因为java面试题是一个较为大的体系，所以我们会拆分为几季来讲解，预计是会推出3季，本次是第一季，第一季课程包含的内容如下：

# 1、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？

## 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？

# 2、高并发架构

## 2.1 如何设计一个高并发系统？

## 2.2 消息队列

（1）为什么使用消息队列啊？消息队列有什么优点和缺点啊？kafka、activemq、rabbitmq、rocketmq都有什么优点和缺点啊？

（2）如何保证消息队列的高可用啊？

（3）如何保证消息不被重复消费啊（如何进行消息队列的幂等性问题）？

（4）如何保证消息的可靠性传输（如何处理消息丢失的问题）？

（5）如何保证消息的顺序性？

（6）如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

（7）如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路

## 2.3 搜索引擎

（1）es的分布式架构原理能说一下么（es是如何实现分布式的啊）？

（2）es写入数据的工作原理是什么啊？es查询数据的工作原理是什么啊？底层的lucene介绍一下呗？倒排索引了解吗？

（3）es在数据量很大的情况下（数十亿级别）如何提高查询效率啊？

（4）es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

## 2.4 缓存

（1）在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？

（2）redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多？

（3）redis都有哪些数据类型？分别在哪些场景下使用比较合适？

（5）redis的过期策略都有哪些？手写一下LRU代码实现？

（6）如何保证Redis高并发、高可用、持久化？redis的主从复制原理能介绍一下么？redis的哨兵原理能介绍一下么？

（7）redis的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

（8）redis集群模式的工作原理能说一下么？在集群模式下，redis的key是如何寻址的？分布式寻址都有哪些算法？了解一致性hash算法吗？如何动态增加和删除一个节点？

（9）了解什么是redis的雪崩和穿透？redis崩溃之后会怎么样？系统该如何应对这种情况？如何处理redis的穿透？

（10）如何保证缓存与数据库的双写一致性？

（11）redis的并发竞争问题是什么？如何解决这个问题？了解Redis事务的CAS方案吗？

（12）生产环境中的redis是怎么部署的？

## 2.5 分库分表

（2）为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

（3）现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

（4）如何设计可以动态扩容缩容的分库分表方案？

（5）分库分表之后，id主键如何处理？

## 2.6 读写分离

（1）如何实现mysql的读写分离？MySQL主从复制原理的是啥？如何解决mysql主从同步的延时问题？

# 3、高可用架构

## 3.1 如何设计一个高可用系统？

## 3.2 限流

1. 如何限流？在工作中是怎么做的？说一下具体的实现？

## 3.3 熔断

（1）如何进行熔断？熔断框架都有哪些？具体实现原理知道吗？

## 3.4 降级

（1）如何进行降级？

Day3

这个课程针对的人群

（1）好多同学出去面试，尤其是面互联网行业的Java职位，经常会问一些消息队列、缓存、分库分表、分布式系统之类的问题，但是可能不少同学之前没有搞过这些，或者略知一二，但是被面试官几个连环炮就给打蒙了

（2）或者是不少同学很长时间不面试了，最近要找工作，需要短期突击快速准备一下常见的面试题，至少先给自己热个身，不至于面试的时候一问三不知

（3）或者是不面试，但是对这套技术体系不怎么会的同学，可以用这个课程算是扫盲，快速夯实一下互联网公司技术栈的一些基础

（4）如果你看一眼，觉得自己基本都会，那么可以绕行，这个课程应该不适合你，可以加我助理QQ，未来我会出一些高级的免费课程

这个课程的定位：

（1）两周让你快速突击完整的互联网java工程师面试中要掌握的技术体系

（2）针对的是互联网行业的普通java岗位常问的一些面试题，不是针对一些高级、资深、专家等高阶岗位的面试，因为你要是面那些高阶岗位，靠两三周突击是不可能的

（3）所以这里很多问题以及问题的解答，基本可能都是属于科普性的，有一些技术实力比较好的同学，不要看到后面来骂我，说讲的啥，这么简单，都是些入门和基础，确实是，我也觉得这些问题都很简单，但是确实大量的java同学面试被问到可能都答不出来，所以这个课程是有针对人群的。觉得这个课程讲解内容过于简单的同学，请绕道，这个课程就别看了，但是欢迎关注我们的公众号，后面我们可能还会推出更加高级的一些免费课程。

（4）讲解方式走短平快路线，大白话+手工画图，语言精简不拖拉，不说废话，每个问题都直击问题要点即可。课程出发点是帮助那些想要在2周之内完成面试准备，出去找工作的那帮同学

（5）对每个面试题都给你先剖析一下面试官问这个面试题背后的愿意是什么？他为什么要问你这个问题？

（6）能解答每个问题背后的基本原理，至少被问到这些问题别一无所知，自己能回答出来一些原理以及技术方案，代表至少你有这个知识积累

（7）能在面试现场动手画图讲解

（8）能对常见的要求写代码的环节现场手写代码

（9）试想，面试考察范围那么大，如果你每个技术都想要深入学习达到精通，上手能用，能做项目，那你任何一个技术都要学习很长时间。但是面试的时候可不是那么问的，不会问你api和具体操作步骤，主要考察思想、原理以及经验。所以这里我们为了面试准备，就是对每个技术讲解面试的时候常问的一些问题。

能达到这个标准，那么可以短期内快速提升你对常见面试题的应答能力，让很多同学也许在一些互联网java面试时，也许原本很多题目都几乎说不出什么回答，到现在学完这个课程之后可以基本回答常见的技术体系内的面试题

这个课程，主要设计思想就是针对有面试需求的同学，短期内，短平快的突击一下，大幅度增强你在面试现场的一个表现，增加你通过面试的机会，让你跟很多普通的工程师在面试时的表现迅速拉开差距

所以针对这个定位，我进行几点说明：

1、面试题剖析如何回答时，我会剔除掉很多不必要的细节，就突出很重点。比如有的技术，网上一些博客也许会零零散散写几十点，但是面试的时候你不是背书，不是背博客来回答的，就是讲出来几个重点就可以了。所以有些东西也许我没讲，但是大家要明白不是我不讲，是因为我们要挑重点了解，在面试的时候挑重点来说。

2、对大部分技术点，不会动手去落地实践，比如说RabbitMQ的高可用性，我们就是剖析MQ高可用部署的一个原理，现场大量手工绘图讲清楚如何回答这个东西，确保你被问到的时候，能说清楚整个高可用部署架构以及现场动手画图。但是我不会带着你去安装虚拟机，然后安装rabbitmq，然后再部署高可用架构，然后再进行高可用容灾演练。因为我们的定位是短期内快随突击面试题，转为面试准备，不是动手项目实战课程。

3、对很多技术，我们的定位就是面试被问到的时候，你可以回答出其基本原理，我们不可能对有些技术点刨根问底，深挖到源码级别，那是需要大量时间去讲解的，不是所谓的两三周短平快可以突击的完的。所以确实有可能对于一些要求很高的公司，是可能存在一个问题，人家问你一个问题，你答出来了，接着深挖，你答不出来了，那么此时是没办法的，你要想对各种技术都有深入底层的功底，那不是两三周的事儿，而是两三年的苦功夫才能得到的。

4、有些特别简单的问题，明显只要是个java工程师，学过java的都该知道的问题，我就不讲了。比如说equals和hashcode是啥关系？string类为啥是final的？这类问题，实在是太过基础。

所以特别用几分钟时间跟大家说明白这个问题，大家了解一下这个课程定位，后面学习的时候相信就不会产生什么歧义、误会以及不满了。

最后一点说明，这个课程会作为“石杉码农学院”一个长期对外开放的免费课程，大家在学习这个课程之后，去外面面试，如果遇到这个课程里没有涵盖掉的问题，然后你觉得是比较难的比较典型的问题，可以通过微信公众号给我们留言，我们会定期免费持续更新这个课程的，持续解答更多的面试题

同时我们这里还会定期发布一些同学给我们反馈的互联网公司的java面试经历，我们会整理成文章后发布出来，供大家参考

Day4

面试官：你好

候选人：你好

大家寒暄一下。。。

（面试官在你的简历上面看到了，呦，有个亮点，就是你在项目里用过MQ，比如说你用过ActiveMQ）

面试官：你在系统里用过消息队列吗？（面试官在随和的语气中展开了面试）

候选人：用过的（此时感觉没啥）

面试官：那你说一下你们在项目里是怎么用消息队列的？

候选人：巴拉巴拉，我们啥啥系统发送个啥啥消息到队列，别的系统来消费啥啥的（很多同学在这里会进入一个误区，就是你仅仅就是知道以及回答你们是怎么用这个消息队列的，用这个消息队列来干了个什么事情？）

比如我们有个订单系统，订单系统会每次下一个新的订单的时候，就会发送时一条消息到ActiveMQ里面去，后台有个库存系统负责获取了消息然后更新库存。

面试官：那你们为什么使用消息队列啊？

（你的订单系统不发送消息到MQ，直接订单系统调用库存系统一个接口，咔嚓一下，直接就调用成功能了，库存就更新了）

候选人：额。。。（楞了一下，为什么？我没怎么仔细想过啊，老大让用就用了），硬着头皮胡言乱语了几句

（面试官此时听你楞了一下，然后听你胡言乱语了几句，开始心里觉得有点儿那什么了，怀疑你之前就压根儿没思考过这问题）

面试官：那你说说用消息队列都有什么优点和缺点？

（面试官此时心里想的是，你的MQ在项目里为啥要用？你没考虑过，那我稍微简单点儿，我问问你消息队列你之前有没有考虑过如果用的话，优点和缺点分别是啥？）

候选人：这个。。。（确实平时没怎么考虑过这个问题啊。。。胡言乱语了）

（面试官此时心里已经更觉得你这哥儿们不行，平时都没什么思考）

面试官：kafka、activemq、rabbitmq、rocketmq都有什么区别？

（面试官问你这个问题，就是说，绕过比较虚的话题，直接看看你对各种MQ中间件是否了解，是否做过功课，是否做过调研）

候选人：我们就用过activemq，所以别的没用过。。。区别，也不太清楚

（面试官此时却是觉得你这哥儿们平时就是瞎用，根本就没什么思考，觉得不行）

面试官：那你们是如何保证消息队列的高可用啊？

候选人：这个。。。我平时就是简单走api调用一下，不太清楚消息队列怎么部署的。。。

面试官：如何保证消息不被重复消费啊？如何保证消费的时候是幂等的啊？

候选人：啥？（mq不就是写入和消费就可以了，哪来这么多问题）

面试官：如何保证消息的可靠性传输啊？要是消息丢失了怎么办啊？

候选人：我们没怎么丢过消息啊。。。

面试官：那如何保证消息的顺序性？

候选人：顺序性？什么意思？我为什么要保证消息的顺序性？

面试官：如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

候选人：不是，我这平时没遇到过这些问题啊，就是简单用用，知道mq的一些功能

面试官：如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路

候选人：。。。。。我还是走吧。。。。

我个人的意见，包括我常年在BAT公司里面干了很多年，各种同学都面过，一种比较类似的一种面试风格。就是一般我们不是发散的，我们是从点，铺开，比如说我们会可能跟你聊聊高并发话题，就这个话题里面跟你聊聊缓存、MQ等等东西吧。

对于每个小话题，比如说MQ，我们会从浅入深，这个上面的这个面试套路其实是蛮典型的，如果你确实说自己会MQ，你出去面试，去一些大公司面试，有些面试官可能就是这种风格，就是一步一步深挖

其实上面是一个非常典型的关于消息队列的技术考察过程，好的面试官一定是从你做过的某一个点切入，然后层层展开深入考察，一个接一个问，直到把这个技术点刨根问底，问到最底层。

如果没有刻意的对这种面试方式锻炼一下，出去面试碰到难一点的面试，大多会手忙脚乱，基本面试以失败为告终。

但是如果你把这些常见问题都掌握了，哪怕是面试官没问到你这么深入，他问你一个消息队列问题，你就自己给他说出自己的一整套见解，那么恭喜你，就是plus加分项了

Day5

1、面试题

为什么使用消息队列啊？消息队列有什么优点和缺点啊？kafka、activemq、rabbitmq、rocketmq都有什么区别以及适合哪些场景？

2、面试官心理分析

其实面试官主要是想看看：

（1）第一，你知道不知道你们系统里为什么要用消息队列这个东西？

我之前面试就见过大量的候选人，说自己项目里用了redis、mq，但是其实他并不知道自己为什么要用这个东西。其实说白了，就是为了用而用，或者是别人设计的架构，他从头到尾没思考过。

没有对自己的架构问过为什么的人，一定是平时没有思考的人，面试官对这类候选人印象通常很不好。因为进了团队担心你就木头木脑的干呆活儿，不会自己思考。

（2）第二，你既然用了消息队列这个东西，你知道不知道用了有什么好处？

系统中引入消息队列之后会不会有什么坏处？你要是没考虑过这个，那你盲目弄个MQ进系统里，后面出了问题你是不是就自己溜了给公司留坑？你要是没考虑过引入一个技术可能存在的弊端和风险，面试官把这类候选人招进来了，基本可能就是挖坑型选手。

就怕你干1年挖一堆坑，自己跳槽了，给公司留下后患无穷

（3）第三，既然你用了MQ，可能是某一种MQ，那么你当时做没做过调研啊？

你别傻乎乎的自己拍脑袋看个人喜好就瞎用了一个MQ，比如kafka。甚至都从没调研过业界到底流行的MQ有哪几种？每一个MQ的优点和缺点是什么？每一个MQ没有绝对的好坏，但是就是看用在哪个场景可以扬长避短，利用其优势，规避其劣势。

如果是一个不考虑技术选型的候选人招进了团队，面试官交给他一个任务，去设计个什么系统，他在里面用一些技术，可能都没考虑过选型，最后选的技术可能并不一定合适，一样是留坑

3、额外的友情提示

同学啊，如果你看到这里，连activemq、rabbitmq、rocketmq、kafka是什么都不知道？连个hello world demo都没写过？那你。。。

通过网上查阅技术资料和博客，用于快速入门，是比较合适的，但是如果要比如系统梳理你的面试技术体系，或者是系统的深入的研究和学习一些东西，看博客实际上是不太合适的

那也没事，我们这个课程的定位是不会去讲这些的，建议你马上暂停一下课程，然后上百度搜一下，这4个东西是什么？每个东西找一个教你hello world的博客，自己跟着做一遍。我保证你1个小时之内就可以快速入门这几个东西。

等你先知道这几个东西是什么，同时写过hello world之后，你再来继续看我们的课程

4、面试题剖析

（1）为什么使用消息队列啊？

其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么

面试官问你这个问题，期望的一个回答是说，你们公司有个什么业务场景，这个业务场景有个什么技术挑战，如果不用MQ可能会很麻烦，但是你现在用了MQ之后带给了你很多的好处

先说一下消息队列的常见使用场景吧，其实场景有很多，但是比较核心的有3个：解耦、异步、削峰

解耦：现场画个图来说明一下，A系统发送个数据到BCD三个系统，接口调用发送，那如果E系统也要这个数据呢？那如果C系统现在不需要了呢？现在A系统又要发送第二种数据了呢？A系统负责人濒临崩溃中。。。再来点更加崩溃的事儿，A系统要时时刻刻考虑BCDE四个系统如果挂了咋办？我要不要重发？我要不要把消息存起来？头发都白了啊。。。

面试技巧：你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用MQ给他异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个MQ去进行系统的解耦。在简历中体现出来这块东西，用MQ作解耦。

异步：现场画个图来说明一下，A系统接收一个请求，需要在自己本地写库，还需要在BCD三个系统写库，自己本地写库要3ms，BCD三个系统分别写库要300ms、450ms、200ms。最终请求总延时是3 + 300 + 450 + 200 = 953ms，接近1s，用户感觉搞个什么东西，慢死了慢死了。

削峰：每天0点到11点，A系统风平浪静，每秒并发请求数量就100个。结果每次一到11点~1点，每秒并发请求数量突然会暴增到1万条。但是系统最大的处理能力就只能是每秒钟处理1000个请求啊。。。尴尬了，系统会死。。。

（2）消息队列有什么优点和缺点啊？

优点上面已经说了，就是在特殊场景下有其对应的好处，解耦、异步、削峰

缺点呢？显而易见的

系统可用性降低：系统引入的外部依赖越多，越容易挂掉，本来你就是A系统调用BCD三个系统的接口就好了，人ABCD四个系统好好的，没啥问题，你偏加个MQ进来，万一MQ挂了咋整？MQ挂了，整套系统崩溃了，你不就完了么。

系统复杂性提高：硬生生加个MQ进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已

一致性问题：A系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是BCD三个系统那里，BD两个系统写库成功了，结果C系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，最好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了10倍。但是关键时刻，用，还是得用的。。。

（3）kafka、activemq、rabbitmq、rocketmq都有什么优点和缺点啊？

常见的MQ其实就这几种，别的还有很多其他MQ，但是比较冷门的，那么就别多说了

作为一个码农，你起码得知道各种mq的优点和缺点吧，咱们来画个表格看看

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | ActiveMQ | RabbitMQ | RocketMQ | Kafka |
| 单机吞吐量 | 万级，吞吐量比RocketMQ和Kafka要低了一个数量级 | 万级，吞吐量比RocketMQ和Kafka要低了一个数量级 | 10万级，RocketMQ也是可以支撑高吞吐的一种MQ | 10万级别，这是kafka最大的优点，就是吞吐量高。  一般配合大数据类的系统来进行实时数据计算、日志采集等场景 |
| topic数量对吞吐量的影响 |  |  | 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  这是RocketMQ的一大优势，在同等机器下，可以支撑大量的topic | 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 | 微秒级，这是rabbitmq的一大特点，延迟是最低的 | ms级 | 延迟在ms级以内 |
| 可用性 | 高，基于主从架构实现高可用性 | 高，基于主从架构实现高可用性 | 非常高，分布式架构 | 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 | 有较低的概率丢失数据 |  | 经过参数优化配置，可以做到0丢失 | 经过参数优化配置，消息可以做到0丢失 |
| 功能支持 | MQ领域的功能极其完备 | 基于erlang开发，所以并发能力很强，性能极其好，延时很低 | MQ功能较为完善，还是分布式的，扩展性好 | 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，是事实上的标准 |
| 优劣势总结 | 非常成熟，功能强大，在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  而且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  而且确实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 | 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  而且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。  而且erlang开发，国内有几个公司有实力做erlang源码级别的研究和定制？如果说你没这个实力的话，确实偶尔会有一些问题，你很难去看懂源码，你公司对这个东西的掌控很弱，基本职能依赖于开源社区的快速维护和修复bug。  而且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。 | 接口简单易用，而且毕竟在阿里大规模应用过，有阿里品牌保障  日处理消息上百亿之多，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，阿里出品都是java系的，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  还有就是阿里出台的技术，你得做好这个技术万一被抛弃，社区黄掉的风险，那如果你们公司有技术实力我觉得用RocketMQ挺好的 | 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  而且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  这个特性天然适合大数据实时计算以及日志收集 |

综上所述，各种对比之后，我个人倾向于是：

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，所以大家还是算了吧，我个人不推荐用这个了；

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，但是我提醒一下自己想好社区万一突然黄掉的风险，对自己公司技术实力有绝对自信的，我推荐用RocketMQ，否则回去老老实实用RabbitMQ吧，人是活跃开源社区，绝对不会黄

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

Day5

1、面试题

如何保证消息队列的高可用啊？

2、面试官心理分析

如果有人问到你MQ的知识，高可用是必问的，因为MQ的缺点，我刚才已经说过了，有好多，导致系统可用性降低，等等。所以只要你用了MQ，接下来问的一些要点肯定就是围绕着MQ的那些缺点怎么来解决了。

要是你傻乎乎的就干用了一个MQ，各种问题从来没考虑过，那你就杯具了，面试官对你的印象就是，只会简单实用一些技术，没任何思考，马上对你的印象就不太好了。这样的同学招进来要是做个20k薪资以内的普通小弟还凑合。如果招进来做薪资20多k的高工，那就惨了，让你设计个系统，里面肯定一堆坑，出了事故公司受损失，团队一起背锅。

去年的事儿，非常大的互联网公司，非常核心的系统，就是疏忽了MQ，没考虑MQ如何保证高可用，如果MQ挂了怎么办，导致几个小时系统不可用，公司损失几千万，team背锅，你闹的祸，你老大帮你一起背锅

3、面试题剖析

这个问题这么问是很好的，因为不能问你kafka的高可用性怎么保证啊？ActiveMQ的高可用性怎么保证啊？一个面试官要是这么问就显得很没水平，人家可能用的就是RabbitMQ，没用过Kafka，你上来问人家kafka干什么？这不是摆明了刁难人么。

所以有水平的面试官，问的是MQ的高可用性怎么保证？这样就是你用过哪个MQ，你就说说你对那个MQ的高可用性的理解。

（1）RabbitMQ的高可用性

RabbitMQ是比较有代表性的，因为是基于主从做高可用性的，我们就以他为例子讲解第一种MQ的高可用性怎么实现。

rabbitmq有三种模式：单机模式，普通集群模式，镜像集群模式

1）单机模式

就是demo级别的，一般就是你本地启动了玩玩儿的，没人生产用单机模式

2）普通集群模式

意思就是在多台机器上启动多个rabbitmq实例，每个机器启动一个。但是你创建的queue，只会放在一个rabbtimq实例上，但是每个实例都同步queue的元数据。完了你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从queue所在实例上拉取数据过来。

这种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个queue所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放queue的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让rabbitmq落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个queue拉取数据。

所以这个事儿就比较尴尬了，这就没有什么所谓的高可用性可言了，这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个queue的读写操作。

3）镜像集群模式

这种模式，才是所谓的rabbitmq的高可用模式，跟普通集群模式不一样的是，你创建的queue，无论元数据还是queue里的消息都会存在于多个实例上，然后每次你写消息到queue的时候，都会自动把消息到多个实例的queue里进行消息同步。

这样的话，好处在于，你任何一个机器宕机了，没事儿，别的机器都可以用。坏处在于，第一，这个性能开销也太大了吧，消息同步所有机器，导致网络带宽压力和消耗很重！第二，这么玩儿，就没有扩展性可言了，如果某个queue负载很重，你加机器，新增的机器也包含了这个queue的所有数据，并没有办法线性扩展你的queue

那么怎么开启这个镜像集群模式呢？我这里简单说一下，避免面试人家问你你不知道，其实很简单rabbitmq有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候可以要求数据同步到所有节点的，也可以要求就同步到指定数量的节点，然后你再次创建queue的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

