implements Watcher{    private ZooKeeper zk;  private String locksRoot= "/locks";  private String productId;  private String waitNode;  private String lockNode;  private CountDownLatch latch;  private CountDownLatch connectedLatch = new CountDownLatch(1);  private int sessionTimeout = 30000;  public ZooKeeperDistributedLock(String productId){  this.productId = productId;  try {  String address = "192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181";  zk = new ZooKeeper(address, sessionTimeout, this);  connectedLatch.await();  } catch (IOException e) {  throw new LockException(e);  } catch (KeeperException e) {  throw new LockException(e);  } catch (InterruptedException e) {  throw new LockException(e);  }  }  public void process(WatchedEvent event) {  if(event.getState()==KeeperState.SyncConnected){  connectedLatch.countDown();  return;  }  if(this.latch != null) {  this.latch.countDown();  }  }  public void acquireDistributedLock() {  try {  if(this.tryLock()){  return;  }  else{  waitForLock(waitNode, sessionTimeout);  }  } catch (KeeperException e) {  throw new LockException(e);  } catch (InterruptedException e) {  throw new LockException(e);  }  }  public boolean tryLock() {  try {  // 传入进去的locksRoot + “/” + productId  // 假设productId代表了一个商品id，比如说1  // locksRoot = locks  // /locks/10000000000，/locks/10000000001，/locks/10000000002  lockNode = zk.create(locksRoot + "/" + productId, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);    // 看看刚创建的节点是不是最小的节点  // locks：10000000000，10000000001，10000000002  List<String> locks = zk.getChildren(locksRoot, false);  Collections.sort(locks);    if(lockNode.equals(locksRoot+"/"+ locks.get(0))){  //如果是最小的节点,则表示取得锁  return true;  }    //如果不是最小的节点，找到比自己小1的节点  int previousLockIndex = -1;  for(int i = 0; i < locks.size(); i++) {  if(lockNode.equals(locksRoot + “/” + locks.get(i))) {  previousLockIndex = i - 1;  break;  }  }    this.waitNode = locks.get(previousLockIndex);  } catch (KeeperException e) {  throw new LockException(e);  } catch (InterruptedException e) {  throw new LockException(e);  }  return false;  }    private boolean waitForLock(String waitNode, long waitTime) throws InterruptedException, KeeperException {  Stat stat = zk.exists(locksRoot + "/" + waitNode, true);  if(stat != null){  this.latch = new CountDownLatch(1);  this.latch.await(waitTime, TimeUnit.MILLISECONDS); this.latch = null;  }  return true;  }  public void unlock() {  try {  // 删除/locks/10000000000节点  // 删除/locks/10000000001节点  System.out.println("unlock " + lockNode);  zk.delete(lockNode,-1);  lockNode = null;  zk.close();  } catch (InterruptedException e) {  e.printStackTrace();  } catch (KeeperException e) {  e.printStackTrace();  }  }  public class LockException extends RuntimeException {  private static final long serialVersionUID = 1L;  public LockException(String e){  super(e);  }  public LockException(Exception e){  super(e);  }  }  // 如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁，后面的每个人都会去监听排在自己前面的那个人创建的node上，一旦某个人释放了锁，排在自己后面的人就会被zookeeper给通知，一旦被通知了之后，就ok了，自己就获取到了锁，就可以执行代码了  } |