（2）kafka的高可用性

kafka一个最基本的架构认识：多个broker组成，每个broker是一个节点；你创建一个topic，这个topic可以划分为多个partition，每个partition可以存在于不同的broker上，每个partition就放一部分数据。

这就是天然的分布式消息队列，就是说一个topic的数据，是分散放在多个机器上的，每个机器就放一部分数据。

实际上rabbitmq之类的，并不是分布式消息队列，他就是传统的消息队列，只不过提供了一些集群、HA的机制而已，因为无论怎么玩儿，rabbitmq一个queue的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个queue的完整数据。

kafka 0.8以前，是没有HA机制的，就是任何一个broker宕机了，那个broker上的partition就废了，没法写也没法读，没有什么高可用性可言。

kafka 0.8以后，提供了HA机制，就是replica副本机制。每个partition的数据都会同步到吉他机器上，形成自己的多个replica副本。然后所有replica会选举一个leader出来，那么生产和消费都跟这个leader打交道，然后其他replica就是follower。写的时候，leader会负责把数据同步到所有follower上去，读的时候就直接读leader上数据即可。只能读写leader？很简单，要是你可以随意读写每个follower，那么就要care数据一致性的问题，系统复杂度太高，很容易出问题。kafka会均匀的将一个partition的所有replica分布在不同的机器上，这样才可以提高容错性。

这么搞，就有所谓的高可用性了，因为如果某个broker宕机了，没事儿，那个broker上面的partition在其他机器上都有副本的，如果这上面有某个partition的leader，那么此时会重新选举一个新的leader出来，大家继续读写那个新的leader即可。这就有所谓的高可用性了。

写数据的时候，生产者就写leader，然后leader将数据落地写本地磁盘，接着其他follower自己主动从leader来pull数据。一旦所有follower同步好数据了，就会发送ack给leader，leader收到所有follower的ack之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）

消费的时候，只会从leader去读，但是只有一个消息已经被所有follower都同步成功返回ack的时候，这个消息才会被消费者读到。

实际上这块机制，讲深了，是可以非常之深入的，但是我还是回到我们这个课程的主题和定位，聚焦面试，至少你听到这里大致明白了kafka是如何保证高可用机制的了，对吧？不至于一无所知，现场还能给面试官画画图。要遇上面试官确实是kafka高手，深挖了问，那你只能说不好意思，太深入的你没研究过。

但是大家一定要明白，这个事情是要权衡的，你现在是要快速突击常见面试题体系，而不是要深入学习kafka，要深入学习kafka，你是没那么多时间的。你只能确保，你之前也许压根儿不知道这块，但是现在你知道了，面试被问到，你大概可以说一说。然后很多其他的候选人，也许还不如你，没看过这个，被问到了压根儿答不出来，相比之下，你还能说点出来，大概就是这个意思了。

Day7

1、面试题

如何保证消息不被重复消费啊（如何保证消息消费时的幂等性）？

2、面试官心里分析

其实这个很常见的一个问题，这俩问题基本可以连起来问。既然是消费消息，那肯定要考虑考虑会不会重复消费？能不能避免重复消费？或者重复消费了也别造成系统异常可以吗？这个是MQ领域的基本问题，其实本质上还是问你使用消息队列如何保证幂等性，这个是你架构里要考虑的一个问题。

面试官问你，肯定是必问的，这是你要考虑的实际生产上的系统设计问题。

3、面试题剖析

回答这个问题，首先你别听到重复消息这个事儿，就一无所知吧，你先大概说一说可能会有哪些重复消费的问题。

首先就是比如rabbitmq、rocketmq、kafka，都有可能会出现消费重复消费的问题，正常。因为这问题通常不是mq自己保证的，是给你保证的。然后我们挑一个kafka来举个例子，说说怎么重复消费吧。

kafka实际上有个offset的概念，就是每个消息写进去，都有一个offset，代表他的序号，然后consumer消费了数据之后，每隔一段时间，会把自己消费过的消息的offset提交一下，代表我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的offset来继续消费吧。

但是凡事总有意外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接kill进程了，再重启。这会导致consumer有些消息处理了，但是没来得及提交offset，尴尬了。重启之后，少数消息会再次消费一次。

其实重复消费不可怕，可怕的是你没考虑到重复消费之后，怎么保证幂等性。

给你举个例子吧。假设你有个系统，消费一条往数据库里插入一条，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下已经消费过了，直接扔了，不就保留了一条数据？

一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性

幂等性，我通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

那所以第二个问题来了，怎么保证消息队列消费的幂等性？

其实还是得结合业务来思考，我这里给几个思路：

（1）比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update一下好吧

（2）比如你是写redis，那没问题了，反正每次都是set，天然幂等性

（3）比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的id，类似订单id之类的东西，然后你这里消费到了之后，先根据这个id去比如redis里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个id写redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

还有比如基于数据库的唯一键来保证重复数据不会重复插入多条，我们之前线上系统就有这个问题，就是拿到数据的时候，每次重启可能会有重复，因为kafka消费者还没来得及提交offset，重复数据拿到了以后我们插入的时候，因为有唯一键约束了，所以重复数据只会插入报错，不会导致数据库中出现脏数据

如何保证MQ的消费是幂等性的，需要结合具体的业务来看

Day8

1、面试题

如何保证消息的可靠性传输（如何处理消息丢失的问题）？

2、面试官心里分析

这个是肯定的，用mq有个基本原则，就是数据不能多一条，也不能少一条，不能多，就是刚才说的重复消费和幂等性问题。不能少，就是说这数据别搞丢了。那这个问题你必须得考虑一下。

如果说你这个是用mq来传递非常核心的消息，比如说计费，扣费的一些消息，因为我以前设计和研发过一个公司非常核心的广告平台，计费系统，计费系统是很重的一个业务，操作是很耗时的。所以说广告系统整体的架构里面，实际上是将计费做成异步化的，然后中间就是加了一个MQ。

我们当时为了确保说这个MQ传递过程中绝对不会把计费消息给弄丢，花了很多的精力。广告主投放了一个广告，明明说好了，用户点击一次扣费1块钱。结果要是用户动不动点击了一次，扣费的时候搞的消息丢了，我们公司就会不断的少几块钱，几块钱，积少成多，这个就对公司是一个很大的损失。

3、面试题剖析

这个丢数据，mq一般分为两种，要么是mq自己弄丢了，要么是我们消费的时候弄丢了。咱们从rabbitmq和kafka分别来分析一下吧

rabbitmq这种mq，一般来说都是承载公司的核心业务的，数据是绝对不能弄丢的

（1）rabbitmq

1）生产者弄丢了数据

生产者将数据发送到rabbitmq的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。

此时可以选择用rabbitmq提供的事务功能，就是生产者发送数据之前开启rabbitmq事务（channel.txSelect），然后发送消息，如果消息没有成功被rabbitmq接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，rabbitmq事务机制一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果你要确保说写rabbitmq的消息别丢，可以开启confirm模式，在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了rabbitmq中，rabbitmq会给你回传一个ack消息，告诉你说这个消息ok了。如果rabbitmq没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和cnofirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息rabbitmq接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用confirm机制的。

2）rabbitmq弄丢了数据

就是rabbitmq自己弄丢了数据，这个你必须开启rabbitmq的持久化，就是消息写入之后会持久化到磁盘，哪怕是rabbitmq自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，rabbitmq还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证rabbitmq持久化queue的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时rabbitmq就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，rabbitmq哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。

而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，rabbitmq挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。

哪怕是你给rabbitmq开启了持久化机制，也有一种可能，就是这个消息写到了rabbitmq中，但是还没来得及持久化到磁盘上，结果不巧，此时rabbitmq挂了，就会导致内存里的一点点数据会丢失。

3）消费端弄丢了数据

rabbitmq如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，rabbitmq认为你都消费了，这数据就丢了。

这个时候得用rabbitmq提供的ack机制，简单来说，就是你关闭rabbitmq自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那rabbitmq就认为你还没处理完，这个时候rabbitmq会把这个消费分配给别的consumer去处理，消息是不会丢的。

（2）kafka

1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

这不是一样么，大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。

然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了

2）kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partiton的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了

所以此时一般是要求起码设置如下4个参数：

给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本

在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个是要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧

在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了

在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了

我们生产环境就是按照上述要求配置的，这样配置之后，至少在kafka broker端就可以保证在leader所在broker发生故障，进行leader切换时，数据不会丢失

3）生产者会不会弄丢数据

如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

Day9

1、面试题

如何保证消息的顺序性？

2、面试官心里分析

其实这个也是用MQ的时候必问的话题，第一看看你了解不了解顺序这个事儿？第二看看你有没有办法保证消息是有顺序的？这个生产系统中常见的问题。

3、面试题剖析

我举个例子，我们以前做过一个mysql binlog同步的系统，压力还是非常大的，日同步数据要达到上亿。mysql -> mysql，常见的一点在于说大数据team，就需要同步一个mysql库过来，对公司的业务系统的数据做各种复杂的操作。

你在mysql里增删改一条数据，对应出来了增删改3条binlog，接着这三条binlog发送到MQ里面，到消费出来依次执行，起码得保证人家是按照顺序来的吧？不然本来是：增加、修改、删除；你楞是换了顺序给执行成删除、修改、增加，不全错了么。

本来这个数据同步过来，应该最后这个数据被删除了；结果你搞错了这个顺序，最后这个数据保留下来了，数据同步就出错了。

先看看顺序会错乱的俩场景

（1）rabbitmq：一个queue，多个consumer，这不明显乱了

（2）kafka：一个topic，一个partition，一个consumer，内部多线程，这不也明显乱了

那如何保证消息的顺序性呢？简单简单

（1）rabbitmq：拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理

（2）kafka：一个topic，一个partition，一个consumer，内部单线程消费，写N个内存queue，然后N个线程分别消费一个内存queue即可

Day10

1、面试题

如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

2、面试官心里分析

你看这问法，其实本质针对的场景，都是说，可能你的消费端出了问题，不消费了，或者消费的极其极其慢。接着就坑爹了，可能你的消息队列集群的磁盘都快写满了，都没人消费，这个时候怎么办？或者是整个这就积压了几个小时，你这个时候怎么办？或者是你积压的时间太长了，导致比如rabbitmq设置了消息过期时间后就没了怎么办？

所以就这事儿，其实线上挺常见的，一般不出，一出就是大case，一般常见于，举个例子，消费端每次消费之后要写mysql，结果mysql挂了，消费端hang那儿了，不动了。或者是消费端出了个什么叉子，导致消费速度极其慢。

3、面试题分析

关于这个事儿，我们一个一个来梳理吧，先假设一个场景，我们现在消费端出故障了，然后大量消息在mq里积压，现在事故了，慌了

（1）大量消息在mq里积压了几个小时了还没解决

几千万条数据在MQ里积压了七八个小时，从下午4点多，积压到了晚上很晚，10点多，11点多

这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条

所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

1）先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉

2）新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量

3）然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue

4）接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据

5）这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据

6）等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息

（2）这里我们假设再来第二个坑

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次

（3）然后我们再来假设第三个坑

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

Day11

1、面试题

如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路

2、面试官心里分析

其实聊到这个问题，一般面试官要考察两块：

（1）你有没有对某一个消息队列做过较为深入的原理的了解，或者从整体了解把握住一个mq的架构原理

（2）看看你的设计能力，给你一个常见的系统，就是消息队列系统，看看你能不能从全局把握一下整体架构设计，给出一些关键点出来

说实话，我一般面类似问题的时候，大部分人基本都会蒙，因为平时从来没有思考过类似的问题，大多数人就是平时埋头用，从来不去思考背后的一些东西。类似的问题，我经常问的还有，如果让你来设计一个spring框架你会怎么做？如果让你来设计一个dubbo框架你会怎么做？如果让你来设计一个mybatis框架你会怎么做？

3、面试题剖析

其实回答这类问题，说白了，起码不求你看过那技术的源码，起码你大概知道那个技术的基本原理，核心组成部分，基本架构构成，然后参照一些开源的技术把一个系统设计出来的思路说一下就好

比如说这个消息队列系统，我们来从以下几个角度来考虑一下

（1）首先这个mq得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下kafka的设计理念，broker -> topic -> partition，每个partition放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给topic增加partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

（2）其次你得考虑一下这个mq的数据要不要落地磁盘吧？那肯定要了，落磁盘，才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是kafka的思路。

1. 其次你考虑一下你的mq的可用性啊？这个事儿，具体参考我们之前可用性那个环节讲解的kafka的高可用保障机制。多副本 -> leader & follower -> broker挂了重新选举leader即可对外服务。

（4）能不能支持数据0丢失啊？可以的，参考我们之前说的那个kafka数据零丢失方案

其实一个mq肯定是很复杂的，面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。

Day12

一般而言，如果一个面试官水平还算不错，会沿着从浅入深的环节深入挖一个点。比如我吧，其实按照这个思路可以一直问下去，除了这里的7个问题之外，甚至能挑着你熟悉的一个mq一直问到源码级别非常底层。我还可能会结合项目来仔细问，我可能会先让你给我详细说说你的业务细节，然后将你的业务跟这些mq的问题场景结合起来，看看你每个细节是怎么处理的。

但是确实因为我们这个是面试突击型课程，不是什么kafka源码剖析课，也不是什么RocketMQ高并发架构项目实战课程，所以只能讲到这个程度。

所以我们这个课程只能让你从大面儿上，基本常见问题可以回答出来。基本上mq这块你能答到这个程度，你基本知识面儿是有了，但是深度是绝对没有的。所以如果一个面试官就问问这些问题，感觉你面儿上过的去了，那就恭喜你了。但是如果碰到我这种难缠的面试官，喜欢深挖底层，细扣项目细节的，那可能确实是不行的。

如果你碰到人家在7个问题之外还死扣着你问的，那你最好是认一下怂，就说你确实没研究那么深过，如果你面的就是个一般的职位，那可能就过去了。就我而言，如果招聘的就是个普通职位，而你能答到这个程度，那么就觉得说的过去了。毕竟说实话，相当大比例的程序员出去面java职位的时候，mq这块还回答不到这个程度呢。你能答好这些，至少比之前一无所知的你好了一些，也比很多没准备过的程序员都好了很多。

最后说一个技巧，要是确实碰一个面试官连这7个问题都没问满，只要他提到mq，你自己就和盘托出一整套的东西，你就说，mq你们之前遇到过什么问题，巴拉巴拉，你们的方案是什么，自己突出自己会的东西

Day13

分布式开始

业内目前来说事实上的一个标准，就是分布式搜索引擎一般大家都用elasticsearch，es，solr，但是确实，这两年大家一般都用更加易用的es。

lucene

如果你确实真的不连lucene都不知道是什么？我觉得你确实不应该，lucene底层的原理是一个东西，叫做倒排索引。太基础了。

百度，搜索一下lucene入门，了解一下lucene是什么？倒排索引是什么？全文检索是什么？写个lucene的demo程序体验一把。

elasticsearch

百度，搜索一下：elasticsearch入门，初步至少知道es的一些基本概念，然后包括es的基本部署和基本的使用

面试官可能会怎么问？

（1）es的分布式架构原理能说一下么（es是如何实现分布式的啊）？

（2）es写入数据的工作原理是什么啊？es查询数据的工作原理是什么啊？

（3）es在数据量很大的情况下（数十亿级别）如何提高查询性能啊？

（4）es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

Day14

1、面试题

es的分布式架构原理能说一下么（es是如何实现分布式的啊）？

2、面试官心里分析

在搜索这块，lucene是最流行的搜索库。几年前业内一般都问，你了解lucene吗？你知道倒排索引的原理吗？现在早已经out了，因为现在很多项目都是直接用基于lucene的分布式搜索引擎——elasticsearch，简称为es。

而现在分布式搜索基本已经成为大部分互联网行业的java系统的标配，其中尤为流行的就是es，前几年es没火的时候，大家一般用solr。但是这两年基本大部分企业和项目都开始转向es了。

所以互联网面试，肯定会跟你聊聊分布式搜索引擎，也就一定会聊聊es，如果你确实不知道，那你真的就out了。

如果面试官问你第一个问题，确实一般都会问你es的分布式架构设计能介绍一下么？就看看你对分布式搜索引擎架构的一个基本理解。

3、额外的友情提示

同学啊，如果你看到这里发现自己对es一无所知，没事儿，保持淡定，暂停一下课程。然后上百度搜一下es是啥？本机启动个es？然后写个es的hello world感受一下？然后搜个帖子把es常见的几个操作都执行一遍（聚合、常见搜索语法之类的）？ok了，1~2小时熟悉足够了，回来吧，继续看我们的课程。

4、面试题剖析

elasticsearch设计的理念就是分布式搜索引擎，底层其实还是基于lucene的。

核心思想就是在多台机器上启动多个es进程实例，组成了一个es集群。

es中存储数据的基本单位是索引，比如说你现在要在es中存储一些订单数据，你就应该在es中创建一个索引，order\_idx，所有的订单数据就都写到这个索引里面去，一个索引差不多就是相当于是mysql里的一张表。index -> type -> mapping -> document -> field。

index：mysql里的一张表

type：没法跟mysql里去对比，一个index里可以有多个type，每个type的字段都是差不多的，但是有一些略微的差别。

好比说，有一个index，是订单index，里面专门是放订单数据的。就好比说你在mysql中建表，有些订单是实物商品的订单，就好比说一件衣服，一双鞋子；有些订单是虚拟商品的订单，就好比说游戏点卡，话费充值。就两种订单大部分字段是一样的，但是少部分字段可能有略微的一些差别。

所以就会在订单index里，建两个type，一个是实物商品订单type，一个是虚拟商品订单type，这两个type大部分字段是一样的，少部分字段是不一样的。

很多情况下，一个index里可能就一个type，但是确实如果说是一个index里有多个type的情况，你可以认为index是一个类别的表，具体的每个type代表了具体的一个mysql中的表

每个type有一个mapping，如果你认为一个type是一个具体的一个表，index代表了多个type的同属于的一个类型，mapping就是这个type的表结构定义，你在mysql中创建一个表，肯定是要定义表结构的，里面有哪些字段，每个字段是什么类型。。。

mapping就代表了这个type的表结构的定义，定义了这个type中每个字段名称，字段是什么类型的，然后还有这个字段的各种配置

实际上你往index里的一个type里面写的一条数据，叫做一条document，一条document就代表了mysql中某个表里的一行给，每个document有多个field，每个field就代表了这个document中的一个字段的值

接着你搞一个索引，这个索引可以拆分成多个shard，每个shard存储部分数据。

接着就是这个shard的数据实际是有多个备份，就是说每个shard都有一个primary shard，负责写入数据，但是还有几个replica shard。primary shard写入数据之后，会将数据同步到其他几个replica shard上去。

通过这个replica的方案，每个shard的数据都有多个备份，如果某个机器宕机了，没关系啊，还有别的数据副本在别的机器上呢。高可用了吧。

es集群多个节点，会自动选举一个节点为master节点，这个master节点其实就是干一些管理的工作的，比如维护索引元数据拉，负责切换primary shard和replica shard身份拉，之类的。

要是master节点宕机了，那么会重新选举一个节点为master节点。

如果是非master节点宕机了，那么会由master节点，让那个宕机节点上的primary shard的身份转移到其他机器上的replica shard。急着你要是修复了那个宕机机器，重启了之后，master节点会控制将缺失的replica shard分配过去，同步后续修改的数据之类的，让集群恢复正常。

其实上述就是elasticsearch作为一个分布式搜索引擎最基本的一个架构设计

Day15

1、面试题

es写入数据的工作原理是什么啊？es查询数据的工作原理是什么啊？

2、面试官心理分析

问这个，其实面试官就是要看看你了解不了解es的一些基本原理，因为用es无非就是写入数据，搜索数据。你要是不明白你发起一个写入和搜索请求的时候，es在干什么，那你真的就是。。。。

对es基本就是个黑盒，你还能干啥？你唯一能干的就是用es的api读写数据了。。。要是出点什么问题，你啥都不知道，那还能指望你什么呢？是不是。。

3、面试题剖析

（1）es写数据过程

1）客户端选择一个node发送请求过去，这个node就是coordinating node（协调节点）

2）coordinating node，对document进行路由，将请求转发给对应的node（有primary shard）

3）实际的node上的primary shard处理请求，然后将数据同步到replica node

4）coordinating node，如果发现primary node和所有replica node都搞定之后，就返回响应结果给客户端

（2）es读数据过程

查询，GET某一条数据，写入了某个document，这个document会自动给你分配一个全局唯一的id，doc id，同时也是根据doc id进行hash路由到对应的primary shard上面去。也可以手动指定doc id，比如用订单id，用户id。

你可以通过doc id来查询，会根据doc id进行hash，判断出来当时把doc id分配到了哪个shard上面去，从那个shard去查询

1）客户端发送请求到任意一个node，成为coordinate node

2）coordinate node对document进行路由，将请求转发到对应的node，此时会使用round-robin随机轮询算法，在primary shard以及其所有replica中随机选择一个，让读请求负载均衡

3）接收请求的node返回document给coordinate node

4）coordinate node返回document给客户端

（3）es搜索数据过程

es最强大的是做全文检索，就是比如你有三条数据

java真好玩儿啊

java好难学啊

j2ee特别牛

你根据java关键词来搜索，将包含java的document给搜索出来

es就会给你返回：java真好玩儿啊，java好难学啊

1）客户端发送请求到一个coordinate node

2）协调节点将搜索请求转发到所有的shard对应的primary shard或replica shard也可以

3）query phase：每个shard将自己的搜索结果（其实就是一些doc id），返回给协调节点，由协调节点进行数据的合并、排序、分页等操作，产出最终结果

4）fetch phase：接着由协调节点，根据doc id去各个节点上拉取实际的document数据，最终返回给客户端

（4）搜索的底层原理，倒排索引，画图说明传统数据库和倒排索引的区别

（5）写数据底层原理

1）先写入buffer，在buffer里的时候数据是搜索不到的；同时将数据写入translog日志文件

2）如果buffer快满了，或者到一定时间，就会将buffer数据refresh到一个新的segment file中，但是此时数据不是直接进入segment file的磁盘文件的，而是先进入os cache的。这个过程就是refresh。

每隔1秒钟，es将buffer中的数据写入一个新的segment file，每秒钟会产生一个新的磁盘文件，segment file，这个segment file中就存储最近1秒内buffer中写入的数据

但是如果buffer里面此时没有数据，那当然不会执行refresh操作咯，每秒创建换一个空的segment file，如果buffer里面有数据，默认1秒钟执行一次refresh操作，刷入一个新的segment file中