# Redis分布式锁的实现原理

说实话，如果在公司里落地生产环境用分布式锁的时候，一定是会用开源类库的，比如Redis分布式锁，一般就是用**Redisson**框架就好了，非常的简便易用。

大家如果有兴趣，可以去看看Redisson的官网，看看如何在项目中引入Redisson的依赖，然后基于Redis实现分布式锁的加锁与释放锁。

下面给大家看一段简单的使用代码片段，先直观的感受一下：

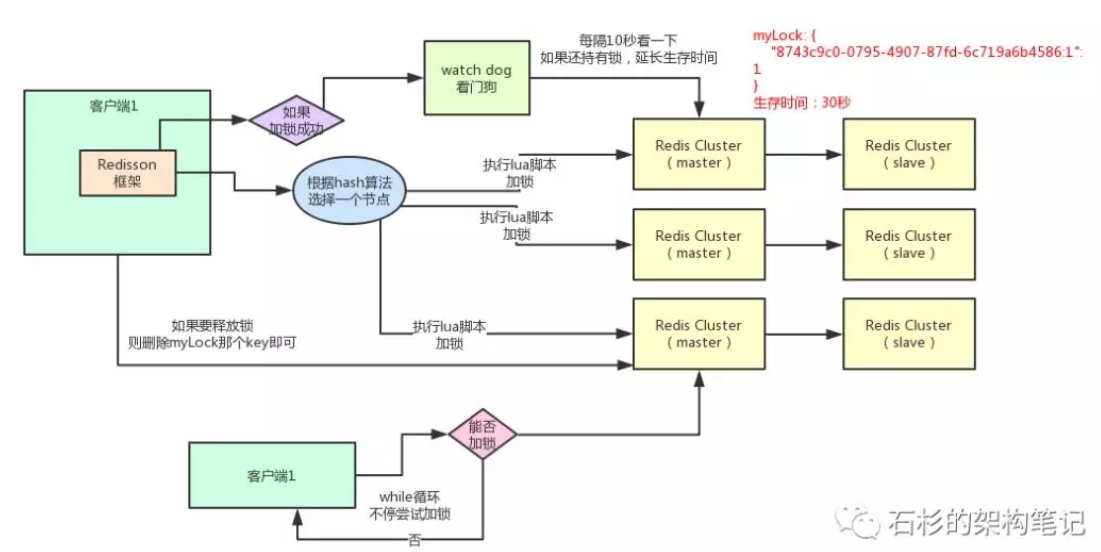


怎么样，上面那段代码，是不是感觉简单的不行！

此外，人家还支持redis单实例、redis哨兵、redis cluster、redis master-slave等各种部署架构，都可以给你完美实现。

**Redisson实现Redis分布式锁的底层原理**

好的，接下来就通过一张手绘图，给大家说说Redisson这个开源框架对Redis分布式锁的实现原理。

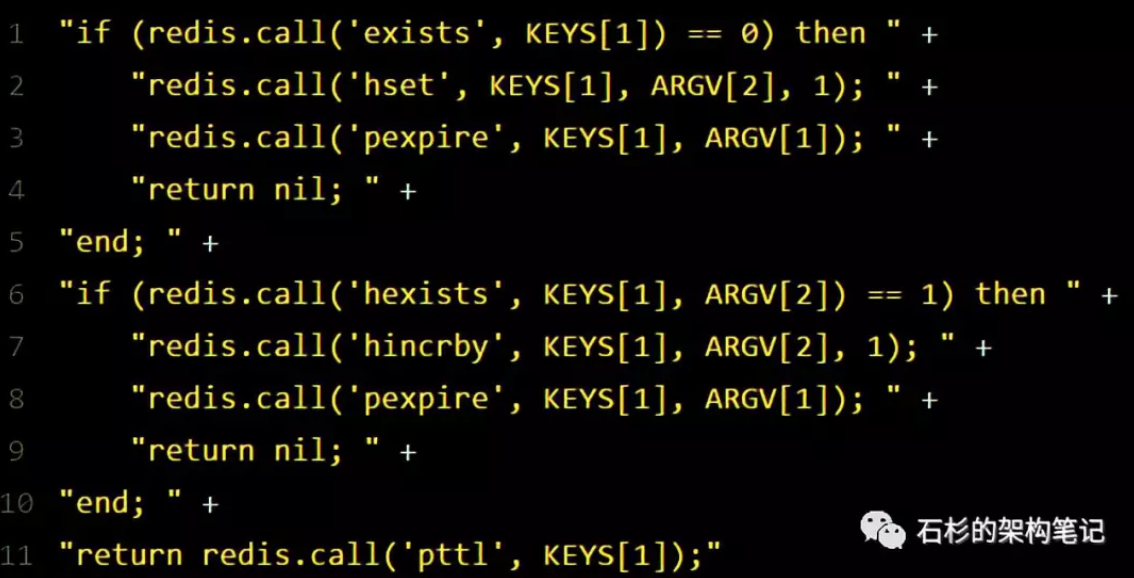


### （1）加锁机制

咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

这里注意，仅仅只是选择一台机器！这点很关键！

紧接着，就会发送一段lua脚本到redis上，那段lua脚本如下所示：



为啥要用lua脚本呢？

因为一大坨复杂的业务逻辑，可以通过封装在lua脚本中发送给redis，保证这段复杂业务逻辑执行的原子性。

那么，这段lua脚本是什么意思呢？

KEYS[1]代表的是你加锁的那个key，比如说：

RLock lock = redisson.getLock("myLock");