操作系统里面，磁盘文件其实都有一个东西，叫做os cache，操作系统缓存，就是说数据写入磁盘文件之前，会先进入os cache，先进入操作系统级别的一个内存缓存中去

只要buffer中的数据被refresh操作，刷入os cache中，就代表这个数据就可以被搜索到了

为什么叫es是准实时的？NRT，near real-time，准实时。默认是每隔1秒refresh一次的，所以es是准实时的，因为写入的数据1秒之后才能被看到。

可以通过es的restful api或者java api，手动执行一次refresh操作，就是手动将buffer中的数据刷入os cache中，让数据立马就可以被搜索到。

只要数据被输入os cache中，buffer就会被清空了，因为不需要保留buffer了，数据在translog里面已经持久化到磁盘去一份了

3）只要数据进入os cache，此时就可以让这个segment file的数据对外提供搜索了

4）重复1~3步骤，新的数据不断进入buffer和translog，不断将buffer数据写入一个又一个新的segment file中去，每次refresh完buffer清空，translog保留。随着这个过程推进，translog会变得越来越大。当translog达到一定长度的时候，就会触发commit操作。

buffer中的数据，倒是好，每隔1秒就被刷到os cache中去，然后这个buffer就被清空了。所以说这个buffer的数据始终是可以保持住不会填满es进程的内存的。

每次一条数据写入buffer，同时会写入一条日志到translog日志文件中去，所以这个translog日志文件是不断变大的，当translog日志文件大到一定程度的时候，就会执行commit操作。

5）commit操作发生第一步，就是将buffer中现有数据refresh到os cache中去，清空buffer

6）将一个commit point写入磁盘文件，里面标识着这个commit point对应的所有segment file

7）强行将os cache中目前所有的数据都fsync到磁盘文件中去

translog日志文件的作用是什么？就是在你执行commit操作之前，数据要么是停留在buffer中，要么是停留在os cache中，无论是buffer还是os cache都是内存，一旦这台机器死了，内存中的数据就全丢了。

所以需要将数据对应的操作写入一个专门的日志文件，translog日志文件中，一旦此时机器宕机，再次重启的时候，es会自动读取translog日志文件中的数据，恢复到内存buffer和os cache中去。

commit操作：1、写commit point；2、将os cache数据fsync强刷到磁盘上去；3、清空translog日志文件

8）将现有的translog清空，然后再次重启启用一个translog，此时commit操作完成。默认每隔30分钟会自动执行一次commit，但是如果translog过大，也会触发commit。整个commit的过程，叫做flush操作。我们可以手动执行flush操作，就是将所有os cache数据刷到磁盘文件中去。

不叫做commit操作，flush操作。es中的flush操作，就对应着commit的全过程。我们也可以通过es api，手动执行flush操作，手动将os cache中的数据fsync强刷到磁盘上去，记录一个commit point，清空translog日志文件。

9）translog其实也是先写入os cache的，默认每隔5秒刷一次到磁盘中去，所以默认情况下，可能有5秒的数据会仅仅停留在buffer或者translog文件的os cache中，如果此时机器挂了，会丢失5秒钟的数据。但是这样性能比较好，最多丢5秒的数据。也可以将translog设置成每次写操作必须是直接fsync到磁盘，但是性能会差很多。

实际上你在这里，如果面试官没有问你es丢数据的问题，你可以在这里给面试官炫一把，你说，其实es第一是准实时的，数据写入1秒后可以搜索到；可能会丢失数据的，你的数据有5秒的数据，停留在buffer、translog os cache、segment file os cache中，有5秒的数据不在磁盘上，此时如果宕机，会导致5秒的数据丢失。

如果你希望一定不能丢失数据的话，你可以设置个参数，官方文档，百度一下。每次写入一条数据，都是写入buffer，同时写入磁盘上的translog，但是这会导致写性能、写入吞吐量会下降一个数量级。本来一秒钟可以写2000条，现在你一秒钟只能写200条，都有可能。

10）如果是删除操作，commit的时候会生成一个.del文件，里面将某个doc标识为deleted状态，那么搜索的时候根据.del文件就知道这个doc被删除了

11）如果是更新操作，就是将原来的doc标识为deleted状态，然后新写入一条数据

12）buffer每次refresh一次，就会产生一个segment file，所以默认情况下是1秒钟一个segment file，segment file会越来越多，此时会定期执行merge

13）每次merge的时候，会将多个segment file合并成一个，同时这里会将标识为deleted的doc给物理删除掉，然后将新的segment file写入磁盘，这里会写一个commit point，标识所有新的segment file，然后打开segment file供搜索使用，同时删除旧的segment file。

es里的写流程，有4个底层的核心概念，refresh、flush、translog、merge

当segment file多到一定程度的时候，es就会自动触发merge操作，将多个segment file给merge成一个segment file。

Day16

1、面试题

es在数据量很大的情况下（数十亿级别）如何提高查询效率啊？

2、面试官心里分析

问这个问题，是肯定的，说白了，就是看你有没有实际干过es，因为啥？es说白了其实性能并没有你想象中那么好的。很多时候数据量大了，特别是有几亿条数据的时候，可能你会懵逼的发现，跑个搜索怎么一下5秒~10秒，坑爹了。第一次搜索的时候，是5~10秒，后面反而就快了，可能就几百毫秒。

你就很懵，每个用户第一次访问都会比较慢，比较卡么？

所以你要是没玩儿过es，或者就是自己玩玩儿demo，被问到这个问题容易懵逼，显示出你对es确实玩儿的不怎么样

3、面试题剖析

说实话，es性能优化是没有什么银弹的，啥意思呢？就是不要期待着随手调一个参数，就可以万能的应对所有的性能慢的场景。也许有的场景是你换个参数，或者调整一下语法，就可以搞定，但是绝对不是所有场景都可以这样。

一块一块来分析吧

在这个海量数据的场景下，如何提升es搜索的性能，也是我们之前生产环境实践经验所得

（1）性能优化的杀手锏——filesystem cache

os cache，操作系统的缓存

你往es里写的数据，实际上都写到磁盘文件里去了，磁盘文件里的数据操作系统会自动将里面的数据缓存到os cache里面去

es的搜索引擎严重依赖于底层的filesystem cache，你如果给filesystem cache更多的内存，尽量让内存可以容纳所有的indx segment file索引数据文件，那么你搜索的时候就基本都是走内存的，性能会非常高。

性能差距可以有大，我们之前很多的测试和压测，如果走磁盘一般肯定上秒，搜索性能绝对是秒级别的，1秒，5秒，10秒。但是如果是走filesystem cache，是走纯内存的，那么一般来说性能比走磁盘要高一个数量级，基本上就是毫秒级的，从几毫秒到几百毫秒不等。

之前有个学员，一直在问我，说他的搜索性能，聚合性能，倒排索引，正排索引，磁盘文件，十几秒。。。。

学员的真实案例

比如说，你，es节点有3台机器，每台机器，看起来内存很多，64G，总内存，64 \* 3 = 192g

每台机器给es jvm heap是32G，那么剩下来留给filesystem cache的就是每台机器才32g，总共集群里给filesystem cache的就是32 \* 3 = 96g内存

我就问他，ok，那么就是你往es集群里写入的数据有多少数据量？

如果你此时，你整个，磁盘上索引数据文件，在3台机器上，一共占用了1T的磁盘容量，你的es数据量是1t，每台机器的数据量是300g

你觉得你的性能能好吗？filesystem cache的内存才100g，十分之一的数据可以放内存，其他的都在磁盘，然后你执行搜索操作，大部分操作都是走磁盘，性能肯定差

当时他们的情况就是这样子，es在测试，弄了3台机器，自己觉得还不错，64G内存的物理机。自以为可以容纳1T的数据量。

归根结底，你要让es性能要好，最佳的情况下，就是你的机器的内存，至少可以容纳你的总数据量的一半

比如说，你一共要在es中存储1T的数据，那么你的多台机器留个filesystem cache的内存加起来综合，至少要到512G，至少半数的情况下，搜索是走内存的，性能一般可以到几秒钟，2秒，3秒，5秒

如果最佳的情况下，我们自己的生产环境实践经验，所以说我们当时的策略，是仅仅在es中就存少量的数据，就是你要用来搜索的那些索引，内存留给filesystem cache的，就100G，那么你就控制在100gb以内，相当于是，你的数据几乎全部走内存来搜索，性能非常之高，一般可以在1秒以内

比如说你现在有一行数据

id name age ....30个字段

但是你现在搜索，只需要根据id name age三个字段来搜索

如果你傻乎乎的往es里写入一行数据所有的字段，就会导致说70%的数据是不用来搜索的，结果硬是占据了es机器上的filesystem cache的空间，单挑数据的数据量越大，就会导致filesystem cahce能缓存的数据就越少

仅仅只是写入es中要用来检索的少数几个字段就可以了，比如说，就写入es id name age三个字段就可以了，然后你可以把其他的字段数据存在mysql里面，我们一般是建议用es + hbase的这么一个架构。

hbase的特点是适用于海量数据的在线存储，就是对hbase可以写入海量数据，不要做复杂的搜索，就是做很简单的一些根据id或者范围进行查询的这么一个操作就可以了

从es中根据name和age去搜索，拿到的结果可能就20个doc id，然后根据doc id到hbase里去查询每个doc id对应的完整的数据，给查出来，再返回给前端。

你最好是写入es的数据小于等于，或者是略微大于es的filesystem cache的内存容量

然后你从es检索可能就花费20ms，然后再根据es返回的id去hbase里查询，查20条数据，可能也就耗费个30ms，可能你原来那么玩儿，1T数据都放es，会每次查询都是5~10秒，现在可能性能就会很高，每次查询就是50ms。

elastcisearch减少数据量仅仅放要用于搜索的几个关键字段即可，尽量写入es的数据量跟es机器的filesystem cache是差不多的就可以了；其他不用来检索的数据放hbase里，或者mysql。

所以之前有些学员也是问，我也是跟他们说，尽量在es里，就存储必须用来搜索的数据，比如说你现在有一份数据，有100个字段，其实用来搜索的只有10个字段，建议是将10个字段的数据，存入es，剩下90个字段的数据，可以放mysql，hadoop hbase，都可以

这样的话，es数据量很少，10个字段的数据，都可以放内存，就用来搜索，搜索出来一些id，通过id去mysql，hbase里面去查询明细的数据

（2）数据预热

假如说，哪怕是你就按照上述的方案去做了，es集群中每个机器写入的数据量还是超过了filesystem cache一倍，比如说你写入一台机器60g数据，结果filesystem cache就30g，还是有30g数据留在了磁盘上。

举个例子，就比如说，微博，你可以把一些大v，平时看的人很多的数据给提前你自己后台搞个系统，每隔一会儿，你自己的后台系统去搜索一下热数据，刷到filesystem cache里去，后面用户实际上来看这个热数据的时候，他们就是直接从内存里搜索了，很快。

电商，你可以将平时查看最多的一些商品，比如说iphone 8，热数据提前后台搞个程序，每隔1分钟自己主动访问一次，刷到filesystem cache里去。

对于那些你觉得比较热的，经常会有人访问的数据，最好做一个专门的缓存预热子系统，就是对热数据，每隔一段时间，你就提前访问一下，让数据进入filesystem cache里面去。这样期待下次别人访问的时候，一定性能会好一些。

（3）冷热分离

关于es性能优化，数据拆分，我之前说将大量不搜索的字段，拆分到别的存储中去，这个就是类似于后面我最后要讲的mysql分库分表的垂直拆分。

es可以做类似于mysql的水平拆分，就是说将大量的访问很少，频率很低的数据，单独写一个索引，然后将访问很频繁的热数据单独写一个索引

你最好是将冷数据写入一个索引中，然后热数据写入另外一个索引中，这样可以确保热数据在被预热之后，尽量都让他们留在filesystem os cache里，别让冷数据给冲刷掉。

你看，假设你有6台机器，2个索引，一个放冷数据，一个放热数据，每个索引3个shard

3台机器放热数据index；另外3台机器放冷数据index

然后这样的话，你大量的时候是在访问热数据index，热数据可能就占总数据量的10%，此时数据量很少，几乎全都保留在filesystem cache里面了，就可以确保热数据的访问性能是很高的。

但是对于冷数据而言，是在别的index里的，跟热数据index都不再相同的机器上，大家互相之间都没什么联系了。如果有人访问冷数据，可能大量数据是在磁盘上的，此时性能差点，就10%的人去访问冷数据；90%的人在访问热数据。

（4）document模型设计

有不少同学问我，mysql，有两张表

订单表：id order\_code total\_price

1 测试订单 5000

订单条目表：id order\_id goods\_id purchase\_count price

1 1 1 2 2000

2 1 2 5 200

我在mysql里，都是select \* from order join order\_item on order.id=order\_item.order\_id where order.id=1

1 测试订单 5000 1 1 1 2 2000

1 测试订单 5000 2 1 2 5 200

在es里该怎么玩儿，es里面的复杂的关联查询，复杂的查询语法，尽量别用，一旦用了性能一般都不太好

设计es里的数据模型

写入es的时候，搞成两个索引，order索引，orderItem索引

order索引，里面就包含id order\_code total\_price

orderItem索引，里面写入进去的时候，就完成join操作，id order\_code total\_price id order\_id goods\_id purchase\_count price

写入es的java系统里，就完成关联，将关联好的数据直接写入es中，搜索的时候，就不需要利用es的搜索语法去完成join来搜索了

document模型设计是非常重要的，很多操作，不要在搜索的时候才想去执行各种复杂的乱七八糟的操作。es能支持的操作就是那么多，不要考虑用es做一些它不好操作的事情。如果真的有那种操作，尽量在document模型设计的时候，写入的时候就完成。另外对于一些太复杂的操作，比如join，nested，parent-child搜索都要尽量避免，性能都很差的。

很多同学在问我，很多复杂的乱七八糟的一些操作，如何执行

两个思路，在搜索/查询的时候，要执行一些业务强相关的特别复杂的操作：

1）在写入数据的时候，就设计好模型，加几个字段，把处理好的数据写入加的字段里面

2）自己用java程序封装，es能做的，用es来做，搜索出来的数据，在java程序里面去做，比如说我们，基于es，用java封装一些特别复杂的操作

（5）分页性能优化

es的分页是较坑的，为啥呢？举个例子吧，假如你每页是10条数据，你现在要查询第100页，实际上是会把每个shard上存储的前1000条数据都查到一个协调节点上，如果你有个5个shard，那么就有5000条数据，接着协调节点对这5000条数据进行一些合并、处理，再获取到最终第100页的10条数据。

分布式的，你要查第100页的10条数据，你是不可能说从5个shard，每个shard就查2条数据？最后到协调节点合并成10条数据？你必须得从每个shard都查1000条数据过来，然后根据你的需求进行排序、筛选等等操作，最后再次分页，拿到里面第100页的数据。

你翻页的时候，翻的越深，每个shard返回的数据就越多，而且协调节点处理的时间越长。非常坑爹。所以用es做分页的时候，你会发现越翻到后面，就越是慢。

我们之前也是遇到过这个问题，用es作分页，前几页就几十毫秒，翻到10页之后，几十页的时候，基本上就要5~10秒才能查出来一页数据了

1）不允许深度分页/默认深度分页性能很惨

你系统不允许他翻那么深的页，pm，默认翻的越深，性能就越差

2）类似于app里的推荐商品不断下拉出来一页一页的

类似于微博中，下拉刷微博，刷出来一页一页的，你可以用scroll api，自己百度

scroll会一次性给你生成所有数据的一个快照，然后每次翻页就是通过游标移动，获取下一页下一页这样子，性能会比上面说的那种分页性能也高很多很多

针对这个问题，你可以考虑用scroll来进行处理，scroll的原理实际上是保留一个数据快照，然后在一定时间内，你如果不断的滑动往后翻页的时候，类似于你现在在浏览微博，不断往下刷新翻页。那么就用scroll不断通过游标获取下一页数据，这个性能是很高的，比es实际翻页要好的多的多。

但是唯一的一点就是，这个适合于那种类似微博下拉翻页的，不能随意跳到任何一页的场景。同时这个scroll是要保留一段时间内的数据快照的，你需要确保用户不会持续不断翻页翻几个小时。

无论翻多少页，性能基本上都是毫秒级的

因为scroll api是只能一页一页往后翻的，是不能说，先进入第10页，然后去120页，回到58页，不能随意乱跳页。所以现在很多产品，都是不允许你随意翻页的，app，也有一些网站，做的就是你只能往下拉，一页一页的翻

Day17

1、面试题

es生产集群的部署架构是什么？每个索引的数据量大概有多少？每个索引大概有多少个分片？

2、面试官心里分析

这个问题，包括后面的redis什么的，谈到es、redis、mysql分库分表等等技术，面试必问！就是你生产环境咋部署的？说白了，这个问题没啥技术含量，就是看你有没有在真正的生产环境里干过这事儿！

有些同学可能是没在生产环境中干过的，没实际去拿线上机器部署过es集群，也没实际玩儿过，也没往es集群里面导入过几千万甚至是几亿的数据量，可能你就不太清楚这里面的一些生产项目中的细节

如果你是自己就玩儿过demo，没碰过真实的es集群，那你可能此时会懵，但是别懵。。。你一定要云淡风轻的回答出来这个问题，表示你确实干过这事儿

3、面试题剖析

其实这个问题没啥，如果你确实干过es，那你肯定了解你们生产es集群的实际情况，部署了几台机器？有多少个索引？每个索引有多大数据量？每个索引给了多少个分片？你肯定知道！

但是如果你确实没干过，也别虚，我给你说一个基本的版本，你到时候就简单说一下就好了

（1）es生产集群我们部署了5台机器，每台机器是6核64G的，集群总内存是320G

（2）我们es集群的日增量数据大概是2000万条，每天日增量数据大概是500MB，每月增量数据大概是6亿，15G。目前系统已经运行了几个月，现在es集群里数据总量大概是100G左右。

（3）目前线上有5个索引（这个结合你们自己业务来，看看自己有哪些数据可以放es的），每个索引的数据量大概是20G，所以这个数据量之内，我们每个索引分配的是8个shard，比默认的5个shard多了3个shard。

大概就这么说一下就行了

Day18

消息队列、分布式搜索引擎

其实如果我是面试官的话，我如果感觉你都把刚才那些问题都答出来了，我可能会继续刨根问底，深挖，问你，直到把你给问倒

消息队列，kafka，复制的底层原理，leader选举的算法，增加partition以后的rebalance算法，扣很多很多的细节，如何优化kafka写入的吞吐量

其实这块如果挖深了可以问的极其深，如果是我来深挖，可能会挖到es底层的相关度评分算法（TF/IDF算法）、deep paging、上千万数据批处理、跨机房多集群同步、搜索效果优化，等等吧，很多的实际生产问题。

3种，一种比较水的面试官，他可能还掌握不到我们这个面试突击课程的水准，他其实连我们这个课程里的这些问题都没问到位；比较nice的面试官，问的差不多了，对你还挺欣赏的，基本上就让你过了；比较hard的面试官，干倒，虐你，故意要让你出丑，我不是这种，我只不过严格，我希望招到的是一个最好的这么一个人，我之所以会深挖深挖深挖，我想看看你的极限到底在哪里，你对这么技术掌握的最深的深度在哪儿？如果你进来成为为的下属，我就可以对你的能力各方面都非常的了解

但是。。。既然我们课程定位是2周快速突击，帮助的是那些原本可能连一些基础问题都没法答出来的同学去面试，而且确实现在很多公司面试官问到这块可能也就是问一些基础的问题，那么。。。我们就不忘初心吧

我唯一能说的，就是只要有人跟你聊到es，你可以自己合盘脱出自己对分布式搜索引擎基本原理的一个理解，以及你们在项目中一般是如何优化的，包括你们生产环境是怎么部署的，数据量多大。让别人感觉你这块至少还是正经了解和干过的。

Day19

1、面试题

在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？

2、面试官心里分析

这个问题，互联网公司必问，要是一个人连缓存都不太清楚，那确实比较尴尬

只要问到缓存，上来第一个问题，肯定能是先问问你项目哪里用了缓存？为啥要用？不用行不行？如果用了以后可能会有什么不良的后果？

这就是看看你对你用缓存这个东西背后，有没有思考，如果你就是傻乎乎的瞎用，没法给面试官一个合理的解答。那我只能说，面试官对你印象肯定不太好，觉得你平时思考太少，就知道干活儿。

3、面试题剖析

一个一个来看

（1）在项目中缓存是如何使用的？

这个，你结合你自己项目的业务来，你如果用了那恭喜你，你如果没用那不好意思，你硬加也得加一个场景吧

（2）为啥在项目里要用缓存呢？

用缓存，主要是俩用途，高性能和高并发

1）高性能

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作mysql，半天查出来一个结果，耗时600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？

缓存啊，折腾600ms查出来的结果，扔缓存里，一个key对应一个value，下次再有人查，别走mysql折腾600ms了。直接从缓存里，通过一个key查出来一个value，2ms搞定。性能提升300倍。

这就是所谓的高性能。

就是把你一些复杂操作耗时查出来的结果，如果确定后面不咋变了，然后但是马上还有很多读请求，那么直接结果放缓存，后面直接读缓存就好了。

2）高并发

mysql这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql单机支撑到2000qps也开始容易报警了。

所以要是你有个系统，高峰期一秒钟过来的请求有1万，那一个mysql单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放mysql。缓存功能简单，说白了就是key-value式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发so easy。单机承载并发量是mysql单机的几十倍。

3）所以你要结合这俩场景考虑一下，你为啥要用缓存？

一般很多同学项目里没啥高并发场景，那就别折腾了，直接用高性能那个场景吧，就思考有没有可以缓存结果的复杂查询场景，后续可以大幅度提升性能，优化用户体验，有，就说这个理由，没有？？那你也得编一个出来吧，不然你不是在搞笑么

（3）用了缓存之后会有啥不良的后果？

呵呵。。。你要是没考虑过这个问题，那你就尴尬了，面试官会觉得你头脑简单，四肢也不发达。你别光是傻用一个东西，多考虑考虑背后的一些事儿。

常见的缓存问题有仨（当然其实有很多，我这里就说仨，你能说出来也可以了）

1）缓存与数据库双写不一致

2）缓存雪崩

3）缓存穿透

4）缓存并发竞争

这仨问题是常见面试题，后面我要讲，大家看到后面自然就知道了，但是人要是问你，你至少自己能说出来，并且给出对应的解决方案

Day20

1、面试题

redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多（为什么redis是单线程的但是还可以支撑高并发）？