这里你自己设置了加锁的那个锁key就是“myLock”。

ARGV[1]代表的就是锁key的默认生存时间，默认30秒。

ARGV[2]代表的是加锁的客户端的ID，类似于下面这样：

8743c9c0-0795-4907-87fd-6c719a6b4586:1

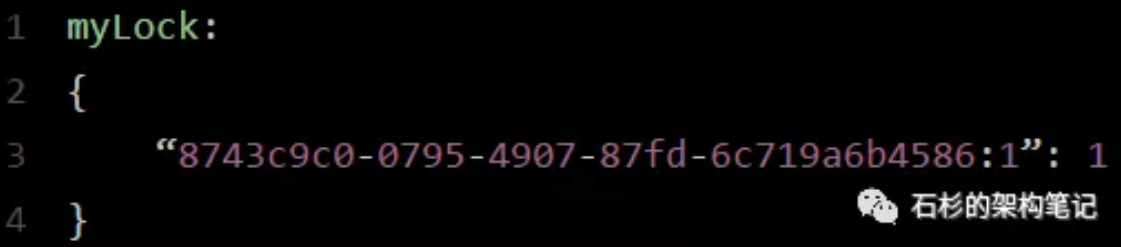
给大家解释一下，第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的命令：

hset myLock

    8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：



上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

好了，到此为止，ok，加锁完成了。

### （2）锁互斥机制

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的**剩余生存时间。**比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

### （3）watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢？

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

### （4）可重入加锁机制

那如果客户端1都已经持有了这把锁了，结果可重入的加锁会怎么样呢？

比如下面这种代码：



这时我们来分析一下上面那段lua脚本。

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

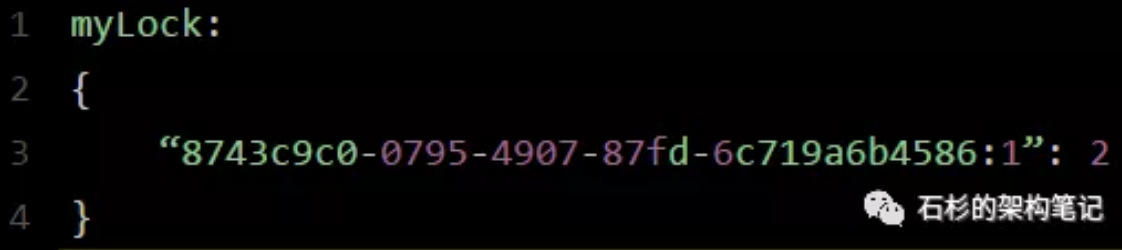
此时就会执行可重入加锁的逻辑，他会用：

incrby myLock

 8743c9c0-0795-4907-87fd-6c71a6b4586:1 1

通过这个命令，对客户端1的加锁次数，累加1。

此时myLock数据结构变为下面这样：



大家看到了吧，那个myLock的hash数据结构中的那个客户端ID，就对应着加锁的次数

### （5）释放锁机制

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

这就是所谓的**分布式锁的开源Redisson框架的实现机制。**

一般我们在生产系统中，可以用Redisson框架提供的这个类库来基于redis进行分布式锁的加锁与释放锁。

### （6）上述Redis分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是redis cluster，或者是redis master-slave架构的**主从异步复制**导致的redis分布式锁的最大缺陷：在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。

# 彻底讲清楚ZooKeeper分布式锁的实现原理

同理，我是直接基于比较常用的**Curator**这个开源框架，聊一下这个框架对ZooKeeper（以下简称zk）分布式锁的实现。

一般除了大公司是自行封装分布式锁框架之外，建议大家用这些开源框架封装好的分布式锁实现，这是一个比较快捷省事儿的方式。

### ZooKeeper分布式锁机制

接下来我们一起来看看，多客户端获取及释放zk分布式锁的整个流程及背后的原理。

首先大家看看下面的图，如果现在有两个客户端一起要争抢zk上的一把分布式锁，会是个什么场景？



如果大家对zk还不太了解，建议先百度一下，快速了解一些基本概念，比如zk有哪些节点类型等等。

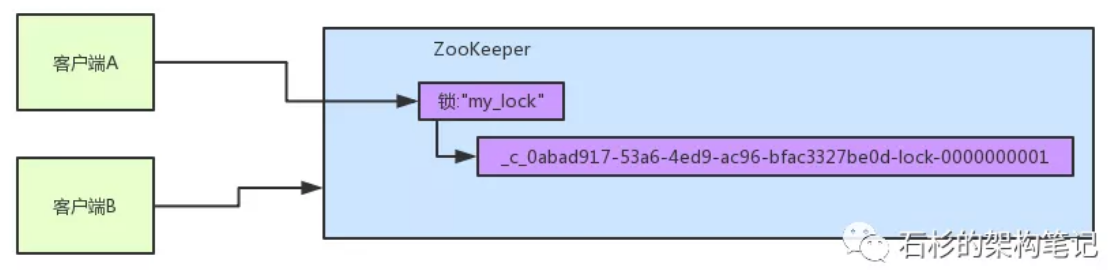
参见上图。zk里有一把锁，这个锁就是zk上的一个节点。然后呢，两个客户端都要来获取这个锁，具体是怎么来获取呢？

咱们就假设客户端A抢先一步，对zk发起了加分布式锁的请求，这个加锁请求是用到了zk中的一个特殊的概念，叫做**“临时顺序节点”。**

简单来说，就是直接在"my\_lock"这个锁节点下，创建一个顺序节点，这个顺序节点有zk内部自行维护的一个节点序号。

比如说，第一个客户端来搞一个顺序节点，zk内部会给起个名字叫做：xxx-000001。然后第二个客户端来搞一个顺序节点，zk可能会起个名字叫做：xxx-000002。大家注意一下，**最后一个数字都是依次递增的**，从1开始逐次递增。zk会维护这个顺序。

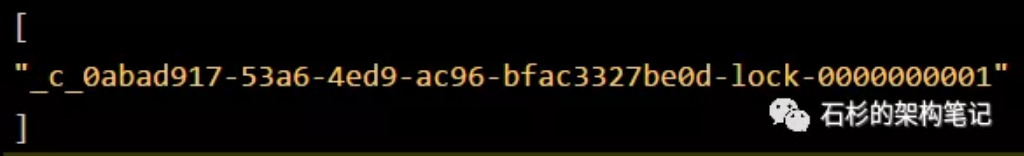
所以这个时候，假如说客户端A先发起请求，就会搞出来一个顺序节点，大家看下面的图，Curator框架大概会弄成如下的样子：



大家看，客户端A发起一个加锁请求，先会在你要加锁的node下搞一个临时顺序节点，这一大坨长长的名字都是Curator框架自己生成出来的。

然后，那个最后一个数字是"1"。大家注意一下，因为客户端A是第一个发起请求的，所以给他搞出来的顺序节点的序号是"1"。

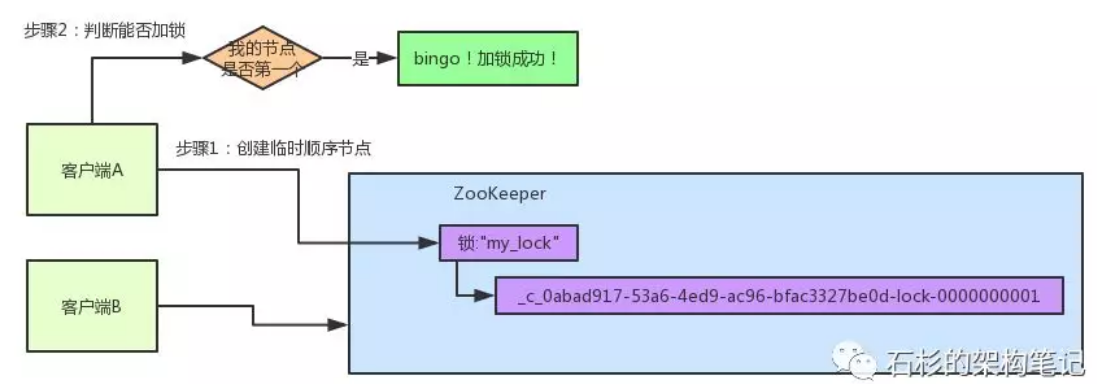
接着客户端A创建完一个顺序节点。还没完，他会查一下"**my\_lock**"这个锁节点下的所有子节点，并且这些子节点是按照序号排序的，这个时候他大概会拿到这么一个集合：