2、面试官心里分析

这个是问redis的时候，最基本的问题吧，redis最基本的一个内部原理和特点，就是redis实际上是个单线程工作模型，你要是这个都不知道，那后面玩儿redis的时候，出了问题岂不是什么都不知道？

还有可能面试官会问问你redis和memcached的区别，不过说实话，最近这两年，我作为面试官都不太喜欢这么问了，memched是早些年各大互联网公司常用的缓存方案，但是现在近几年基本都是redis，没什么公司用memcached了

3、额外的友情提示

同学，你要是现在还不知道redis和memcached是啥？那你赶紧百度一下redis入门和memcahced入门，简单启动一下，然后试一下几个简单操作，先感受一下。接着回来继续听课，我觉得1小时以内你就搞定了。

另外一个友情提示，要听明白redis的线程模型，你需要了解socket网络相关的基本知识，如果不懂。。。那我觉得你java没学好吧。初学者都该学习java的socket网络通信相关知识的。。。

4、面试题剖析

（1）redis和memcached有啥区别

这个事儿吧，你可以比较出N多个区别来，但是我还是采取redis作者给出的几个比较吧

1）Redis支持服务器端的数据操作：Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

~~2）内存使用效率对比：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。~~

~~3）性能对比：由于Redis只使用单核，而Memcached可以使用多核，所以平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。~~

4）集群模式：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是redis目前是原生支持cluster模式的，redis官方就是支持redis cluster集群模式的，比memcached来说要更好

（2）redis的线程模型

1）文件事件处理器

redis基于reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器，file event handler。这个文件事件处理器，是单线程的，redis才叫做单线程的模型，采用IO多路复用机制同时监听多个socket，根据socket上的事件来选择对应的事件处理器来处理这个事件。

如果被监听的socket准备好执行accept、read、write、close等操作的时候，跟操作对应的文件事件就会产生，这个时候文件事件处理器就会调用之前关联好的事件处理器来处理这个事件。

文件事件处理器是单线程模式运行的，但是通过IO多路复用机制监听多个socket，可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了redis内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个socket，IO多路复用程序，文件事件分派器，事件处理器（命令请求处理器、命令回复处理器、连接应答处理器，等等）。

多个socket可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个socket，但是会将socket放入一个队列中排队，每次从队列中取出一个socket给事件分派器，事件分派器把socket给对应的事件处理器。

然后一个socket的事件处理完之后，IO多路复用程序才会将队列中的下一个socket给事件分派器。文件事件分派器会根据每个socket当前产生的事件，来选择对应的事件处理器来处理。

2）文件事件

当socket变得可读时（比如客户端对redis执行write操作，或者close操作），或者有新的可以应答的sccket出现时（客户端对redis执行connect操作），socket就会产生一个AE\_READABLE事件。

当socket变得可写的时候（客户端对redis执行read操作），socket会产生一个AE\_WRITABLE事件。

IO多路复用程序可以同时监听AE\_REABLE和AE\_WRITABLE两种事件，要是一个socket同时产生了AE\_READABLE和AE\_WRITABLE两种事件，那么文件事件分派器优先处理AE\_REABLE事件，然后才是AE\_WRITABLE事件。

3）文件事件处理器

如果是客户端要连接redis，那么会为socket关联连接应答处理器

如果是客户端要写数据到redis，那么会为socket关联命令请求处理器

如果是客户端要从redis读数据，那么会为socket关联命令回复处理器

4）客户端与redis通信的一次流程

在redis启动初始化的时候，redis会将连接应答处理器跟AE\_READABLE事件关联起来，接着如果一个客户端跟redis发起连接，此时会产生一个AE\_READABLE事件，然后由连接应答处理器来处理跟客户端建立连接，创建客户端对应的socket，同时将这个socket的AE\_READABLE事件跟命令请求处理器关联起来。

当客户端向redis发起请求的时候（不管是读请求还是写请求，都一样），首先就会在socket产生一个AE\_READABLE事件，然后由对应的命令请求处理器来处理。这个命令请求处理器就会从socket中读取请求相关数据，然后进行执行和处理。

接着redis这边准备好了给客户端的响应数据之后，就会将socket的AE\_WRITABLE事件跟命令回复处理器关联起来，当客户端这边准备好读取响应数据时，就会在socket上产生一个AE\_WRITABLE事件，会由对应的命令回复处理器来处理，就是将准备好的响应数据写入socket，供客户端来读取。

命令回复处理器写完之后，就会删除这个socket的AE\_WRITABLE事件和命令回复处理器的关联关系。

（3）为啥redis单线程模型也能效率这么高？

1）纯内存操作

2）核心是基于非阻塞的IO多路复用机制

3）单线程反而避免了多线程的频繁上下文切换问题（百度）

Day21

1、面试题

redis都有哪些数据类型？分别在哪些场景下使用比较合适？

2、面试官心里分析

除非是我感觉看你简历，就是工作3年以内的比较初级的一个同学，可能对技术没有很深入的研究过，我才会问这类问题，在宝贵的面试时间里，我实在是不想多问

其实问这个问题呢。。。主要就俩原因

第一，看看你到底有没有全面的了解redis有哪些功能，一般怎么来用，啥场景用什么，就怕你别就会最简单的kv操作

第二，看看你在实际项目里都怎么玩儿过redis

要是你回答的不好，没说出几种数据类型，也没说什么场景，你完了，面试官对你印象肯定不好，觉得你平时就是做个简单的set和get。

3、面试题剖析

（1）string

这是最基本的类型了，没啥可说的，就是普通的set和get，做简单的kv缓存

（2）hash

这个是类似map的一种结构，这个一般就是可以将结构化的数据，比如一个对象（前提是这个对象没嵌套其他的对象）给缓存在redis里，然后每次读写缓存的时候，可以就操作hash里的某个字段。

key=150

value={

“id”: 150,

“name”: “zhangsan”,

“age”: 20

}

hash类的数据结构，主要是用来存放一些对象，把一些简单的对象给缓存起来，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值

value={

“id”: 150,

“name”: “zhangsan”,

“age”: 21

}

（3）list

有序列表，这个是可以玩儿出很多花样的

微博，某个大v的粉丝，就可以以list的格式放在redis里去缓存

key=某大v

value=[zhangsan, lisi, wangwu]

比如可以通过list存储一些列表型的数据结构，类似粉丝列表了、文章的评论列表了之类的东西

比如可以通过lrange命令，就是从某个元素开始读取多少个元素，可以基于list实现分页查询，这个很棒的一个功能，基于redis实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西，性能高，就一页一页走

比如可以搞个简单的消息队列，从list头怼进去，从list尾巴那里弄出来

（4）set

无序集合，自动去重

直接基于set将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于jvm内存里的HashSet进行去重，但是如果你的某个系统部署在多台机器上呢？

得基于redis进行全局的set去重

可以基于set玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁？对吧

把两个大v的粉丝都放在两个set中，对两个set做交集

（5）sorted set

排序的set，去重但是可以排序，写进去的时候给一个分数，自动根据分数排序，这个可以玩儿很多的花样，最大的特点是有个分数可以自定义排序规则

比如说你要是想根据时间对数据排序，那么可以写入进去的时候用某个时间作为分数，人家自动给你按照时间排序了

排行榜：将每个用户以及其对应的什么分数写入进去，zadd board score username，接着zrevrange board 0 99，就可以获取排名前100的用户；zrank board username，可以看到用户在排行榜里的排名

zadd board 85 zhangsan

zadd board 72 wangwu

zadd board 96 lisi

zadd board 62 zhaoliu

96 lisi

85 zhangsan

72 wangwu

62 zhaoliu

zrevrange board 0 3

获取排名前3的用户

96 lisi

85 zhangsan

72 wangwu

zrank board zhaoliu

4

Day22

1、面试题

redis的过期策略都有哪些？内存淘汰机制都有哪些？手写一下LRU代码实现？

2、面试官心里分析

1）老师啊，我往redis里写的数据怎么没了？

之前有同学问过我，说我们生产环境的redis怎么经常会丢掉一些数据？写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明redis你就没用对啊。redis是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个G的内存，但是可以有几个T的硬盘空间。redis主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如redis就只能用10个G，你要是往里面写了20个G的数据，会咋办？当然会干掉10个G的数据，然后就保留10个G的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

所以说，这是缓存的一个最基本的概念，数据是会过期的，要么是你自己设置个过期时间，要么是redis自己给干掉。

set key value 过期时间（1小时）

set进去的key，1小时之后就没了，就失效了

2）老师，我的数据明明都过期了，怎么还占用着内存啊？

还有一种就是如果你设置好了一个过期时间，你知道redis是怎么给你弄成过期的吗？什么时候删除掉？如果你不知道，之前有个学员就问了，为啥好多数据明明应该过期了，结果发现redis内存占用还是很高？那是因为你不知道redis是怎么删除那些过期key的。

redis 内存一共是10g，你现在往里面写了5g的数据，结果这些数据明明你都设置了过期时间，要求这些数据1小时之后都会过期，结果1小时之后，你回来一看，redis机器，怎么内存占用还是50%呢？5g数据过期了，我从redis里查，是查不到了，结果过期的数据还占用着redis的内存。

如果你连这个问题都不知道，上来就懵了，回答不出来，那线上你写代码的时候，想当然的认为写进redis的数据就一定会存在，后面导致系统各种漏洞和bug，谁来负责？

3、面试题剖析

（1）设置过期时间

我们set key的时候，都可以给一个expire time，就是过期时间，指定这个key比如说只能存活1个小时？10分钟？这个很有用，我们自己可以指定缓存到期就失效。

如果假设你设置一个一批key只能存活1个小时，那么接下来1小时后，redis是怎么对这批key进行删除的？

答案是：定期删除+惰性删除

所谓定期删除，指的是redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。假设redis里放了10万个key，都设置了过期时间，你每隔几百毫秒，就检查10万个key，那redis基本上就死了，cpu负载会很高的，消耗在你的检查过期key上了。注意，这里可不是每隔100ms就遍历所有的设置过期时间的key，那样就是一场性能上的灾难。实际上redis是每隔100ms随机抽取一些key来检查和删除的。

但是问题是，定期删除可能会导致很多过期key到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，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，不会给你返回任何东西。

并不是key到时间就被删除掉，而是你查询这个key的时候，redis再懒惰的检查一下

通过上述两种手段结合起来，保证过期的key一定会被干掉。

很简单，就是说，你的过期key，靠定期删除没有被删除掉，还停留在内存里，占用着你的内存呢，除非你的系统去查一下那个key，才会被redis给删除掉。

但是实际上这还是有问题的，如果定期删除漏掉了很多过期key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期key堆积在内存里，导致redis内存块耗尽了，咋整？

答案是：走内存淘汰机制。

（2）内存淘汰

如果redis的内存占用过多的时候，此时会进行内存淘汰，有如下一些策略：

redis 10个key，现在已经满了，redis需要删除掉5个key

1个key，最近1分钟被查询了100次

1个key，最近10分钟被查询了50次

1个key，最近1个小时倍查询了1次

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了

2）allkeys-lru：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（这个是最常用的）

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，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的key给干掉啊

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（这个一般不太合适）

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

百度，问题啊，网上鱼龙混杂

如果百度一些api操作，入门的知识，ok的，随便找一个博客都可以

一些高级别的，redis单线程模型

很简单，你写的数据太多，内存满了，或者触发了什么条件，redis lru，自动给你清理掉了一些最近很少使用的数据

（3）要不你手写一个LRU算法？

我确实有时会问这个，因为有些候选人如果确实过五关斩六将，前面的问题都答的很好，那么其实让他写一下LRU算法，可以考察一下编码功底

你可以现场手写最原始的LRU算法，那个代码量太大了，我觉得不太现实

public class LRUCache<K, V> extends LinkedHashMap<K, V> {

private final int CACHE\_SIZE;

// 这里就是传递进来最多能缓存多少数据

public LRUCache(int cacheSize) {

super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true); // 这块就是设置一个hashmap的初始大小，同时最后一个true指的是让linkedhashmap按照访问顺序来进行排序，最近访问的放在头，最老访问的就在尾

CACHE\_SIZE = cacheSize;

}

@Override

protected boolean removeEldestEntry(Map.Entry eldest) {

return size() > CACHE\_SIZE; // 这个意思就是说当map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据

}

}

我给你看上面的代码，是告诉你最起码你也得写出来上面那种代码，不求自己纯手工从底层开始打造出自己的LRU，但是起码知道如何利用已有的jdk数据结构实现一个java版的LRU

Day23

1、面试题

如何保证Redis的高并发和高可用？redis的主从复制原理能介绍一下么？redis的哨兵原理能介绍一下么？

2、面试官心里分析

其实问这个问题，主要是考考你，redis单机能承载多高并发？如果单机扛不住如何扩容抗更多的并发？redis会不会挂？既然redis会挂那怎么保证redis是高可用的？

其实针对的都是项目中你肯定要考虑的一些问题，如果你没考虑过，那确实你对生产系统中的问题思考太少。

3、面试题剖析

就是如果你用redis缓存技术的话，肯定要考虑如何用redis来加多台机器，保证redis是高并发的，还有就是如何让Redis保证自己不是挂掉以后就直接死掉了，redis高可用

我这里会选用我之前讲解过这一块内容，redis高并发、高可用、缓存一致性

redis高并发：主从架构，一主多从，一般来说，很多项目其实就足够了，单主用来写入数据，单机几万QPS，多从用来查询数据，多个从实例可以提供每秒10万的QPS。

redis高并发的同时，还需要容纳大量的数据：一主多从，每个实例都容纳了完整的数据，比如redis主就10G的内存量，其实你就最对只能容纳10g的数据量。如果你的缓存要容纳的数据量很大，达到了几十g，甚至几百g，或者是几t，那你就需要redis集群，而且用redis集群之后，可以提供可能每秒几十万的读写并发。

redis高可用：如果你做主从架构部署，其实就是加上哨兵就可以了，就可以实现，任何一个实例宕机，自动会进行主备切换。

1、redis高并发跟整个系统的高并发之间的关系

redis，你要搞高并发的话，不可避免，要把底层的缓存搞得很好

mysql，高并发，做到了，那么也是通过一系列复杂的分库分表，订单系统，事务要求的，QPS到几万，比较高了

要做一些电商的商品详情页，真正的超高并发，QPS上十万，甚至是百万，一秒钟百万的请求量

光是redis是不够的，但是redis是整个大型的缓存架构中，支撑高并发的架构里面，非常重要的一个环节

首先，你的底层的缓存中间件，缓存系统，必须能够支撑的起我们说的那种高并发，其次，再经过良好的整体的缓存架构的设计（多级缓存架构、热点缓存），支撑真正的上十万，甚至上百万的高并发

2、redis不能支撑高并发的瓶颈在哪里？

单机

3、如果redis要支撑超过10万+的并发，那应该怎么做？

单机的redis几乎不太可能说QPS超过10万+，除非一些特殊情况，比如你的机器性能特别好，配置特别高，物理机，维护做的特别好，而且你的整体的操作不是太复杂

单机在几万

读写分离，一般来说，对缓存，一般都是用来支撑读高并发的，写的请求是比较少的，可能写请求也就一秒钟几千，一两千

大量的请求都是读，一秒钟二十万次读

读写分离

主从架构 -> 读写分离 -> 支撑10万+读QPS的架构

4、接下来要讲解的一个topic

redis replication

redis主从架构 -> 读写分离架构 -> 可支持水平扩展的读高并发架构

课程大纲

1、图解redis replication基本原理

2、redis replication的核心机制

3、master持久化对于主从架构的安全保障的意义

redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

redis replication的最最基本的原理，铺垫

------------------------------------------------------------------------

1、图解redis replication基本原理

------------------------------------------------------------------------

2、redis replication的核心机制

（1）redis采用异步方式复制数据到slave节点，不过redis 2.8开始，slave node会周期性地确认自己每次复制的数据量

（2）一个master node是可以配置多个slave node的

（3）slave node也可以连接其他的slave node

（4）slave node做复制的时候，是不会block master node的正常工作的

（5）slave node在做复制的时候，也不会block对自己的查询操作，它会用旧的数据集来提供服务; 但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了

（6）slave node主要用来进行横向扩容，做读写分离，扩容的slave node可以提高读的吞吐量

slave，高可用性，有很大的关系

------------------------------------------------------------------------

3、master持久化对于主从架构的安全保障的意义

如果采用了主从架构，那么建议必须开启master node的持久化！

不建议用slave node作为master node的数据热备，因为那样的话，如果你关掉master的持久化，可能在master宕机重启的时候数据是空的，然后可能一经过复制，salve node数据也丢了

master -> RDB和AOF都关闭了 -> 全部在内存中

master宕机，重启，是没有本地数据可以恢复的，然后就会直接认为自己IDE数据是空的

master就会将空的数据集同步到slave上去，所有slave的数据全部清空

100%的数据丢失

master节点，必须要使用持久化机制

第二个，master的各种备份方案，要不要做，万一说本地的所有文件丢失了; 从备份中挑选一份rdb去恢复master; 这样才能确保master启动的时候，是有数据的

即使采用了后续讲解的高可用机制，slave node可以自动接管master node，但是也可能sentinal还没有检测到master failure，master node就自动重启了，还是可能导致上面的所有slave node数据清空故障

课程大纲

1、主从架构的核心原理

当启动一个slave node的时候，它会发送一个PSYNC命令给master node

如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据; 否则如果是slave node第一次连接master node，那么会触发一次full resynchronization

开始full resynchronization的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

2、主从复制的断点续传

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中常见一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次resynchronization

3、无磁盘化复制

master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了

repl-diskless-sync

repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

4、过期key处理

slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。

1、复制的完整流程

（1）slave node启动，仅仅保存master node的信息，包括master node的host和ip，但是复制流程没开始

master host和ip是从哪儿来的，redis.conf里面的slaveof配置的

（2）slave node内部有个定时任务，每秒检查是否有新的master node要连接和复制，如果发现，就跟master node建立socket网络连接

（3）slave node发送ping命令给master node

（4）口令认证，如果master设置了requirepass，那么salve node必须发送masterauth的口令过去进行认证

（5）master node第一次执行全量复制，将所有数据发给slave node

（6）master node后续持续将写命令，异步复制给slave node

2、数据同步相关的核心机制

指的就是第一次slave连接msater的时候，执行的全量复制，那个过程里面你的一些细节的机制

（1）master和slave都会维护一个offset

master会在自身不断累加offset，slave也会在自身不断累加offset

slave每秒都会上报自己的offset给master，同时master也会保存每个slave的offset

这个倒不是说特定就用在全量复制的，主要是master和slave都要知道各自的数据的offset，才能知道互相之间的数据不一致的情况

（2）backlog

master node有一个backlog，默认是1MB大小

master node给slave node复制数据时，也会将数据在backlog中同步写一份

backlog主要是用来做全量复制中断候的增量复制的

（3）master run id

info server，可以看到master run id

如果根据host+ip定位master node，是不靠谱的，如果master node重启或者数据出现了变化，那么slave node应该根据不同的run id区分，run id不同就做全量复制

如果需要不更改run id重启redis，可以使用redis-cli debug reload命令

（4）psync

从节点使用psync从master node进行复制，psync runid offset

master node会根据自身的情况返回响应信息，可能是FULLRESYNC runid offset触发全量复制，可能是CONTINUE触发增量复制

3、全量复制

（1）master执行bgsave，在本地生成一份rdb快照文件

（2）master node将rdb快照文件发送给salve node，如果rdb复制时间超过60秒（repl-timeout），那么slave node就会认为复制失败，可以适当调节大这个参数

（3）对于千兆网卡的机器，一般每秒传输100MB，6G文件，很可能超过60s

（4）master node在生成rdb时，会将所有新的写命令缓存在内存中，在salve node保存了rdb之后，再将新的写命令复制给salve node

（5）client-output-buffer-limit slave 256MB 64MB 60，如果在复制期间，内存缓冲区持续消耗超过64MB，或者一次性超过256MB，那么停止复制，复制失败

（6）slave node接收到rdb之后，清空自己的旧数据，然后重新加载rdb到自己的内存中，同时基于旧的数据版本对外提供服务

（7）如果slave node开启了AOF，那么会立即执行BGREWRITEAOF，重写AOF

rdb生成、rdb通过网络拷贝、slave旧数据的清理、slave aof rewrite，很耗费时间

如果复制的数据量在4G~6G之间，那么很可能全量复制时间消耗到1分半到2分钟

4、增量复制

（1）如果全量复制过程中，master-slave网络连接断掉，那么salve重新连接master时，会触发增量复制

（2）master直接从自己的backlog中获取部分丢失的数据，发送给slave node，默认backlog就是1MB

（3）msater就是根据slave发送的psync中的offset来从backlog中获取数据的

5、heartbeat

主从节点互相都会发送heartbeat信息

master默认每隔10秒发送一次heartbeat，salve node每隔1秒发送一个heartbeat

6、异步复制

master每次接收到写命令之后，现在内部写入数据，然后异步发送给slave node

1、什么是99.99%高可用？

架构上，高可用性，99.99%的高可用性

讲的学术，99.99%，公式，系统可用的时间 / 系统故障的时间，365天，在365天 \* 99.99%的时间内，你的系统都是可以哗哗对外提供服务的，那就是高可用性，99.99%

系统可用的时间 / 总的时间 = 高可用性，然后会对各种时间的概念，说一大堆解释

2、redis不可用是什么？单实例不可用？主从架构不可用？不可用的后果是什么？

3、redis怎么才能做到高可用？

1、哨兵的介绍

sentinal，中文名是哨兵

哨兵是redis集群架构中非常重要的一个组件，主要功能如下

（1）集群监控，负责监控redis master和slave进程是否正常工作

（2）消息通知，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员

（3）故障转移，如果master node挂掉了，会自动转移到slave node上

（4）配置中心，如果故障转移发生了，通知client客户端新的master地址

哨兵本身也是分布式的，作为一个哨兵集群去运行，互相协同工作

（1）故障转移时，判断一个master node是宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题

（2）即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了

目前采用的是sentinal 2版本，sentinal 2相对于sentinal 1来说，重写了很多代码，主要是让故障转移的机制和算法变得更加健壮和简单

2、哨兵的核心知识

（1）哨兵至少需要3个实例，来保证自己的健壮性

（2）哨兵 + redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性

（3）对于哨兵 + redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练

3、为什么redis哨兵集群只有2个节点无法正常工作？

哨兵集群必须部署2个以上节点

如果哨兵集群仅仅部署了个2个哨兵实例，quorum=1

+----+ +----+

| M1 |---------| R1 |

| S1 | | S2 |

+----+ +----+

Configuration: quorum = 1

master宕机，s1和s2中只要有1个哨兵认为master宕机就可以还行切换，同时s1和s2中会选举出一个哨兵来执行故障转移

同时这个时候，需要majority，也就是大多数哨兵都是运行的，2个哨兵的majority就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移