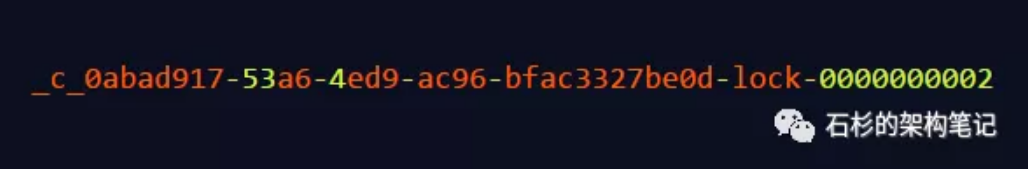
接着客户端A会走一个关键性的判断，就是说：唉！兄弟，这个集合里，我创建的那个顺序节点，是不是排在第一个啊？

如果是的话，那我就可以加锁了啊！因为明明我就是第一个来创建顺序节点的人，所以我就是第一个尝试加分布式锁的人啊！

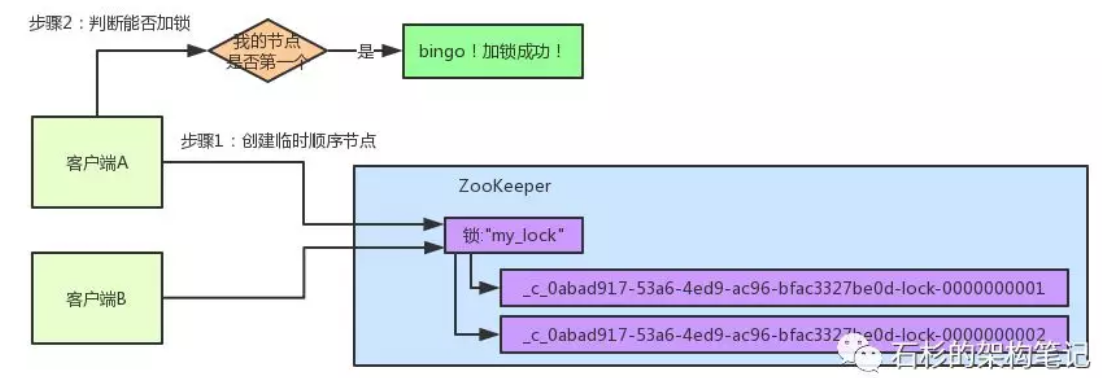
bingo！**加锁成功**！大家看下面的图，再来直观的感受一下整个过程。



接着假如说，客户端A都加完锁了，客户端B过来想要加锁了，这个时候他会干一样的事儿：先是在"**my\_lock**"这个锁节点下创建一个**临时顺序节点**，此时名字会变成类似于：

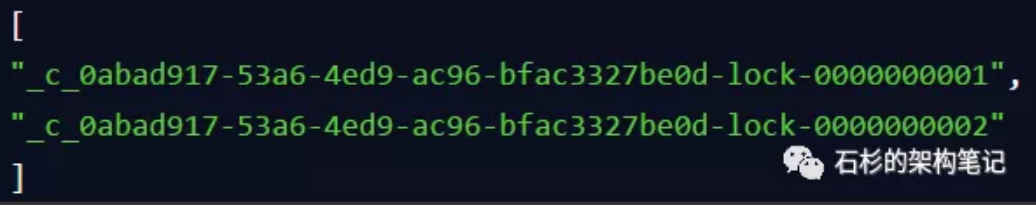


**大家看看下面的图：**



客户端B因为是第二个来创建顺序节点的，所以zk内部会维护序号为"2"。

接着客户端B会走加锁判断逻辑，查询"my\_lock"锁节点下的所有子节点，按序号顺序排列，此时他看到的类似于：

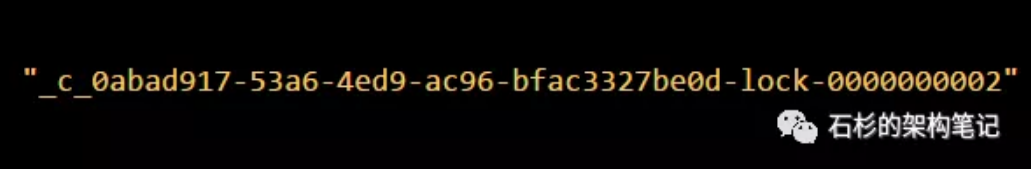


同时检查自己创建的顺序节点，是不是集合中的第一个？

明显不是啊，此时第一个是客户端A创建的那个顺序节点，序号为"01"的那个。**所以加锁失败**！

加锁失败以后，客户端B就会通过ZK的API，**对他的上一个顺序节点加一个监听器。**zk天然就可以实现对某个节点的监听。

我们举例说明，客户端B的顺序节点是：



他的上一个顺序节点，不就是下面这个吗？



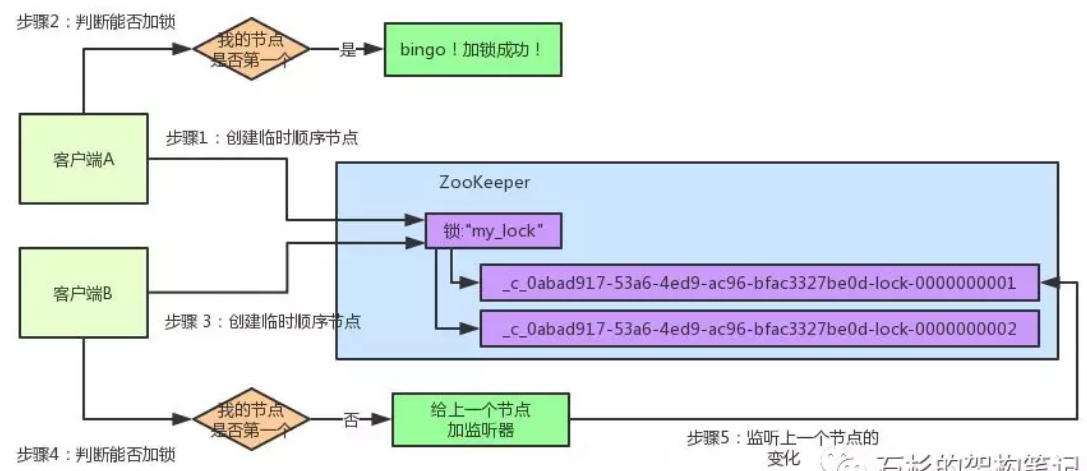
也就是客户端A创建的那个顺序节点！

所以，客户端B会对：



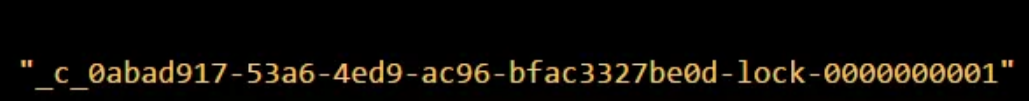
这个节点加一个监听器，监听这个节点是否被删除等变化！

说了那么多，老规矩，给大家来一张图，直观的感受一下：



接着，客户端A加锁之后，可能处理了一些代码逻辑，然后就会释放锁。那么，释放锁是个什么过程呢？

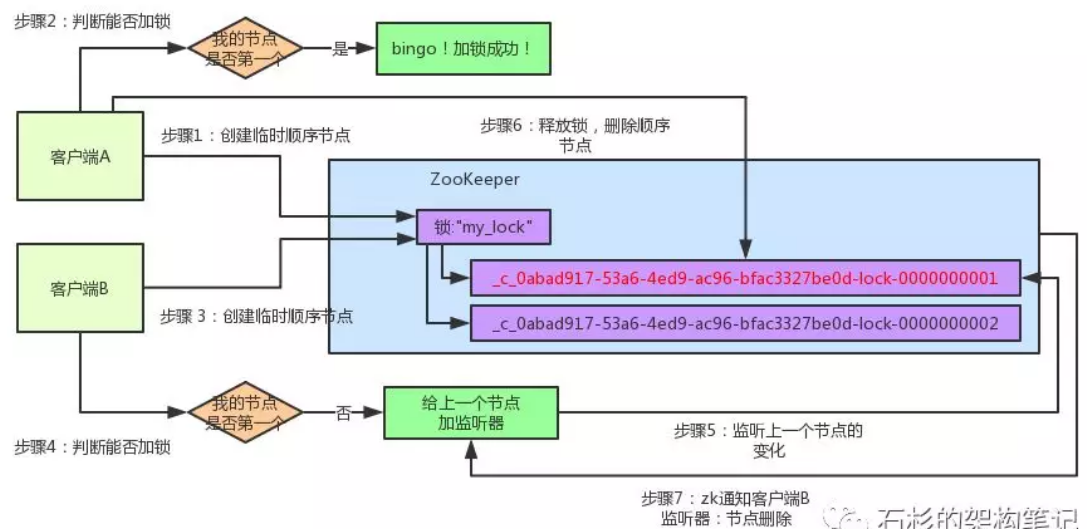
其实很简单，就是把自己在zk里创建的那个顺序节点，也就是：



这个节点给删除。

删除了那个节点之后，zk会负责通知监听这个节点的监听器，也就是客户端B之前加的那个监听器，说：兄弟，你监听的那个节点被删除了，有人释放了锁。

我们一起来看看下面的图，体会一下这个过程：



此时客户端B的监听器感知到了上一个顺序节点被删除，也就是排在他之前的某个客户端释放了锁。