但是如果整个M1和S1运行的机器宕机了，那么哨兵只有1个了，此时就没有majority来允许执行故障转移，虽然另外一台机器还有一个R1，但是故障转移不会执行

4、经典的3节点哨兵集群

+----+

| M1 |

| S1 |

+----+

|

+----+ | +----+

| R2 |----+----| R3 |

| S2 | | S3 |

+----+ +----+

Configuration: quorum = 2，majority

如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移

同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移

课程大纲

1、两种数据丢失的情况

2、解决异步复制和脑裂导致的数据丢失

------------------------------------------------------------------

1、两种数据丢失的情况

主备切换的过程，可能会导致数据丢失

（1）异步复制导致的数据丢失

因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这些部分数据就丢失了

（2）脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着

此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master

这个时候，集群里就会有两个master，也就是所谓的脑裂

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了

因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据

------------------------------------------------------------------

2、解决异步复制和脑裂导致的数据丢失

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

上面两个配置可以减少异步复制和脑裂导致的数据丢失

（1）减少异步复制的数据丢失

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失

上面的配置就确保了，如果跟任何一个slave丢了连接，在10秒后发现没有slave给自己ack，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

1、sdown和odown转换机制

sdown和odown两种失败状态

sdown是主观宕机，就一个哨兵如果自己觉得一个master宕机了，那么就是主观宕机

odown是客观宕机，如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机

sdown达成的条件很简单，如果一个哨兵ping一个master，超过了is-master-down-after-milliseconds指定的毫秒数之后，就主观认为master宕机

sdown到odown转换的条件很简单，如果一个哨兵在指定时间内，收到了quorum指定数量的其他哨兵也认为那个master是sdown了，那么就认为是odown了，客观认为master宕机

2、哨兵集群的自动发现机制

哨兵互相之间的发现，是通过redis的pub/sub系统实现的，每个哨兵都会往\_\_sentinel\_\_:hello这个channel里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在

每隔两秒钟，每个哨兵都会往自己监控的某个master+slaves对应的\_\_sentinel\_\_:hello channel里发送一个消息，内容是自己的host、ip和runid还有对这个master的监控配置

每个哨兵也会去监听自己监控的每个master+slaves对应的\_\_sentinel\_\_:hello channel，然后去感知到同样在监听这个master+slaves的其他哨兵的存在

每个哨兵还会跟其他哨兵交换对master的监控配置，互相进行监控配置的同步

3、slave配置的自动纠正

哨兵会负责自动纠正slave的一些配置，比如slave如果要成为潜在的master候选人，哨兵会确保slave在复制现有master的数据; 如果slave连接到了一个错误的master上，比如故障转移之后，那么哨兵会确保它们连接到正确的master上

4、slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来

会考虑slave的一些信息

（1）跟master断开连接的时长

（2）slave优先级

（3）复制offset

（4）run id

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master

(down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state

接下来会对slave进行排序

（1）按照slave优先级进行排序，slave priority越低，优先级就越高

（2）如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高

（3）如果上面两个条件都相同，那么选择一个run id比较小的那个slave

5、quorum和majority

每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换

如果quorum < majority，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换

但是如果quorum >= majority，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换

6、configuration epoch

哨兵会对一套redis master+slave进行监控，有相应的监控的配置

执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch，作为新的version号

7、configuraiton传播

哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制

这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的

其他的哨兵都是根据版本号的大小来更新自己的master配置的

Day24

1、面试题

redis的持久化有哪几种方式？不同的持久化机制都有什么优缺点？持久化机制具体底层是如何实现的？

2、面试官心里分析

redis如果仅仅只是将数据缓存在内存里面，如果redis宕机了，再重启，内存里的数据就全部都弄丢了啊。。。。。。你必须得用redis的持久化机制，将数据写入内存的同时，异步的慢慢的将数据写入磁盘文件里，进行持久化

如果redis宕机了，重启启动，自动从磁盘上加载之前持久化的一些数据，就可以了，也许会丢失少许数据，但是至少不会将所有数据都弄丢

这个其实一样，针对的都是redis的生产环境可能遇到的一些问题，就是redis要是挂了再重启，内存里的数据不就全丢了？能不能重启的时候把数据给恢复了？

3、面试题剖析

课程大纲

1、故障发生的时候会怎么样

2、如何应对故障的发生

很多同学，自己也看过一些redis的资料和书籍，当然可能也看过一些redis视频课程

所有的资料，其实都会讲解redis持久化，但是有个问题，我到目前为止，没有看到有人很仔细的去讲解，redis的持久化意义

redis的持久化，RDB，AOF，区别，各自的特点是什么，适合什么场景

redis的企业级的持久化方案是什么，是用来跟哪些企业级的场景结合起来使用的？？？

redis持久化的意义，在于故障恢复

比如你部署了一个redis，作为cache缓存，当然也可以保存一些较为重要的数据

如果没有持久化的话，redis遇到灾难性故障的时候，就会丢失所有的数据

如果通过持久化将数据搞一份儿在磁盘上去，然后定期比如说同步和备份到一些云存储服务上去，那么就可以保证数据不丢失全部，还是可以恢复一部分数据回来的

课程大纲

1、RDB和AOF两种持久化机制的介绍

2、RDB持久化机制的优点

3、RDB持久化机制的缺点

4、AOF持久化机制的优点

5、AOF持久化机制的缺点

6、RDB和AOF到底该如何选择

我们已经知道对于一个企业级的redis架构来说，持久化是不可减少的

企业级redis集群架构：海量数据、高并发、高可用

持久化主要是做灾难恢复，数据恢复，也可以归类到高可用的一个环节里面去

比如你redis整个挂了，然后redis就不可用了，你要做的事情是让redis变得可用，尽快变得可用

重启redis，尽快让它对外提供服务，但是就像上一讲说，如果你没做数据备份，这个时候redis启动了，也不可用啊，数据都没了

很可能说，大量的请求过来，缓存全部无法命中，在redis里根本找不到数据，这个时候就死定了，缓存雪崩问题，所有请求，没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了

mysql挂掉，你都没法去找数据恢复到redis里面去，redis的数据从哪儿来？从mysql来。。。

具体的完整的缓存雪崩的场景，还有企业级的解决方案，到后面讲

如果你把redis的持久化做好，备份和恢复方案做到企业级的程度，那么即使你的redis故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务

redis的持久化，跟高可用，是有关系的，企业级redis架构中去讲解

redis持久化：RDB，AOF

-------------------------------------------------------------------------------------

1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

-------------------------------------------------------------------------------------

2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

-------------------------------------------------------------------------------------

3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒

-------------------------------------------------------------------------------------

4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据

-------------------------------------------------------------------------------------

5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

-------------------------------------------------------------------------------------

6、RDB和AOF到底该如何选择

（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

（2）也不要仅仅使用AOF，因为那样有两个问题，第一，你通过AOF做冷备，没有RDB做冷备，来的恢复速度更快; 第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug

（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复

课程大纲

1、RDB和AOF两种持久化机制的介绍

2、RDB持久化机制的优点

3、RDB持久化机制的缺点

4、AOF持久化机制的优点

5、AOF持久化机制的缺点

6、RDB和AOF到底该如何选择

我们已经知道对于一个企业级的redis架构来说，持久化是不可减少的

企业级redis集群架构：海量数据、高并发、高可用

持久化主要是做灾难恢复，数据恢复，也可以归类到高可用的一个环节里面去

比如你redis整个挂了，然后redis就不可用了，你要做的事情是让redis变得可用，尽快变得可用

重启redis，尽快让它对外提供服务，但是就像上一讲说，如果你没做数据备份，这个时候redis启动了，也不可用啊，数据都没了

很可能说，大量的请求过来，缓存全部无法命中，在redis里根本找不到数据，这个时候就死定了，缓存雪崩问题，所有请求，没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了

mysql挂掉，你都没法去找数据恢复到redis里面去，redis的数据从哪儿来？从mysql来。。。

具体的完整的缓存雪崩的场景，还有企业级的解决方案，到后面讲

如果你把redis的持久化做好，备份和恢复方案做到企业级的程度，那么即使你的redis故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务

redis的持久化，跟高可用，是有关系的，企业级redis架构中去讲解

redis持久化：RDB，AOF

-------------------------------------------------------------------------------------

1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

-------------------------------------------------------------------------------------

2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

RDB也可以做冷备，生成多个文件，每个文件都代表了某一个时刻的完整的数据快照

AOF也可以做冷备，只有一个文件，但是你可以，每隔一定时间，去copy一份这个文件出来

RDB做冷备，优势在哪儿呢？由redis去控制固定时长生成快照文件的事情，比较方便; AOF，还需要自己写一些脚本去做这个事情，各种定时

RDB数据做冷备，在最坏的情况下，提供数据恢复的时候，速度比AOF快

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

RDB，每次写，都是直接写redis内存，只是在一定的时候，才会将数据写入磁盘中

AOF，每次都是要写文件的，虽然可以快速写入os cache中，但是还是有一定的时间开销的,速度肯定比RDB略慢一些

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

AOF，存放的指令日志，做数据恢复的时候，其实是要回放和执行所有的指令日志，来恢复出来内存中的所有数据的

RDB，就是一份数据文件，恢复的时候，直接加载到内存中即可

结合上述优点，RDB特别适合做冷备份，冷备

-------------------------------------------------------------------------------------

3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据

这个问题，也是rdb最大的缺点，就是不适合做第一优先的恢复方案，如果你依赖RDB做第一优先恢复方案，会导致数据丢失的比较多

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒

一般不要让RDB的间隔太长，否则每次生成的RDB文件太大了，对redis本身的性能可能会有影响的

-------------------------------------------------------------------------------------

4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

每隔1秒，就执行一次fsync操作，保证os cache中的数据写入磁盘中

redis进程挂了，最多丢掉1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据

-------------------------------------------------------------------------------------

5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

如果你要保证一条数据都不丢，也是可以的，AOF的fsync设置成没写入一条数据，fsync一次，那就完蛋了，redis的QPS大降

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

（4）唯一的比较大的缺点，其实就是做数据恢复的时候，会比较慢，还有做冷备，定期的备份，不太方便，可能要自己手写复杂的脚本去做，做冷备不太合适

-------------------------------------------------------------------------------------

6、RDB和AOF到底该如何选择

（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

（2）也不要仅仅使用AOF，因为那样有两个问题，第一，你通过AOF做冷备，没有RDB做冷备，来的恢复速度更快; 第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug

（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复

Day25

1、面试题

redis集群模式的工作原理能说一下么？在集群模式下，redis的key是如何寻址的？分布式寻址都有哪些算法？了解一致性hash算法吗？

2、面试官心里分析

在以前，如果前几年的时候，一般来说，redis如果要搞几个节点，每个节点存储一部分的数据，得借助一些中间件来实现，比如说有codis，或者twemproxy，都有。有一些redis中间件，你读写redis中间件，redis中间件负责将你的数据分布式存储在多台机器上的redis实例中。

这两年，redis不断在发展，redis也不断的有新的版本，redis cluster，redis集群模式，你可以做到在多台机器上，部署多个redis实例，每个实例存储一部分的数据，同时每个redis实例可以挂redis从实例，自动确保说，如果redis主实例挂了，会自动切换到redis从实例顶上来。

现在redis的新版本，大家都是用redis cluster的，也就是redis原生支持的redis集群模式，那么面试官肯定会就redis cluster对你来个几连炮。要是你没用过redis cluster，正常，以前很多人用codis之类的客户端来支持集群，但是起码你得研究一下redis cluster吧。

3、面试题剖析

1、单机redis在海量数据面前的瓶颈

2、怎么才能够突破单机瓶颈，让redis支撑海量数据？

3、redis的集群架构

redis cluster

支撑N个redis master node，每个master node都可以挂载多个slave node

读写分离的架构，对于每个master来说，写就写到master，然后读就从mater对应的slave去读

高可用，因为每个master都有salve节点，那么如果mater挂掉，redis cluster这套机制，就会自动将某个slave切换成master

redis cluster（多master + 读写分离 + 高可用）

我们只要基于redis cluster去搭建redis集群即可，不需要手工去搭建replication复制+主从架构+读写分离+哨兵集群+高可用

4、redis cluster vs. replication + sentinal

如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个G，单机足够了

replication，一个mater，多个slave，要几个slave跟你的要求的读吞吐量有关系，然后自己搭建一个sentinal集群，去保证redis主从架构的高可用性，就可以了

redis cluster，主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用redis cluster

讲解分布式数据存储的核心算法，数据分布的算法

hash算法 -> 一致性hash算法（memcached） -> redis cluster，hash slot算法

用不同的算法，就决定了在多个master节点的时候，数据如何分布到这些节点上去，解决这个问题

1、redis cluster介绍

redis cluster

（1）自动将数据进行分片，每个master上放一部分数据

（2）提供内置的高可用支持，部分master不可用时，还是可以继续工作的

在redis cluster架构下，每个redis要放开两个端口号，比如一个是6379，另外一个就是加10000的端口号，比如16379

16379端口号是用来进行节点间通信的，也就是cluster bus的东西，集群总线。cluster bus的通信，用来进行故障检测，配置更新，故障转移授权

cluster bus用了另外一种二进制的协议，主要用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间

2、最老土的hash算法和弊端（大量缓存重建）

3、一致性hash算法（自动缓存迁移）+虚拟节点（自动负载均衡）

4、redis cluster的hash slot算法

redis cluster有固定的16384个hash slot，对每个key计算CRC16值，然后对16384取模，可以获取key对应的hash slot

redis cluster中每个master都会持有部分slot，比如有3个master，那么可能每个master持有5000多个hash slot

hash slot让node的增加和移除很简单，增加一个master，就将其他master的hash slot移动部分过去，减少一个master，就将它的hash slot移动到其他master上去

移动hash slot的成本是非常低的

客户端的api，可以对指定的数据，让他们走同一个hash slot，通过hash tag来实现

一、节点间的内部通信机制

1、基础通信原理

（1）redis cluster节点间采取gossip协议进行通信

跟集中式不同，不是将集群元数据（节点信息，故障，等等）集中存储在某个节点上，而是互相之间不断通信，保持整个集群所有节点的数据是完整的

维护集群的元数据用得，集中式，一种叫做gossip

集中式：好处在于，元数据的更新和读取，时效性非常好，一旦元数据出现了变更，立即就更新到集中式的存储中，其他节点读取的时候立即就可以感知到; 不好在于，所有的元数据的跟新压力全部集中在一个地方，可能会导致元数据的存储有压力

gossip：好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，有一定的延时，降低了压力; 缺点，元数据更新有延时，可能导致集群的一些操作会有一些滞后

我们刚才做reshard，去做另外一个操作，会发现说，configuration error，达成一致

（2）10000端口

每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是17001端口

每隔节点每隔一段时间都会往另外几个节点发送ping消息，同时其他几点接收到ping之后返回pong

（3）交换的信息

故障信息，节点的增加和移除，hash slot信息，等等

2、gossip协议

gossip协议包含多种消息，包括ping，pong，meet，fail，等等

meet: 某个节点发送meet给新加入的节点，让新节点加入集群中，然后新节点就会开始与其他节点进行通信

redis-trib.rb add-node

其实内部就是发送了一个gossip meet消息，给新加入的节点，通知那个节点去加入我们的集群

ping: 每个节点都会频繁给其他节点发送ping，其中包含自己的状态还有自己维护的集群元数据，互相通过ping交换元数据

每个节点每秒都会频繁发送ping给其他的集群，ping，频繁的互相之间交换数据，互相进行元数据的更新

pong: 返回ping和meet，包含自己的状态和其他信息，也可以用于信息广播和更新

fail: 某个节点判断另一个节点fail之后，就发送fail给其他节点，通知其他节点，指定的节点宕机了

3、ping消息深入

ping很频繁，而且要携带一些元数据，所以可能会加重网络负担

每个节点每秒会执行10次ping，每次会选择5个最久没有通信的其他节点

当然如果发现某个节点通信延时达到了cluster\_node\_timeout / 2，那么立即发送ping，避免数据交换延时过长，落后的时间太长了

比如说，两个节点之间都10分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题

所以cluster\_node\_timeout可以调节，如果调节比较大，那么会降低发送的频率

每次ping，一个是带上自己节点的信息，还有就是带上1/10其他节点的信息，发送出去，进行数据交换

至少包含3个其他节点的信息，最多包含总节点-2个其他节点的信息

-------------------------------------------------------------------------------------------------------

二、面向集群的jedis内部实现原理

开发，jedis，redis的java client客户端，redis cluster，jedis cluster api

jedis cluster api与redis cluster集群交互的一些基本原理

1、基于重定向的客户端

redis-cli -c，自动重定向

（1）请求重定向

客户端可能会挑选任意一个redis实例去发送命令，每个redis实例接收到命令，都会计算key对应的hash slot

如果在本地就在本地处理，否则返回moved给客户端，让客户端进行重定向

cluster keyslot mykey，可以查看一个key对应的hash slot是什么

用redis-cli的时候，可以加入-c参数，支持自动的请求重定向，redis-cli接收到moved之后，会自动重定向到对应的节点执行命令

（2）计算hash slot

计算hash slot的算法，就是根据key计算CRC16值，然后对16384取模，拿到对应的hash slot

用hash tag可以手动指定key对应的slot，同一个hash tag下的key，都会在一个hash slot中，比如set mykey1:{100}和set mykey2:{100}

（3）hash slot查找

节点间通过gossip协议进行数据交换，就知道每个hash slot在哪个节点上

2、smart jedis

（1）什么是smart jedis

基于重定向的客户端，很消耗网络IO，因为大部分情况下，可能都会出现一次请求重定向，才能找到正确的节点

所以大部分的客户端，比如java redis客户端，就是jedis，都是smart的

本地维护一份hashslot -> node的映射表，缓存，大部分情况下，直接走本地缓存就可以找到hashslot -> node，不需要通过节点进行moved重定向

（2）JedisCluster的工作原理

在JedisCluster初始化的时候，就会随机选择一个node，初始化hashslot -> node映射表，同时为每个节点创建一个JedisPool连接池

每次基于JedisCluster执行操作，首先JedisCluster都会在本地计算key的hashslot，然后在本地映射表找到对应的节点

如果那个node正好还是持有那个hashslot，那么就ok; 如果说进行了reshard这样的操作，可能hashslot已经不在那个node上了，就会返回moved

如果JedisCluter API发现对应的节点返回moved，那么利用该节点的元数据，更新本地的hashslot -> node映射表缓存

重复上面几个步骤，直到找到对应的节点，如果重试超过5次，那么就报错，JedisClusterMaxRedirectionException

jedis老版本，可能会出现在集群某个节点故障还没完成自动切换恢复时，频繁更新hash slot，频繁ping节点检查活跃，导致大量网络IO开销

jedis最新版本，对于这些过度的hash slot更新和ping，都进行了优化，避免了类似问题

（3）hashslot迁移和ask重定向

如果hash slot正在迁移，那么会返回ask重定向给jedis

jedis接收到ask重定向之后，会重新定位到目标节点去执行，但是因为ask发生在hash slot迁移过程中，所以JedisCluster API收到ask是不会更新hashslot本地缓存

已经可以确定说，hashslot已经迁移完了，moved是会更新本地hashslot->node映射表缓存的

-------------------------------------------------------------------------------------------------------

三、高可用性与主备切换原理

redis cluster的高可用的原理，几乎跟哨兵是类似的

1、判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是pfail，主观宕机

如果多个节点都认为另外一个节点宕机了，那么就是fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown

在cluster-node-timeout内，某个节点一直没有返回pong，那么就被认为pfail

如果一个节点认为某个节点pfail了，那么会在gossip ping消息中，ping给其他节点，如果超过半数的节点都认为pfail了，那么就会变成fail

2、从节点过滤

对宕机的master node，从其所有的slave node中，选择一个切换成master node

检查每个slave node与master node断开连接的时间，如果超过了cluster-node-timeout \* cluster-slave-validity-factor，那么就没有资格切换成master

这个也是跟哨兵是一样的，从节点超时过滤的步骤

3、从节点选举

哨兵：对所有从节点进行排序，slave priority，offset，run id

每个从节点，都根据自己对master复制数据的offset，来设置一个选举时间，offset越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举

所有的master node开始slave选举投票，给要进行选举的slave进行投票，如果大部分master node（N/2 + 1）都投票给了某个从节点，那么选举通过，那个从节点可以切换成master

从节点执行主备切换，从节点切换为主节点

4、与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，redis cluster功能强大，直接集成了replication和sentinal的功能

没有办法去给大家深入讲解redis底层的设计的细节，核心原理和设计的细节，那个除非单独开一门课，redis底层原理深度剖析，redis源码

对于咱们这个架构课来说，主要关注的是架构，不是底层的细节，对于架构来说，核心的原理的基本思路，是要梳理清晰的

Day26

1、面试题

了解什么是redis的雪崩和穿透？redis崩溃之后会怎么样？系统该如何应对这种情况？如何处理redis的穿透？

2、面试官心里分析

其实这是问到缓存必问的，因为缓存雪崩和穿透，那是缓存最大的两个问题，要么不出现，一旦出现就是致命性的问题。所以面试官一定会问你。

3、面试题剖析

缓存雪崩发生的现象

缓存雪崩的事前事中事后的解决方案

事前：redis高可用，主从+哨兵，redis cluster，避免全盘崩溃

事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL被打死

事后：redis持久化，快速恢复缓存数据

缓存穿透的现象

缓存穿透的解决方法

Day27

1、面试题

如何保证缓存与数据库的双写一致性？

1. 面试官心里分析

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？

3、面试题剖析

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

最经典的缓存+数据库读写的模式，cache aside pattern

1、Cache Aside Pattern

（1）读的时候，先读缓存，缓存没有的话，那么就读数据库，然后取出数据后放入缓存，同时返回响应

（2）更新的时候，先删除缓存，然后再更新数据库

2、为什么是删除缓存，而不是更新缓存呢？

原因很简单，很多时候，复杂点的缓存的场景，因为缓存有的时候，不简单是数据库中直接取出来的值

商品详情页的系统，修改库存，只是修改了某个表的某些字段，但是要真正把这个影响的最终的库存计算出来，可能还需要从其他表查询一些数据，然后进行一些复杂的运算，才能最终计算出