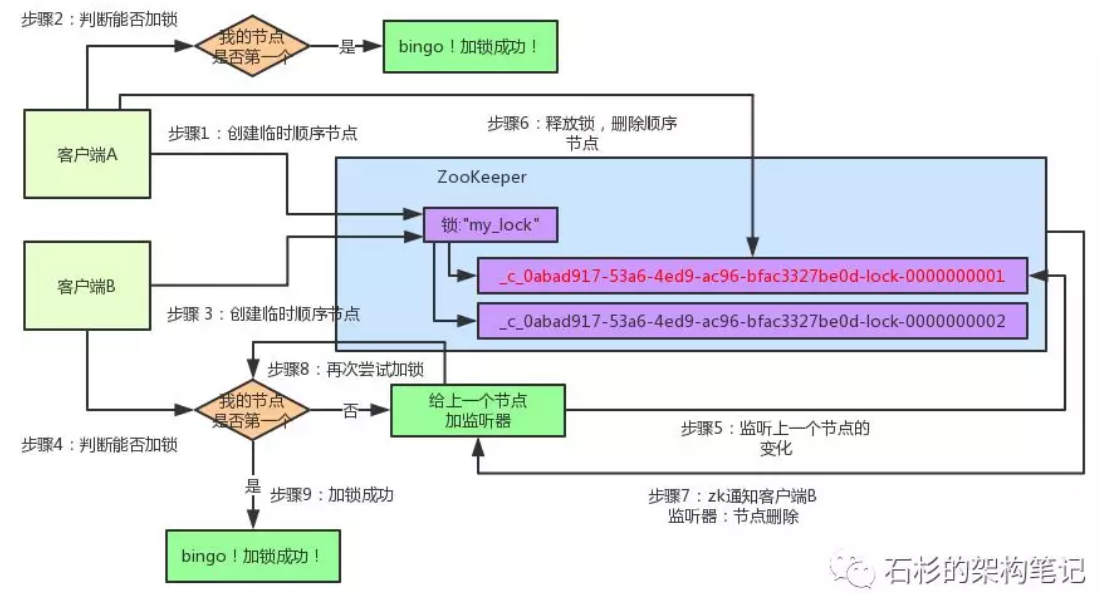
此时，就会通知客户端B重新尝试去获取锁，也就是获取"my\_lock"节点下的子节点集合，此时为：



集合里此时只有客户端B创建的唯一的一个顺序节点了！

然后呢，客户端B一判断，发现自己居然是集合中的第一个顺序节点，bingo！可以加锁了！直接完成加锁，运行后续的业务代码即可，运行完了之后再次释放锁。

最后，再给大家来一张图，大伙儿顺着图仔细的捋一捋这整个过程：



其实如果有客户端C、客户端D等N个客户端争抢一个zk分布式锁，原理都是类似的。

* 大家都是上来直接创建一个锁节点下的一个接一个的临时顺序节点
* 如果自己不是第一个节点，就对自己上一个节点加监听器
* 只要上一个节点释放锁，自己就排到前面去了，相当于是一个排队机制。

而且用临时顺序节点的另外一个用意就是，如果某个客户端创建临时顺序节点之后，不小心自己宕机了也没关系，zk感知到那个客户端宕机，会自动删除对应的临时顺序节点，相当于自动释放锁，或者是自动取消自己的排队。

最后，咱们来看下用Curator框架进行加锁和释放锁的一个过程：



其实用开源框架就是这点好，方便。这个Curator框架的zk分布式锁的加锁和释放锁的实现原理，其实就是上面我们说的那样子。

但是如果你要手动实现一套那个代码的话。还是有点麻烦的，要考虑到各种细节，异常处理等等。所以大家如果考虑用zk分布式锁，可以参考下本文的思路。

# 说说你们的分布式session方案是啥？怎么做的？

集群部署时的分布式session如何实现？

面试官问了你一堆dubbo是怎么玩儿的，你会玩儿dubbo就可以把单块系统弄成分布式系统，然后分布式之后接踵而来的就是一堆问题，最大的问题就是分布式事务、接口幂等性、分布式锁，还有最后一个就是分布式session。