现在最新的库存是多少，然后才能将库存更新到缓存中去

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据，并进行运算，才能计算出缓存最新的值的

更新缓存的代价是很高的

是不是说，每次修改数据库的时候，都一定要将其对应的缓存去跟新一份？也许有的场景是这样的，但是对于比较复杂的缓存数据计算的场景，就不是这样了

如果你频繁修改一个缓存涉及的多个表，那么这个缓存会被频繁的更新，频繁的更新缓存

但是问题在于，这个缓存到底会不会被频繁访问到？？？

举个例子，一个缓存涉及的表的字段，在1分钟内就修改了20次，或者是100次，那么缓存跟新20次，100次; 但是这个缓存在1分钟内就被读取了1次，有大量的冷数据

28法则，黄金法则，20%的数据，占用了80%的访问量

实际上，如果你只是删除缓存的话，那么1分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低

每次数据过来，就只是删除缓存，然后修改数据库，如果这个缓存，在1分钟内只是被访问了1次，那么只有那1次，缓存是要被重新计算的，用缓存才去算缓存

其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算

mybatis，hibernate，懒加载，思想

查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊

80%的情况，查这个部门，就只是要访问这个部门的信息就可以了

先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工

马上开始去开发业务系统

从哪一步开始做，从比较简单的那一块开始做，实时性要求比较高的那块数据的缓存去做

实时性比较高的数据缓存，选择的就是库存的服务

库存可能会修改，每次修改都要去更新这个缓存数据; 每次库存的数据，在缓存中一旦过期，或者是被清理掉了，前端的nginx服务都会发送请求给库存服务，去获取相应的数据

库存这一块，写数据库的时候，直接更新redis缓存

实际上没有这么的简单，这里，其实就涉及到了一个问题，数据库与缓存双写，数据不一致的问题

围绕和结合实时性较高的库存服务，把数据库与缓存双写不一致问题以及其解决方案，给大家讲解一下

数据库与缓存双写不一致，很常见的问题，大型的缓存架构中，第一个解决方案

大型的缓存架构全部讲解完了以后，整套架构是非常复杂，架构可以应对各种各样奇葩和极端的情况

也有一种可能，不是说，来讲课的就是超人，万能的

讲课，就跟写书一样，很可能会写错，也可能有些方案里的一些地方，我没考虑到

也可能说，有些方案只是适合某些场景，在某些场景下，可能需要你进行方案的优化和调整才能适用于你自己的项目

大家觉得对这些方案有什么疑问或者见解，都可以找我，沟通一下

如果的确我觉得是我讲解的不对，或者有些地方考虑不周，那么我可以在视频里补录，更新到网站上面去

多多包涵

1、最初级的缓存不一致问题以及解决方案

问题：先修改数据库，再删除缓存，如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据出现不一致

解决思路

先删除缓存，再修改数据库，如果删除缓存成功了，如果修改数据库失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致

因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中

2、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

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。。。。

3、为什么上亿流量高并发场景下，缓存会出现这个问题？

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题

其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就1万次，那么很少的情况下，会出现刚才描述的那种不一致的场景

但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况

高并发了以后，问题是很多的

4、数据库与缓存更新与读取操作进行异步串行化

更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中

读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中

一个队列对应一个工作线程

每个工作线程串行拿到对应的操作，然后一条一条的执行

这样的话，一个数据变更的操作，先执行，删除缓存，然后再去更新数据库，但是还没完成更新

此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成

这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值

5、高并发的场景下，该解决方案要注意的问题

（1）读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回

该解决方案，最大的风险点在于说，可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库

务必通过一些模拟真实的测试，看看更新数据的频繁是怎样的

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作

如果一个内存队列里居然会挤压100个商品的库存修改操作，每隔库存修改操作要耗费10ms区完成，那么最后一个商品的读请求，可能等待10 \* 100 = 1000ms = 1s后，才能得到数据

这个时候就导致读请求的长时阻塞

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会hang多少时间，如果读请求在200ms返回，如果你计算过后，哪怕是最繁忙的时候，积压10个更新操作，最多等待200ms，那还可以的

如果一个内存队列可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少

其实根据之前的项目经验，一般来说数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的

针对读高并发，读缓存架构的项目，一般写请求相对读来说，是非常非常少的，每秒的QPS能到几百就不错了

一秒，500的写操作，5份，每200ms，就100个写操作

单机器，20个内存队列，每个内存队列，可能就积压5个写操作，每个写操作性能测试后，一般在20ms左右就完成

那么针对每个内存队列中的数据的读请求，也就最多hang一会儿，200ms以内肯定能返回了

写QPS扩大10倍，但是经过刚才的测算，就知道，单机支撑写QPS几百没问题，那么就扩容机器，扩容10倍的机器，10台机器，每个机器20个队列，200个队列

大部分的情况下，应该是这样的，大量的读请求过来，都是直接走缓存取到数据的

少量情况下，可能遇到读跟数据更新冲突的情况，如上所述，那么此时更新操作如果先入队列，之后可能会瞬间来了对这个数据大量的读请求，但是因为做了去重的优化，所以也就一个更新缓存的操作跟在它后面

等数据更新完了，读请求触发的缓存更新操作也完成，然后临时等待的读请求全部可以读到缓存中的数据

（2）读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时hang在服务上，看服务能不能抗的住，需要多少机器才能抗住最大的极限情况的峰值

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大

按1:99的比例计算读和写的请求，每秒5万的读QPS，可能只有500次更新操作

如果一秒有500的写QPS，那么要测算好，可能写操作影响的数据有500条，这500条数据在缓存中失效后，可能导致多少读请求，发送读请求到库存服务来，要求更新缓存

一般来说，1:1，1:2，1:3，每秒钟有1000个读请求，会hang在库存服务上，每个读请求最多hang多少时间，200ms就会返回

在同一时间最多hang住的可能也就是单机200个读请求，同时hang住

单机hang200个读请求，还是ok的

1:20，每秒更新500条数据，这500秒数据对应的读请求，会有20 \* 500 = 1万

1万个读请求全部hang在库存服务上，就死定了

（3）多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过nginx服务器路由到相同的服务实例上

（4）热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能造成某台机器的压力过大

就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以更新频率不是太高的话，这个问题的影响并不是特别大

但是的确可能某些机器的负载会高一些

Day28

1、面试题

redis的并发竞争问题是什么？如何解决这个问题？了解Redis事务的CAS方案吗？

2、面试官心里分析

这个也是线上非常常见的一个问题，就是多客户端同时并发写一个key，可能本来应该先到的数据后到了，导致数据版本错了。或者是多客户端同时获取一个key，修改值之后再写回去，只要顺序错了，数据就错了。

而且redis自己就有天然解决这个问题的CAS类的乐观锁方案

3、面试题剖析

Day29

1、面试题

生产环境中的redis是怎么部署的？

2、面试官心里分析

看看你了解不了解你们公司的redis生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的redis是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上redis给几个G的内存？设置了哪些参数？压测后你们redis集群承载多少QPS？

兄弟，这些你必须是门儿清的，否则你确实是没好好思考过

3、面试题剖析

redis cluster，10台机器，5台机器部署了redis主实例，另外5台机器部署了redis的从实例，每个主实例挂了一个从实例，5个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒5万，5台机器最多是25万读写请求/s。

机器是什么配置？32G内存+8核CPU+1T磁盘，但是分配给redis进程的是10g内存，一般线上生产环境，redis的内存尽量不要超过10g，超过10g可能会有问题。

5台机器对外提供读写，一共有50g内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis从实例会自动变成主实例继续提供读写服务

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是10kb。100条数据是1mb，10万条数据是1g。常驻内存的是200万条商品数据，占用内存是20g，仅仅不到总内存的50%。

目前高峰期每秒就是3500左右的请求量

比如我们吧，大型的公司，其实基础架构的team，会负责缓存集群的运维

Day30

说实话，这一套东西基本构成了缓存这块你必须知道的基础性的知识，如果你不知道，那么说明你有点失职，确实平时没好好积累。

因为这些问题确实不难，如果我往深了问，可以问的很细，结合项目扣的很细，比如你们公司线上系统高峰QPS 3000？那请求主要访问哪些接口？redis抗了多少请求？mysql抗了多少请求？你到底是怎么实现高并发的？咱们聊聊redis的内核吧，看看你对底层了解的多么？如果要缓存几百GB的数据会有什么坑该这么弄？如果缓存出现热点现象该这么处理？某个value特别大把网卡给打死了怎么办？等等等等，可以深挖的东西其实有很多。。。。。

但是如果你掌握好了这套东西的回答，那么你在面试的时候，如果面试官没有全都问到，你可以自己主动合盘脱出。比如你可以说，我们线上的缓存，做了啥啥机制，防止雪崩、防止穿透、保证双写时的数据一致性、保证并发竞争时的数据一致性，我们线上咋部署的，啥架构，怎么玩儿的。这套东西你可以自己说出来，展示一下你对缓存这块的掌握。

Day31

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？

Day32

1、面试题

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？

2、面试官心里分析

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。

3、友情提示

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。

4、面试题剖析

（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。

Day33

1、面试题

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？

2、面试官心里分析

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。

3、面试题剖析

（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

Day34

1、面试题

dubbo支持哪些通信协议？支持哪些序列化协议？

2、面试官心里分析

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？

3、面试题剖析

（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。

Day35

1、面试题

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？

2、面试官心里分析

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？

3、面试题剖析

（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

Day 36

1、面试题

dubbo的spi思想是什么？

2、面试官心里分析

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

3、面试题剖析

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。

Day37

1、面试题

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？

2、面试官心里分析

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

3、面试题剖析

（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

Day38

1、面试题

分布式服务接口的幂等性如何设计（比如不能重复扣款）？

2、面试官心里分析

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

3、面试题剖析

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。

Day39

1、面试题

分布式服务接口请求的顺序性如何保证？

2、面试官心里分析

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

3、面试题剖析

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。

Day40

1、面试题

如何自己设计一个类似dubbo的rpc框架？

2、面试官心里分析

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

3、面试题剖析

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。

Day41

1、面试题

zk都有哪些使用场景？

2、面试官心里分析

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。

3、面试题剖析

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

Day42

1、面试题

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？

2、面试官心里分析

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。

3、面试题剖析

（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

（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();

}

}

（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

}

Day43

1、面试题

集群部署时的分布式session如何实现？

2、面试官心里分析

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。

3、面试题剖析

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sping 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。

Day44

1、面试题

分布式事务了解吗？你们如何解决分布式事务问题的？

2、面试官心里分析

只要聊到你做了分布式系统，必问分布式事务，你对分布式事务一无所知的话，确实会很坑，你起码得知道有哪些方案，一般怎么来做，每个方案的优缺点是什么。

现在面试，分布式系统成了标配，而分布式系统带来的分布式事务也成了标配了。因为你做系统肯定要用事务吧，那你用事务的话，分布式系统之后肯定要用分布式事务吧。。。呵呵。。。先不说你搞过没有，起码你得明白有哪几种方案，每种方案可能有啥坑？比如TCC方案的网络问题、XA方案的一致性问题

3、面试题剖析

（1）两阶段提交方案/XA方案

也叫做两阶段提交事务方案，这个举个例子，比如说咱们公司里经常tb是吧（就是团建），然后一般会有个tb主席（就是负责组织团建的那个人）。

tb，team building，团建

第一个阶段，一般tb主席会提前一周问一下团队里的每个人，说，大家伙，下周六我们去滑雪+烧烤，去吗？这个时候tb主席开始等待每个人的回答，如果所有人都说ok，那么就可以决定一起去这次tb。如果这个阶段里，任何一个人回答说，我有事不去了，那么tb主席就会取消这次活动。

第二个阶段，那下周六大家就一起去滑雪+烧烤了

所以这个就是所谓的XA事务，两阶段提交，有一个事务管理器的概念，负责协调多个数据库（资源管理器）的事务，事务管理器先问问各个数据库你准备好了吗？如果每个数据库都回复ok，那么就正式提交事务，在各个数据库上执行操作；如果任何一个数据库回答不ok，那么就回滚事务。

这种分布式事务方案，比较适合单块应用里，跨多个库的分布式事务，而且因为严重依赖于数据库层面来搞定复杂的事务，效率很低，绝对不适合高并发的场景。如果要玩儿，那么基于spring + JTA就可以搞定，自己随便搜个demo看看就知道了。

这个方案，我们很少用，一般来说某个系统内部如果出现跨多个库的这么一个操作，是不合规的。我可以给大家介绍一下， 现在微服务，一个大的系统分成几百个服务，几十个服务。一般来说，我们的规定和规范，是要求说每个服务只能操作自己对应的一个数据库。

如果你要操作别的服务对应的库，不允许直连别的服务的库，违反微服务架构的规范，你随便交叉胡乱访问，几百个服务的话，全体乱套，这样的一套服务是没法管理的，没法治理的，经常数据被别人改错，自己的库被别人写挂。

如果你要操作别人的服务的库，你必须是通过调用别的服务的接口来实现，绝对不允许你交叉访问别人的数据库！

（2）TCC方案

TCC的全程是：Try、Confirm、Cancel。

这个其实是用到了补偿的概念，分为了三个阶段：

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

给大家举个例子吧，比如说跨银行转账的时候，要涉及到两个银行的分布式事务，如果用TCC方案来实现，思路是这样的：

1）Try阶段：先把两个银行账户中的资金给它冻结住就不让操作了

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

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

这种方案说实话几乎很少用人使用，我们用的也比较少，但是也有使用的场景。因为这个事务回滚实际上是严重依赖于你自己写代码来回滚和补偿了，会造成补偿代码巨大，非常之恶心。

比如说我们，一般来说跟钱相关的，跟钱打交道的，支付、交易相关的场景，我们会用TCC，严格严格保证分布式事务要么全部成功，要么全部自动回滚，严格保证资金的正确性，在资金上出现问题

比较适合的场景：这个就是除非你是真的一致性要求太高，是你系统中核心之核心的场景，比如常见的就是资金类的场景，那你可以用TCC方案了，自己编写大量的业务逻辑，自己判断一个事务中的各个环节是否ok，不ok就执行补偿/回滚代码。

而且最好是你的各个业务执行的时间都比较短。

但是说实话，一般尽量别这么搞，自己手写回滚逻辑，或者是补偿逻辑，实在太恶心了，那个业务代码很难维护。

（3）本地消息表

国外的ebay搞出来的这么一套思想

这个大概意思是这样的

1）A系统在自己本地一个事务里操作同时，插入一条数据到消息表

2）接着A系统将这个消息发送到MQ中去

3）B系统接收到消息之后，在一个事务里，往自己本地消息表里插入一条数据，同时执行其他的业务操作，如果这个消息已经被处理过了，那么此时这个事务会回滚，这样保证不会重复处理消息

4）B系统执行成功之后，就会更新自己本地消息表的状态以及A系统消息表的状态

5）如果B系统处理失败了，那么就不会更新消息表状态，那么此时A系统会定时扫描自己的消息表，如果有没处理的消息，会再次发送到MQ中去，让B再次处理

6）这个方案保证了最终一致性，哪怕B事务失败了，但是A会不断重发消息，直到B那边成功为止

这个方案说实话最大的问题就在于严重依赖于数据库的消息表来管理事务啥的？？？这个会导致如果是高并发场景咋办呢？咋扩展呢？所以一般确实很少用

（4）可靠消息最终一致性方案

这个的意思，就是干脆不要用本地的消息表了，直接基于MQ来实现事务。比如阿里的RocketMQ就支持消息事务。

大概的意思就是：

1）A系统先发送一个prepared消息到mq，如果这个prepared消息发送失败那么就直接取消操作别执行了

2）如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉mq发送确认消息，如果失败就告诉mq回滚消息

3）如果发送了确认消息，那么此时B系统会接收到确认消息，然后执行本地的事务

4）mq会自动定时轮询所有prepared消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认消息？那是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，别确认消息发送失败了。

5）这个方案里，要是系统B的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如B系统本地回滚后，想办法通知系统A也回滚；或者是发送报警由人工来手工回滚和补偿

这个还是比较合适的，目前国内互联网公司大都是这么玩儿的，要不你举用RocketMQ支持的，要不你就自己基于类似ActiveMQ？RabbitMQ？自己封装一套类似的逻辑出来，总之思路就是这样子的

（5）最大努力通知方案

这个方案的大致意思就是：

1）系统A本地事务执行完之后，发送个消息到MQ

2）这里会有个专门消费MQ的最大努力通知服务，这个服务会消费MQ然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统B的接口

3）要是系统B执行成功就ok了；要是系统B执行失败了，那么最大努力通知服务就定时尝试重新调用系统B，反复N次，最后还是不行就放弃

（6）你们公司是如何处理分布式事务的？

这个，说真的，确实我们这个课程没法带着大家来实战，因为定位不是这个。但是如果你真的被问到，你可以这么说，我们某某特别严格的场景，用的是TCC来保证强一致性；然后其他的一些场景基于了阿里的RocketMQ来实现了分布式事务。

你找一个严格资金要求绝对不能错的场景，你可以说你是用的TCC方案；如果是一般的分布式事务场景，订单插入之后要调用库存服务更新库存，库存数据没有资金那么的敏感，可以用可靠消息最终一致性方案

友情提示一下，rocketmq 3.2.6之前的版本，是可以按照上面的思路来的，但是之后接口做了一些改变，我这里不再赘述了。

当然如果你愿意，你可以参考可靠消息最终一致性方案来自己实现一套分布式事务，比如基于rabbitmq来玩儿。

4、昨天学员给我提的一个问题

老师，我们现在想保证我们的某个系统非常的可靠，任何一个数据都不能错，我们用的是微服务架构，几十个服务。结果我们一盘点，发现，如果到处都要搞的话，一个系统要做几十个分布式事务出来。

我们的经验，我带几十人的team，最大的一个项目，起码几百个服务，复杂的分布式大型系统，里面其实也没几个分布式事务。

你其实用任何一个分布式事务的这么一个方案，都会导致你那块儿代码会复杂10倍。很多情况下，系统A调用系统B、系统C、系统D，我们可能根本就不做分布式事务。如果调用报错会打印异常日志。

每个月也就那么几个bug，很多bug是功能性的，体验性的，真的是涉及到数据层面的一些bug，一个月就几个，两三个？如果你为了确保系统自动保证数据100%不能错，上了几十个分布式事务，代码太复杂；性能太差，系统吞吐量、性能大幅度下跌。

99%的分布式接口调用，不要做分布式事务，直接就是监控（发邮件、发短信）、记录日志（一旦出错，完整的日志）、事后快速的定位、排查和出解决方案、修复数据。

每个月，每隔几个月，都会对少量的因为代码bug，导致出错的数据，进行人工的修复数据，自己临时动手写个程序，可能要补一些数据，可能要删除一些数据，可能要修改一些字段的值。

比你做50个分布式事务，成本要来的低上百倍，低几十倍

trade off，权衡，要用分布式事务的时候，一定是有成本，代码会很复杂，开发很长时间，性能和吞吐量下跌，系统更加复杂更加脆弱反而更加容易出bug；好处，如果做好了，TCC、可靠消息最终一致性方案，一定可以100%保证你那快数据不会出错。

1%，0.1%，0.01%的业务，资金、交易、订单，我们会用分布式事务方案来保证，会员积分、优惠券、商品信息，其实不要这么搞了

Day45

1、面试题

如何设计一个高并发系统？

2、面试官心里分析

说实话，如果面试官问你这个题目，那么你必须要使出全身吃奶劲了。为啥？因为你没看到现在很多公司招聘的jd里都是说啥，有高并发就经验者优先。

所以如果你确实有真才实学，在互联网公司里干过高并发系统，那你确实拿offer基本如探囊取物，没啥问题。但是如果你要是真是干过高并发系统，面试官绝对绝对不会问这个问题，否则他就是蠢。

假设你在某知名电商公司干过高并发系统，用户上亿，一天流量几十亿，高峰期并发量上万，甚至是十万。那么人家一定会仔细盘问你的系统架构，你们系统啥架构？怎么部署的？部署了多少台机器？缓存咋用的？MQ咋用的？数据库咋用的？就是深挖你到底是如何抗下高并发的。

因为真正干过高并发的人一定知道，脱离了业务的系统架构都是在纸上谈兵，真正在复杂业务场景而且还高并发的时候，那系统架构一定不是那么简单的，用个redis，用mq就能搞定？当然不是，真实的系统架构搭配上业务之后，会比这种简单的所谓“高并发架构”要复杂很多倍。

如果有面试官问你个问题说，如何设计一个高并发系统？那么不好意思，一定是因为你实际上没干过高并发系统。面试官看你简历就没啥出彩的，感觉就不咋地，所以就会问问你，如何设计一个高并发系统？其实说白了本质就是看看你有没有自己研究过，有没有一定的知识积累。

最好的当然是招聘个真正干过高并发的哥儿们咯，但是这种哥儿们人数稀缺，不好招。所以可能次一点的就是招一个自己研究过的哥儿们，总比招一个傻也不会的哥儿们好吧！

所以这个时候你必须得做一把个人秀了，秀出你所有关于高并发的知识！

3、面试题剖析

其实所谓的高并发，如果你要理解这个问题呢，其实就得从高并发的根源出发，为啥会有高并发？为啥高并发就很牛逼？

我说的浅显一点，很简单，就是因为刚开始系统都是连接数据库的，但是要知道数据库支撑到每秒并发两三千的时候，基本就快完了。所以才有说，很多公司，刚开始干的时候，技术比较low，结果业务发展太快，有的时候系统扛不住压力就挂了。

当然会挂了，凭什么不挂？你数据库如果瞬间承载每秒5000,8000，甚至上万的并发，一定会宕机，因为比如mysql就压根儿扛不住这么高的并发量。

所以为啥高并发牛逼？就是因为现在用互联网的人越来越多，很多app、网站、系统承载的都是高并发请求，可能高峰期每秒并发量几千，很正常的。如果是什么双十一了之类的，每秒并发几万几十万都有可能。

那么如此之高的并发量，加上原本就如此之复杂的业务，咋玩儿？真正厉害的，一定是在复杂业务系统里玩儿过高并发架构的人，但是你没有，那么我给你说一下你该怎么回答这个问题：

（1）系统拆分，将一个系统拆分为多个子系统，用dubbo来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以抗高并发么。

（2）缓存，必须得用缓存。大部分的高并发场景，都是读多写少，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟人家redis轻轻松松单机几万的并发啊。没问题的。所以你可以考虑考虑你的项目里，那些承载主要请求的读场景，怎么用缓存来抗高并发。

（3）MQ，必须得用MQ。可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改，疯了。那高并发绝对搞挂你的系统，你要是用redis来承载写那肯定不行，人家是缓存，数据随时就被LRU了，数据格式还无比简单，没有事务支持。所以该用mysql还得用mysql啊。那你咋办？用MQ吧，大量的写请求灌入MQ里，排队慢慢玩儿，后边系统消费后慢慢写，控制在mysql承载范围之内。所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用MQ来异步写，提升并发性。MQ单机抗几万并发也是ok的，这个之前还特意说过。

（4）分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来抗更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高sql跑的性能。

（5）读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，主库写入，从库读取，搞一个读写分离。读流量太多的时候，还可以加更多的从库。

（6）Elasticsearch，可以考虑用es。es是分布式的，可以随便扩容，分布式天然就可以支撑高并发，因为动不动就可以扩容加机器来抗更高的并发。那么一些比较简单的查询、统计类的操作，可以考虑用es来承载，还有一些全文搜索类的操作，也可以考虑用es来承载。

上面的6点，基本就是高并发系统肯定要干的一些事儿，大家可以仔细结合之前讲过的知识考虑一下，到时候你可以系统的把这块阐述一下，然后每个部分要注意哪些问题，之前都讲过了，你都可以阐述阐述，表明你对这块是有点积累的。

说句实话，毕竟真正你厉害的一点，不是在于弄明白一些技术，或者大概知道一个高并发系统应该长什么样？其实实际上在真正的复杂的业务系统里，做高并发要远远比我这个图复杂几十倍到上百倍。你需要考虑，哪些需要分库分表，哪些不需要分库分表，单库单表跟分库分表如何join，哪些数据要放到缓存里去啊，放哪些数据再可以抗掉高并发的请求，你需要完成对一个复杂业务系统的分析之后，然后逐步逐步的加入高并发的系统架构的改造，这个过程是务必复杂的，一旦做过一次，一旦做好了，你在这个市场上就会非常的吃香。

其实大部分公司，真正看重的，不是说你掌握高并发相关的一些基本的架构知识，架构中的一些技术，RocketMQ、Kafka、Redis、Elasticsearch，高并发这一块，次一等的人才。对一个有几十万行代码的复杂的分布式系统，一步一步架构、设计以及实践过高并发架构的人，这个经验是难能可贵的。

我这边其实平时我会发布一些免费的课程，每隔一段时间定期发布一点，主要是尽可能给大家讲一些免费的课程，保证质量， 让大家学到一些东西。

我主要还是专注在自己的架构师体系的课程上面，是一年多的时间，非常长，内容极其庞大，我从一开始就带着你从0开始，动手构建一个10万行以上代码量的这么个庞大的系统，针对这种复杂系统的业务场景，里面隐含的各种技术问题和坑，我会通过1年多的时间，一步一步的讲解各种技术和架构，解决真实的大型的系统中的各种问题。

Day46

（1）为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

（2）现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

（3）如何设计可以动态扩容缩容的分库分表方案？

（4）分库分表之后，id主键如何处理？

Day47

1、面试题

为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

2、面试官心里分析

其实这块肯定是扯到高并发了，因为分库分表一定是为了支撑高并发、数据量大两个问题的。而且现在说实话，尤其是互联网类的公司面试，基本上都会来这么一下，分库分表如此普遍的技术问题，不问实在是不行，而如果你不知道那也实在是说不过去！

3、面试题剖析

（1）为什么要分库分表？（设计高并发系统的时候，数据库层面该如何设计？）

说白了，分库分表是两回事儿，大家可别搞混了，可能是光分库不分表，也可能是光分表不分库，都有可能。我先给大家抛出来一个场景。

假如我们现在是一个小创业公司（或者是一个BAT公司刚兴起的一个新部门），现在注册用户就20万，每天活跃用户就1万，每天单表数据量就1000，然后高峰期每秒钟并发请求最多就10。。。天，就这种系统，随便找一个有几年工作经验的，然后带几个刚培训出来的，随便干干都可以。

结果没想到我们运气居然这么好，碰上个CEO带着我们走上了康庄大道，业务发展迅猛，过了几个月，注册用户数达到了2000万！每天活跃用户数100万！每天单表数据量10万条！高峰期每秒最大请求达到1000！同时公司还顺带着融资了两轮，紧张了几个亿人民币啊！公司估值达到了惊人的几亿美金！这是小独角兽的节奏！

好吧，没事，现在大家感觉压力已经有点大了，为啥呢？因为每天多10万条数据，一个月就多300万条数据，现在咱们单表已经几百万数据了，马上就破千万了。但是勉强还能撑着。高峰期请求现在是1000，咱们线上部署了几台机器，负载均衡搞了一下，数据库撑1000 QPS也还凑合。但是大家现在开始感觉有点担心了，接下来咋整呢。。。。。。

再接下来几个月，我的天，CEO太牛逼了，公司用户数已经达到1亿，公司继续融资几十亿人民币啊！公司估值达到了惊人的几十亿美金，成为了国内今年最牛逼的明星创业公司！天，我们太幸运了。

但是我们同时也是不幸的，因为此时每天活跃用户数上千万，每天单表新增数据多达50万，目前一个表总数据量都已经达到了两三千万了！扛不住啊！数据库磁盘容量不断消耗掉！高峰期并发达到惊人的5000~8000！别开玩笑了，哥。我跟你保证，你的系统支撑不到现在，已经挂掉了！

好吧，所以看到你这里你差不多就理解分库分表是怎么回事儿了，实际上这是跟着你的公司业务发展走的，你公司业务发展越好，用户就越多，数据量越大，请求量越大，那你单个数据库一定扛不住。

比如你单表都几千万数据了，你确定你能抗住么？绝对不行，单表数据量太大，会极大影响你的sql执行的性能，到了后面你的sql可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。

分表是啥意思？就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户id来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。这样可以控制每个表的数据量在可控的范围内，比如每个表就固定在200万以内。

分库是啥意思？就是你一个库一般我们经验而言，最多支撑到并发2000，一定要扩容了，而且一个健康的单库并发值你最好保持在每秒1000左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。

这就是所谓的分库分表，为啥要分库分表？你明白了吧

（2）用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？

这个其实就是看看你了解哪些分库分表的中间件，各个中间件的优缺点是啥？然后你用过哪些分库分表的中间件。

比较常见的包括：cobar、TDDL、atlas、sharding-jdbc、mycat

cobar：阿里b2b团队开发和开源的，属于proxy层方案。早些年还可以用，但是最近几年都没更新了，基本没啥人用，差不多算是被抛弃的状态吧。而且不支持读写分离、存储过程、跨库join和分页等操作。

TDDL：淘宝团队开发的，属于client层方案。不支持join、多表查询等语法，就是基本的crud语法是ok，但是支持读写分离。目前使用的也不多，因为还依赖淘宝的diamond配置管理系统。

atlas：360开源的，属于proxy层方案，以前是有一些公司在用的，但是确实有一个很大的问题就是社区最新的维护都在5年前了。所以，现在用的公司基本也很少了。

sharding-jdbc：当当开源的，属于client层方案。确实之前用的还比较多一些，因为SQL语法支持也比较多，没有太多限制，而且目前推出到了2.0版本，支持分库分表、读写分离、分布式id生成、柔性事务（最大努力送达型事务、TCC事务）。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从2017年一直到现在，是不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，个人认为算是一个现在也可以选择的方案。

mycat：基于cobar改造的，属于proxy层方案，支持的功能非常完善，而且目前应该是非常火的而且不断流行的数据库中间件，社区很活跃，也有一些公司开始在用了。但是确实相比于sharding jdbc来说，年轻一些，经历的锤炼少一些。

所以综上所述，现在其实建议考量的，就是sharding-jdbc和mycat，这两个都可以去考虑使用。

sharding-jdbc这种client层方案的优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高，但是如果遇到升级啥的需要各个系统都重新升级版本再发布，各个系统都需要耦合sharding-jdbc的依赖；

mycat这种proxy层方案的缺点在于需要部署，自己及运维一套中间件，运维成本高，但是好处在于对于各个项目是透明的，如果遇到升级之类的都是自己中间件那里搞就行了。

通常来说，这两个方案其实都可以选用，但是我个人建议中小型公司选用sharding-jdbc，client层方案轻便，而且维护成本低，不需要额外增派人手，而且中小型公司系统复杂度会低一些，项目也没那么多；

但是中大型公司最好还是选用mycat这类proxy层方案，因为可能大公司系统和项目非常多，团队很大，人员充足，那么最好是专门弄个人来研究和维护mycat，然后大量项目直接透明使用即可。

我们，数据库中间件都是自研的，也用过proxy层，后来也用过client层

（3）你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

水平拆分的意思，就是把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的，所有库表的数据加起来就是全部数据。水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来抗更高的并发，还有就是用多个库的存储容量来进行扩容。

垂直拆分的意思，就是把一个有很多字段的表给拆分成多个表，或者是多个库上去。每个库表的结构都不一样，每个库表都包含部分字段。一般来说，会将较少的访问频率很高的字段放到一个表里去，然后将较多的访问频率很低的字段放到另外一个表里去。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。

这个其实挺常见的，不一定我说，大家很多同学可能自己都做过，把一个大表拆开，订单表、订单支付表、订单商品表。

还有表层面的拆分，就是分表，将一个表变成N个表，就是让每个表的数据量控制在一定范围内，保证SQL的性能。否则单表数据量越大，SQL性能就越差。一般是200万行左右，不要太多，但是也得看具体你怎么操作，也可能是500万，或者是100万。你的SQL越复杂，就最好让单表行数越少。

好了，无论是分库了还是分表了，上面说的那些数据库中间件都是可以支持的。就是基本上那些中间件可以做到你分库分表之后，中间件可以根据你指定的某个字段值，比如说userid，自动路由到对应的库上去，然后再自动路由到对应的表里去。

你就得考虑一下，你的项目里该如何分库分表？一般来说，垂直拆分，你可以在表层面来做，对一些字段特别多的表做一下拆分；水平拆分，你可以说是并发承载不了，或者是数据量太大，容量承载不了，你给拆了，按什么字段来拆，你自己想好；分表，你考虑一下，你如果哪怕是拆到每个库里去，并发和容量都ok了，但是每个库的表还是太大了，那么你就分表，将这个表分开，保证每个表的数据量并不是很大。

而且这儿还有两种分库分表的方式，一种是按照range来分，就是每个库一段连续的数据，这个一般是按比如时间范围来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了；或者是按照某个字段hash一下均匀分散，这个较为常用。

range来分，好处在于说，后面扩容的时候，就很容易，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用range，要看场景，你的用户不是仅仅访问最新的数据，而是均匀的访问现在的数据以及历史的数据

hash分法，好处在于说，可以平均分配没给库的数据量和请求压力；坏处在于说扩容起来比较麻烦，会有一个数据迁移的这么一个过程

Day48

1、面试题

现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

2、面试官心里分析

你看看，你现在已经明白为啥要分库分表了，你也知道常用的分库分表中间件了，你也设计好你们如何分库分表的方案了（水平拆分、垂直拆分、分表），那问题来了，你接下来该怎么把你那个单库单表的系统给迁移到分库分表上去？

所以这都是一环扣一环的，就是看你有没有全流程经历过这个过程

友情提示

假设，你现有有一个单库单表的系统，在线上在跑，假设单表有600万数据

3个库，每个库里分了4个表，每个表要放50万的数据量

假设你已经选择了一个分库分表的数据库中间件，sharding-jdbc，mycat，都可以

你怎么把线上系统平滑地迁移到分库分表上面去

sharding-jdbc：自己上官网，找一个官网最基本的例子，自己写一下，试一下，跑跑看，是非常简单的

mycat：自己上官网，找一个官网最基本的例子，自己写一下，试一下看看

1个小时以内就可以搞定了

3、面试题剖析

这个其实从low到高大上有好几种方案，我们都玩儿过，我都给你说一下

（1）停机迁移方案

我先给你说一个最low的方案，就是很简单，大家伙儿凌晨12点开始运维，网站或者app挂个公告，说0点到早上6点进行运维，无法访问。。。。。。

接着到0点，停机，系统挺掉，没有流量写入了，此时老的单库单表数据库静止了。然后你之前得写好一个导数的一次性工具，此时直接跑起来，然后将单库单表的数据哗哗哗读出来，写到分库分表里面去。

导数完了之后，就ok了，修改系统的数据库连接配置啥的，包括可能代码和SQL也许有修改，那你就用最新的代码，然后直接启动连到新的分库分表上去。

验证一下，ok了，完美，大家伸个懒腰，看看看凌晨4点钟的北京夜景，打个滴滴回家吧

但是这个方案比较low，谁都能干，我们来看看高大上一点的方案

（2）双写迁移方案

这个是我们常用的一种迁移方案，比较靠谱一些，不用停机，不用看北京凌晨4点的风景

简单来说，就是在线上系统里面，之前所有写库的地方，增删改操作，都除了对老库增删改，都加上对新库的增删改，这就是所谓双写，同时写俩库，老库和新库。

然后系统部署之后，新库数据差太远，用之前说的导数工具，跑起来读老库数据写新库，写的时候要根据gmt\_modified这类字段判断这条数据最后修改的时间，除非是读出来的数据在新库里没有，或者是比新库的数据新才会写。

接着导万一轮之后，有可能数据还是存在不一致，那么就程序自动做一轮校验，比对新老库每个表的每条数据，接着如果有不一样的，就针对那些不一样的，从老库读数据再次写。反复循环，直到两个库每个表的数据都完全一致为止。

接着当数据完全一致了，就ok了，基于仅仅使用分库分表的最新代码，重新部署一次，不就仅仅基于分库分表在操作了么，还没有几个小时的停机时间，很稳。所以现在基本玩儿数据迁移之类的，都是这么干了。

Day49

1、面试题

如何设计可以动态扩容缩容的分库分表方案？

2、面试官心里分析

（1）选择一个数据库中间件，调研、学习、测试

（2）设计你的分库分表的一个方案，你要分成多少个库，每个库分成多少个表，3个库每个库4个表

（3）基于选择好的数据库中间件，以及在测试环境建立好的分库分表的环境，然后测试一下能否正常进行分库分表的读写

（4）完成单库单表到分库分表的迁移，双写方案

（5）线上系统开始基于分库分表对外提供服务

（6）扩容了，扩容成6个库，每个库需要12个表，你怎么来增加更多库和表呢？

这个是你必须面对的一个事儿，就是你已经弄好分库分表方案了，然后一堆库和表都建好了，基于分库分表中间件的代码开发啥的都好了，测试都ok了，数据能均匀分布到各个库和各个表里去，而且接着你还通过双写的方案咔嚓一下上了系统，已经直接基于分库分表方案在搞了。

那么现在问题来了，你现在这些库和表又支撑不住了，要继续扩容咋办？这个可能就是说你的每个库的容量又快满了，或者是你的表数据量又太大了，也可能是你每个库的写并发太高了，你得继续扩容。

这都是玩儿分库分表线上必须经历的事儿

3、面试题剖析

（1）停机扩容

这个方案就跟停机迁移一样，步骤几乎一致，唯一的一点就是那个导数的工具，是把现有库表的数据抽出来慢慢倒入到新的库和表里去。但是最好别这么玩儿，有点不太靠谱，因为既然分库分表就说明数据量实在是太大了，可能多达几亿条，甚至几十亿，你这么玩儿，可能会出问题。

从单库单表迁移到分库分表的时候，数据量并不是很大，单表最大也就两三千万

写个工具，多弄几台机器并行跑，1小时数据就导完了

3个库+12个表，跑了一段时间了，数据量都1亿~2亿了。光是导2亿数据，都要导个几个小时，6点，刚刚导完数据，还要搞后续的修改配置，重启系统，测试验证，10点才可以搞完

（2）优化后的方案

一开始上来就是32个库，每个库32个表，1024张表

我可以告诉各位同学说，这个分法，第一，基本上国内的互联网肯定都是够用了，第二，无论是并发支撑还是数据量支撑都没问题

每个库正常承载的写入并发量是1000，那么32个库就可以承载32 \* 1000 = 32000的写并发，如果每个库承载1500的写并发，32 \* 1500 = 48000的写并发，接近5万/s的写入并发，前面再加一个MQ，削峰，每秒写入MQ 8万条数据，每秒消费5万条数据。

有些除非是国内排名非常靠前的这些公司，他们的最核心的系统的数据库，可能会出现几百台数据库的这么一个规模，128个库，256个库，512个库

1024张表，假设每个表放500万数据，在MySQL里可以放50亿条数据

每秒的5万写并发，总共50亿条数据，对于国内大部分的互联网公司来说，其实一般来说都够了

谈分库分表的扩容，第一次分库分表，就一次性给他分个够，32个库，1024张表，可能对大部分的中小型互联网公司来说，已经可以支撑好几年了

一个实践是利用32 \* 32来分库分表，即分为32个库，每个库里一个表分为32张表。一共就是1024张表。根据某个id先根据32取模路由到库，再根据32取模路由到库里的表。

刚开始的时候，这个库可能就是逻辑库，建在一个数据库上的，就是一个mysql服务器可能建了n个库，比如16个库。后面如果要拆分，就是不断在库和mysql服务器之间做迁移就可以了。然后系统配合改一下配置即可。

比如说最多可以扩展到32个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到1024个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是1024个表么。

这么搞，是不用自己写代码做数据迁移的，都交给dba来搞好了，但是dba确实是需要做一些库表迁移的工作，但是总比你自己写代码，抽数据导数据来的效率高得多了。

哪怕是要减少库的数量，也很简单，其实说白了就是按倍数缩容就可以了，然后修改一下路由规则。

对2 ^ n取模

orderId 模 32 = 库

orderId / 32 模 32 = 表

259 3 8

1189 5 5

352 0 11

4593 17 15

1、设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是32库 \* 32表，对于大部分公司来说，可能几年都够了

2、路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表

3、扩容的时候，申请增加更多的数据库服务器，装好mysql，倍数扩容，4台服务器，扩到8台服务器，16台服务器

4、由dba负责将原先数据库服务器的库，迁移到新的数据库服务器上去，很多工具，库迁移，比较便捷

5、我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址

6、重新发布系统，上线，原先的路由规则变都不用变，直接可以基于2倍的数据库服务器的资源，继续进行线上系统的提供服务

Day50

1、面试题

分库分表之后，id主键如何处理？

2、面试官心里分析

其实这是分库分表之后你必然要面对的一个问题，就是id咋生成？因为要是分成多个表之后，每个表都是从1开始累加，那肯定不对啊，需要一个全局唯一的id来支持。所以这都是你实际生产环境中必须考虑的问题。

3、面试题剖析

（1）数据库自增id

这个就是说你的系统里每次得到一个id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个id。拿到这个id之后再往对应的分库分表里去写入。

这个方案的好处就是方便简单，谁都会用；缺点就是单库生成自增id，要是高并发的话，就会有瓶颈的；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前id最大值，然后自己递增几个id，一次性返回一批id，然后再把当前最大id值修改成递增几个id之后的一个值；但是无论怎么说都是基于单个数据库。

适合的场景：你分库分表就俩原因，要不就是单库并发太高，要不就是单库数据量太大；除非是你并发不高，但是数据量太大导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。

并发很低，几百/s，但是数据量大，几十亿的数据，所以需要靠分库分表来存放海量的数据

（2）uuid

好处就是本地生成，不要基于数据库来了；不好之处就是，uuid太长了，作为主键性能太差了，不适合用于主键。

适合的场景：如果你是要随机生成个什么文件名了，编号之类的，你可以用uuid，但是作为主键是不能用uuid的。

UUID.randomUUID().toString().replace(“-”, “”) -> sfsdf23423rr234sfdaf

（3）获取系统当前时间

这个就是获取当前时间即可，但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。基本就不用考虑了。

适合的场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号，订单编号，时间戳 + 用户id + 业务含义编码

（4）snowflake算法

twitter开源的分布式id生成算法，就是把一个64位的long型的id，1个bit是不用的，用其中的41 bit作为毫秒数，用10 bit作为工作机器id，12 bit作为序列号

1 bit：不用，为啥呢？因为二进制里第一个bit为如果是1，那么都是负数，但是我们生成的id都是正数，所以第一个bit统一都是0

41 bit：表示的是时间戳，单位是毫秒。41 bit可以表示的数字多达2^41 - 1，也就是可以标识2 ^ 41 - 1个毫秒值，换算成年就是表示69年的时间。

10 bit：记录工作机器id，代表的是这个服务最多可以部署在2^10台机器上哪，也就是1024台机器。但是10 bit里5个bit代表机房id，5个bit代表机器id。意思就是最多代表2 ^ 5个机房（32个机房），每个机房里可以代表2 ^ 5个机器（32台机器）。

12 bit：这个是用来记录同一个毫秒内产生的不同id，12 bit可以代表的最大正整数是2 ^ 12 - 1 = 4096，也就是说可以用这个12bit代表的数字来区分同一个毫秒内的4096个不同的id

64位的long型的id，64位的long -> 二进制

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

2018-01-01 10:00:00 -> 做了一些计算，再换算成一个二进制，41bit来放 -> 0001100 10100010 10111110 10001001 01011100 00

机房id，17 -> 换算成一个二进制 -> 10001

机器id，25 -> 换算成一个二进制 -> 11001

snowflake算法服务，会判断一下，当前这个请求是否是，机房17的机器25，在2175/11/7 12:12:14时间点发送过来的第一个请求，如果是第一个请求

假设，在2175/11/7 12:12:14时间里，机房17的机器25，发送了第二条消息，snowflake算法服务，会发现说机房17的机器25，在2175/11/7 12:12:14时间里，在这一毫秒，之前已经生成过一个id了，此时如果你同一个机房，同一个机器，在同一个毫秒内，再次要求生成一个id，此时我只能把加1

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000001

比如我们来观察上面的那个，就是一个典型的二进制的64位的id，换算成10进制就是910499571847892992。

public class IdWorker{

private long workerId;

private long datacenterId;

private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence){

// sanity check for workerId

// 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0

if (workerId > maxWorkerId || workerId < 0) {

throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0",maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0) {

throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0",maxDatacenterId));

}

System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worker id bits %d, sequence bits %d, workerid %d",

timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;

private long datacenterIdBits = 5L;

private long maxWorkerId = -1L ^ (-1L << workerIdBits); // 这个是二进制运算，就是5 bit最多只能有31个数字，也就是说机器id最多只能是32以内