当然了，分布式系统中的问题何止这么一点，非常之多，复杂度很高，但是这里就是说下常见的几个，也是面试的时候常问的几个。

session是啥？浏览器有个cookie，在一段时间内这个cookie都存在，然后每次发请求过来都带上一个特殊的jsessionid cookie，就根据这个东西，在服务端可以维护一个对应的session域，里面可以放点儿数据。

一般只要你没关掉浏览器，cookie还在，那么对应的那个session就在，但是cookie没了，session就没了。常见于什么购物车之类的东西，还有登录状态保存之类的。

这个不多说了，懂java的都该知道这个。

但是你单块系统的时候这么玩儿session没问题啊，但是你要是分布式系统了呢，那么多的服务，session状态在哪儿维护啊？

其实方法很多，但是常见常用的是两种：

### （1）tomcat + redis

这个其实还挺方便的，就是使用session的代码跟以前一样，还是基于tomcat原生的session支持即可，然后就是用一个叫做Tomcat RedisSessionManager的东西，让所有我们部署的tomcat都将session数据存储到redis即可。

在tomcat的配置文件中，配置一下

<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHandlerValve" />

<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionManager"

host="{redis.host}"

port="{redis.port}"

database="{redis.dbnum}"

maxInactiveInterval="60"/>

搞一个类似上面的配置即可，你看是不是就是用了RedisSessionManager，然后指定了redis的host和 port就ok了。

<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHandlerValve" />

<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionManager"

sentinelMaster="mymaster"

sentinels="<sentinel1-ip>:26379,<sentinel2-ip>:26379,<sentinel3-ip>:26379"

maxInactiveInterval="60"/>

还可以用上面这种方式基于redis哨兵支持的redis高可用集群来保存session数据，都是ok的

### （2）spring session + redis

分布式会话的这个东西重耦合在tomcat中，如果我要将web容器迁移成jetty，难道你重新把jetty都配置一遍吗？

因为上面那种tomcat + redis的方式好用，但是会严重依赖于web容器，不好将代码移植到其他web容器上去，尤其是你要是换了技术栈咋整？比如换成了spring cloud或者是spring boot之类的。还得好好思忖思忖。

所以现在比较好的还是基于java一站式解决方案，spring了。人家spring基本上包掉了大部分的我们需要使用的框架了，spirng cloud做微服务了，spring boot做脚手架了，所以用sping session是一个很好的选择。

pom.xml

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session-data-redis</artifactId>

<version>1.2.1.RELEASE</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.8.1</version>

</dependency>

spring配置文件中

<bean id="redisHttpSessionConfiguration"

class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration">

<property name="maxInactiveIntervalInSeconds" value="600"/>

</bean>

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">

<property name="maxTotal" value="100" />

<property name="maxIdle" value="10" />

</bean>

<bean id="jedisConnectionFactory"

class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" destroy-method="destroy">

<property name="hostName" value="${redis\_hostname}"/>

<property name="port" value="${redis\_port}"/>

<property name="password" value="${redis\_pwd}" />

<property name="timeout" value="3000"/>

<property name="usePool" value="true"/>

<property name="poolConfig" ref="jedisPoolConfig"/>

</bean>

web.xml

<filter>

<filter-name>springSessionRepositoryFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSessionRepositoryFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

示例代码

@Controller

@RequestMapping("/test")

public class TestController {

@RequestMapping("/putIntoSession")

@ResponseBody

public String putIntoSession(HttpServletRequest request, String username){

request.getSession().setAttribute("name", “leo”);

return "ok";

}

@RequestMapping("/getFromSession")

@ResponseBody

public String getFromSession(HttpServletRequest request, Model model){

String name = request.getSession().getAttribute("name");

return name;

}

}

上面的代码就是ok的，给spring session配置基于redis来存储session数据，然后配置了一个spring session的过滤器，这样的话，session相关操作都会交给spring session来管了。接着在代码中，就用原生的session操作，就是直接基于spring sesion从redis中获取数据了。

实现分布式的会话，有很多种很多种方式，我说的只不过比较常见的两种方式，tomcat + redis早期比较常用；近些年，重耦合到tomcat中去，通过spring session来实现。

01\_分布式会话是什么