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); // 这个是一个意思，就是5 bit最多只能有31个数字，机房id最多只能是32以内

private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId(){

return workerId;

}

public long getDatacenterId(){

return datacenterId;

}

public long getTimestamp(){

return System.currentTimeMillis();

}

public synchronized long nextId() {

// 这儿就是获取当前时间戳，单位是毫秒

long timestamp = timeGen();

if (timestamp < lastTimestamp) {

System.err.printf("clock is moving backwards. Rejecting requests until %d.", lastTimestamp);

throw new RuntimeException(String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",

lastTimestamp - timestamp));

}

// 0

// 在同一个毫秒内，又发送了一个请求生成一个id，0 -> 1

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask; // 这个意思是说一个毫秒内最多只能有4096个数字，无论你传递多少进来，这个位运算保证始终就是在4096这个范围内，避免你自己传递个sequence超过了4096这个范围

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {

sequence = 0;

}

// 这儿记录一下最近一次生成id的时间戳，单位是毫秒

lastTimestamp = timestamp;

// 这儿就是将时间戳左移，放到41 bit那儿；将机房id左移放到5 bit那儿；将机器id左移放到5 bit那儿；将序号放最后10 bit；最后拼接起来成一个64 bit的二进制数字，转换成10进制就是个long型

return ((timestamp - twepoch) << timestampLeftShift) |

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;

}

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

private long timeGen(){

return System.currentTimeMillis();

}

//---------------测试---------------

public static void main(String[] args) {

IdWorker worker = new IdWorker(1,1,1);

for (int i = 0; i < 30; i++) {

System.out.println(worker.nextId());

}

}

}

怎么说呢，大概这个意思吧，就是说41 bit，就是当前毫秒单位的一个时间戳，就这意思；然后5 bit是你传递进来的一个机房id（但是最大只能是32以内），5 bit是你传递进来的机器id（但是最大只能是32以内），剩下的那个10 bit序列号，就是如果跟你上次生成id的时间还在一个毫秒内，那么会把顺序给你累加，最多在4096个序号以内。

所以你自己利用这个工具类，自己搞一个服务，然后对每个机房的每个机器都初始化这么一个东西，刚开始这个机房的这个机器的序号就是0。然后每次接收到一个请求，说这个机房的这个机器要生成一个id，你就找到对应的Worker，生成。

他这个算法生成的时候，会把当前毫秒放到41 bit中，然后5 bit是机房id，5 bit是机器id，接着就是判断上一次生成id的时间如果跟这次不一样，序号就自动从0开始；要是上次的时间跟现在还是在一个毫秒内，他就把seq累加1，就是自动生成一个毫秒的不同的序号。

这个算法那，可以确保说每个机房每个机器每一毫秒，最多生成4096个不重复的id。

利用这个snowflake算法，你可以开发自己公司的服务，甚至对于机房id和机器id，反正给你预留了5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

这个snowflake算法相对来说还是比较靠谱的，所以你要真是搞分布式id生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。

Day51

1、面试题

你们有没有做MySQL读写分离？如何实现mysql的读写分离？MySQL主从复制原理的是啥？如何解决mysql主从同步的延时问题？

2、面试官心里分析

这个，高并发这个阶段，那肯定是需要做读写分离的，啥意思？因为实际上大部分的互联网公司，一些网站，或者是app，其实都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

3、面试题剖析

（1）如何实现mysql的读写分离？

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

（2）MySQL主从复制原理的是啥？

主库将变更写binlog日志，然后从库连接到主库之后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个中继日志中。接着从库中有一个SQL线程会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再次执行一遍SQL，这样就可以保证自己跟主库的数据是一样的。

这里有一个非常重要的一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行的操作，在从库上会串行执行。所以这就是一个非常重要的点了，由于从库从主库拷贝日志以及串行执行SQL的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。

而且这里还有另外一个问题，就是如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

所以mysql实际上在这一块有两个机制，一个是半同步复制，用来解决主库数据丢失问题；一个是并行复制，用来解决主从同步延时问题。

这个所谓半同步复制，semi-sync复制，指的就是主库写入binlog日志之后，就会将强制此时立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。

所谓并行复制，指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

1. 主从复制的原理
2. 主从延迟问题产生的原因
3. 主从复制的数据丢失问题，以及半同步复制的原理
4. 并行复制的原理，多库并发重放relay日志，缓解主从延迟问题

（3）mysql主从同步延时问题（精华）

线上确实处理过因为主从同步延时问题，导致的线上的bug，小型的生产事故

show status，Seconds\_Behind\_Master，你可以看到从库复制主库的数据落后了几ms

其实这块东西我们经常会碰到，就比如说用了mysql主从架构之后，可能会发现，刚写入库的数据结果没查到，结果就完蛋了。。。。

所以实际上你要考虑好应该在什么场景下来用这个mysql主从同步，建议是一般在读远远多于写，而且读的时候一般对数据时效性要求没那么高的时候，用mysql主从同步

所以这个时候，我们可以考虑的一个事情就是，你可以用mysql的并行复制，但是问题是那是库级别的并行，所以有时候作用不是很大

所以这个时候。。通常来说，我们会对于那种写了之后立马就要保证可以查到的场景，采用强制读主库的方式，这样就可以保证你肯定的可以读到数据了吧。其实用一些数据库中间件是没问题的。

一般来说，如果主从延迟较为严重

1. 分库，将一个主库拆分为4个主库，每个主库的写并发就500/s，此时主从延迟可以忽略不计
2. 打开mysql支持的并行复制，多个库并行复制，如果说某个库的写入并发就是特别高，单库写并发达到了2000/s，并行复制还是没意义。28法则，很多时候比如说，就是少数的几个订单表，写入了2000/s，其他几十个表10/s。
3. 重写代码，写代码的同学，要慎重，当时我们其实短期是让那个同学重写了一下代码，插入数据之后，直接就更新，不要查询
4. 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。不推荐这种方法，你这么搞导致读写分离的意义就丧失了

Day52

就是现在，一般来说，互联网的面试，一般都会考察你，什么是分布式系统，高并发，简单的高可用问题。限流、熔断、降级，在分布式的系统架构中，微服务架构中，其实都是最常见、基础和简单的保障系统高可用的手法。dubbo去开发了，spring cloud去开发了，在这个系统的接口调用中，我们是用hystrix去实现一整套的高可用保障机制，基于hystrix去做了限流、熔断和降级。

hystrix是国外的netflix开源的，netflix是国外很大的视频网站，系统非常复杂，微服务架构，多达几千个服务，为自己的场景，经过大量的工业验证，线上生产环境的实践，产出和开源了高可用相关的一个框架，熔断框架，hystrix。

如何用hystrix做限流、熔断和降级。以及这些都是什么鬼？如何使用hystrix来在你的系统中做开发，加入高可用的保障机制？我之前有一个课程都已经讲解过一整套了，hystrix从入门到精通，全程有动手代码实战开发的。

hystrix未来会成为国内的高可用的限流、熔断和降级这一块的事实上的标准，spring cloud微服务框架，就是集成了hystrix来做微服务架构中的限流、降级和熔断的。

## 3.1 如何设计一个高可用系统？

## 3.2 限流

1. 如何限流？在工作中是怎么做的？说一下具体的实现？

## 3.3 熔断

（1）如何进行熔断？熔断框架都有哪些？具体实现原理知道吗？

## 3.4 降级

（1）如何进行降级？

前半部分，专注在高并发这一块，缓存架构，承载高并发，在各种高并发导致的令人崩溃/异常的场景下，运行着

缓存架构，高可用性，在各种系统的各个地方有乱七八糟的异常和故障的情况下，整套缓存系统还能继续健康的run着

HA，HAProxy，主备服务间的切换，这就做到了高可用性，主备实例，多冗余实例，高可用最最基础的东西

什么样的情况下，可能会导致系统的崩溃，以及系统不可用，针对各种各样的一些情况，然后我们用什么技术，去保护整个系统处于高可用的一个情况下

1、hystrix是什么？

netflix（国外最大的类似于，爱奇艺，优酷）视频网站，五六年前，也是，感觉自己的系统，整个网站，经常出故障，可用性不太高

有时候一些vip会员不能支付，有时候看视频就卡顿，看不了视频。。。

影响公司的收入。。。

五六年前，netflix，api team，提升高可用性，开发了一个框架，类似于spring，mybatis，hibernate，等等这种框架

高可用性的框架，hystrix

hystrix，框架，提供了高可用相关的各种各样的功能，然后确保说在hystrix的保护下，整个系统可以长期处于高可用的状态，100%，99.99999%

最理想的状况下，软件的故障，就不应该说导致整个系统的崩溃，服务器硬件的一些故障，服务的冗余

唯一有可能导致系统彻底崩溃，就是类似于之前，支付宝的那个事故，工人施工，挖断了电缆，导致几个机房都停电

不可用，和产生一些故障或者bug的区别

2、高可用系统架构

资源隔离、限流、熔断、降级、运维监控

资源隔离：让你的系统里，某一块东西，在故障的情况下，不会耗尽系统所有的资源，比如线程资源

我实际的项目中的一个case，有一块东西，是要用多线程做一些事情，小伙伴做项目的时候，没有太留神，资源隔离，那块代码，在遇到一些故障的情况下，每个线程在跑的时候，因为那个bug，直接就死循环了，导致那块东西启动了大量的线程，每个线程都死循环

最终导致我的系统资源耗尽，崩溃，不工作，不可用，废掉了

资源隔离，那一块代码，最多最多就是用掉10个线程，不能再多了，就废掉了，限定好的一些资源

限流：高并发的流量涌入进来，比如说突然间一秒钟100万QPS，废掉了，10万QPS进入系统，其他90万QPS被拒绝了

熔断：系统后端的一些依赖，出了一些故障，比如说mysql挂掉了，每次请求都是报错的，熔断了，后续的请求过来直接不接收了，拒绝访问，10分钟之后再尝试去看看mysql恢复没有

降级：mysql挂了，系统发现了，自动降级，从内存里存的少量数据中，去提取一些数据出来

运维监控：监控+报警+优化，各种异常的情况，有问题就及时报警，优化一些系统的配置和参数，或者代码

3、如何讲解这块内容？

（1）如何将eshop-cache，核心的缓存服务改造成高可用的架构

（2）hystrix中的一部分内容，单拉出来，做成一个免费的小课程，作为福利发放出去

（3）eshop-cache，写代码，eshop-cache-ha，业务场景，跟之前衔接起来，重新去写代码

（4）hystrix做服务高可用这一块的内容，讲解成只有一个业务背景，重新写代码，独立

eshop-cache，在各级缓存数据都失效的情况下，会重新从源系统中调用接口，依赖源系统去查询mysql数据库去重新获取数据

如果你的各种依赖的服务有了故障，那么很可能会导致你的系统不可用

hystrix对系统进行各种高可用性的系统加固，来应对各种不可用的情况

缓存雪崩那一块去讲解，redis肯定挂，mysql有较大概率挂掉，在风雨飘摇中

我之前做的一个项目，我们多个项目都用了公司里公用的缓存的存储，缓存彻底挂了，雪崩了，导致各种业务系统全部崩溃，崩溃了好几个小时

导致公司损失了大量的资金的损失

其中导致公司损失最大的负责人，受到了很大的处分

高可用性这个topic，然后咱们会用几讲的时间来讲解一下如何用hystrix，来构建高可用的服务的架构

咱们会用一个真实的项目背景，作为业务场景，来带出来在这个特定的业务场景下，可能会产生哪些各种各样的可用性的一些问题

针对这些问题，我们用hystrix的解决方案和原理是什么

带着大家，纯手工将所有的服务的高可用架构的代码，全部纯手工自己敲出来

形成高可用服务架构的项目实战的一个课程

1、Hystrix是什么？

在分布式系统中，每个服务都可能会调用很多其他服务，被调用的那些服务就是依赖服务，有的时候某些依赖服务出现故障也是很正常的。

Hystrix可以让我们在分布式系统中对服务间的调用进行控制，加入一些调用延迟或者依赖故障的容错机制。

Hystrix通过将依赖服务进行资源隔离，进而组织某个依赖服务出现故障的时候，这种故障在整个系统所有的依赖服务调用中进行蔓延，同时Hystrix还提供故障时的fallback降级机制

总而言之，Hystrix通过这些方法帮助我们提升分布式系统的可用性和稳定性

2、Hystrix的历史

hystrix，就是一种高可用保障的一个框架，类似于spring（ioc，mvc），mybatis，activiti，lucene，框架，预先封装好的为了解决某个特定领域的特定问题的一套代码库

框架，用了框架之后，来解决这个领域的特定的问题，就可以大大减少我们的工作量，提升我们的工作质量和工作效率，框架

hystrix，高可用性保障的一个框架

Netflix（可以认为是国外的优酷或者爱奇艺之类的视频网站），API团队从2011年开始做一些提升系统可用性和稳定性的工作，Hystrix就是从那时候开始发展出来的。

在2012年的时候，Hystrix就变得比较成熟和稳定了，Netflix中，除了API团队以外，很多其他的团队都开始使用Hystrix。

时至今日，Netflix中每天都有数十亿次的服务间调用，通过Hystrix框架在进行，而Hystrix也帮助Netflix网站提升了整体的可用性和稳定性

3、初步看一看Hystrix的设计原则是什么？

hystrix为了实现高可用性的架构，设计hystrix的时候，一些设计原则是什么？？？

（1）对依赖服务调用时出现的调用延迟和调用失败进行控制和容错保护

（2）在复杂的分布式系统中，阻止某一个依赖服务的故障在整个系统中蔓延，服务A->服务B->服务C，服务C故障了，服务B也故障了，服务A故障了，整套分布式系统全部故障，整体宕机

（3）提供fail-fast（快速失败）和快速恢复的支持

（4）提供fallback优雅降级的支持

（5）支持近实时的监控、报警以及运维操作

调用延迟+失败，提供容错

阻止故障蔓延

快速失败+快速恢复

降级

监控+报警+运维

完全描述了hystrix的功能，提供整个分布式系统的高可用的架构

4、Hystrix要解决的问题是什么？

在复杂的分布式系统架构中，每个服务都有很多的依赖服务，而每个依赖服务都可能会故障

如果服务没有和自己的依赖服务进行隔离，那么可能某一个依赖服务的故障就会拖垮当前这个服务

举例来说，某个服务有30个依赖服务，每个依赖服务的可用性非常高，已经达到了99.99%的高可用性

那么该服务的可用性就是99.99%的30次方，也就是99.7%的可用性

99.7%的可用性就意味着3%的请求可能会失败，因为3%的时间内系统可能出现了故障不可用了

对于1亿次访问来说，3%的请求失败，也就意味着300万次请求会失败，也意味着每个月有2个小时的时间系统是不可用的

在真实生产环境中，可能更加糟糕

上面也就是说，即使你每个依赖服务都是99.99%高可用性，但是一旦你有几十个依赖服务，还是会导致你每个月都有几个小时是不可用的

画图分析说，当某一个依赖服务出现了调用延迟或者调用失败时，为什么会拖垮当前这个服务？以及在分布式系统中，故障是如何快速蔓延的？

5、再看Hystrix的更加细节的设计原则是什么？

（1）阻止任何一个依赖服务耗尽所有的资源，比如tomcat中的所有线程资源

（2）避免请求排队和积压，采用限流和fail fast来控制故障

（3）提供fallback降级机制来应对故障

（4）使用资源隔离技术，比如bulkhead（舱壁隔离技术），swimlane（泳道技术），circuit breaker（短路技术），来限制任何一个依赖服务的故障的影响

（5）通过近实时的统计/监控/报警功能，来提高故障发现的速度

（6）通过近实时的属性和配置热修改功能，来提高故障处理和恢复的速度

（7）保护依赖服务调用的所有故障情况，而不仅仅只是网络故障情况

调用这个依赖服务的时候，client调用包有bug，阻塞，等等，依赖服务的各种各样的调用的故障，都可以处理

6、Hystrix是如何实现它的目标的？

（1）通过HystrixCommand或者HystrixObservableCommand来封装对外部依赖的访问请求，这个访问请求一般会运行在独立的线程中，资源隔离

（2）对于超出我们设定阈值的服务调用，直接进行超时，不允许其耗费过长时间阻塞住。这个超时时间默认是99.5%的访问时间，但是一般我们可以自己设置一下

（3）为每一个依赖服务维护一个独立的线程池，或者是semaphore，当线程池已满时，直接拒绝对这个服务的调用

（4）对依赖服务的调用的成功次数，失败次数，拒绝次数，超时次数，进行统计

（5）如果对一个依赖服务的调用失败次数超过了一定的阈值，自动进行熔断，在一定时间内对该服务的调用直接降级，一段时间后再自动尝试恢复

（6）当一个服务调用出现失败，被拒绝，超时，短路等异常情况时，自动调用fallback降级机制

（7）对属性和配置的修改提供近实时的支持

画图分析，对依赖进行资源隔离后，如何避免依赖服务调用延迟或失败导致当前服务的故障

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

我们这个课程，基于hystrix，如何来构建高可用的分布式系统的架构，项目实战

模拟真实业务的这么一个小型的项目，来全程贯穿，用这个项目中的业务场景去一个一个的讲解hystrix高可用的每个技术

纯讲hystrix，脱离实际的业务背景，听起来有点枯燥，大家学完了hystrix以后，可能没法完全感受到技术是如何融入我们的项目中的

大背景：电商网站，首页，商品详情页，搜索结果页，广告页，促销活动，购物车，订单系统，库存系统，物流系统

小背景：商品详情页，如何用最快的结果将商品数据填充到一个页面中，然后将页面显示出来

分布式系统：商品详情页，缓存服务，+底层源数据服务，商品信息服务，店铺信息服务，广告信息服务，推荐信息服务，综合起来组成一个分布式的系统

1、电商网站的商品详情页系统架构

（1）小型电商网站的商品详情页系统架构（不是我们要讲解的）

（2）大型电商网站的商品详情页系统架构

（3）页面模板

举个例子

将数据动态填充/渲染到一个html模板中，是什么意思呢？

<html>

<title>#{name}的页面</title>

<body>

商品的价格是：#{price}

商品的介绍：#{description}

</body>

</html>

上面这个就可以认为是一个页面模板，里面的很多内容是不确定的，#{name}，#{price}，#{description}，这都是一些模板脚本，不确定里面的值是什么？

将数据填充/渲染到html模板中，是什么意思呢？

{

"name": "iphone7 plus（玫瑰金+32G）",

"price": 5599.50

"description": "这个手机特别好用。。。。。。"

}

<html>

<title>iphone7 plus（玫瑰金+32G）的页面</title>

<body>

商品的价格是：5599.50

商品的介绍：这个手机特别好用。。。。。。

</body>

</html>

上面这个就是一份填充好数据的一个html页面

2、缓存服务

缓存服务，订阅一个MQ的消息变更，如果有消息变更的话，那么就会发送一个网络请求，调用一个底层的对应的源数据服务的接口，去获取变更后的数据

将获取到的变更后的数据填充到分布式的redis缓存中去

高可用这一块儿，最可能出现说可用性不高的情况，是什么呢？就是说，在接收到消息之后，可能在调用各种底层依赖服务的接口时，会遇到各种不稳定的情况

比如底层服务的接口调用超时，200ms，2s都没有返回; 底层服务的接口调用失败，比如说卡了500ms之后，返回一个报错

在分布式系统中，对于这种大量的底层依赖服务的调用，就可能会出现各种可用性的问题，一旦没有处理好的话

可能就会导致缓存服务自己本身会挂掉，或者故障掉，就会导致什么呢？不可以对外提供服务，严重情况下，甚至会导致说整个商品详情页显示不出来

缓存服务接收到变更消息后，去调用各个底层依赖服务时的高可用架构的实现

我们刚才讲解的整套大型电商网站的商品详情页的缓存架构，完整的那个流程，《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

3、框架结构

围绕着缓存服务去拉取各种底层的源数据服务的数据，调用其接口时，可能出现的系统不可用的问题

从简

spring boot，微服务的非常快速，非常好用的技术框架，脱胎于spring，具体的东西就不讲解，直接带着大家上手搭建一个spring boot的框架

2个服务，缓存服务，商品服务，缓存服务依赖于商品服务

模拟各种商品服务可能接口调用时出现的各种问题，导致系统不可用的场景，然后用hystrix完整的各种技术点全部贯穿在里面

解决了一大堆设计业务背景下的系统不可用问题，hystrix整个技术体系，知识体系，也就讲解完了

消息队列，redis，咱们都不搞了

分布式系统，微服务，dubbo，不用dubbo，目前比较明显的一个趋势是，行业里，未来主要还是spring boot，spring cloud，主流的开源技术，去构建微服务的分布式系统

基于dubbo，官方很久之前就停止更新了，支持也不是太好

spring boot + http client + hystrix

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.2.5.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jdbc</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.1.43</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestone</id>

<url>https://repo.spring.io/libs-release</url>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-milestone</id>

<url>https://repo.spring.io/libs-release</url>

</pluginRepository>

</pluginRepositories>

你实际在你本地去搭建这个工程的时候，你首先就会发现说，你一修改这个pom.xml，发现下载各种spring boot依赖包，下载巨慢巨慢

北京，宽带，100M，联通，还是下载的巨慢

手工下载依赖，并安装到本地maven仓库

（1）在maven中央仓库搜索jar包，如果没有找到，就得手动在百度里面找，下载jar下来

（2）根据jar对应的group id，artifact id，找到自己本地的maven仓库，对应的目录，将jar包拷贝到那个目录里面去

jmxtool，groupId=com.sun.jdmk，artifactId=jmxtools，version=1.2.1

com\sun\jdmk\jmxtools\1.2.1

（3）手工执行mvn install:install-file的命令，在本地仓库中安装这个依赖

mvn install:install-file -Dfile=E:\apache-maven-3.0.5\mvn\_repo\com\sun\jdmk\jmxtools\1.2.1\jmxtools-1.2.1.jar -DgroupId=com.sun.jdmk -DartifactId=jmxtools -Dversion=1.2.1 -Dpackaging=jar

（4）强制kill掉你的eclipse

（5）重新再进入eclips，这个时候肯定是会报很多的错误的，重新加载maven依赖

（6）反复循环，手工下载了，十几个到二十个依赖，然后最终所有的依赖全部成功下载到了本地，工程部报错

2、配置文件（src/main/resources）

Application.properties

server.port=8081

spring.datasource.url=jdbc:mysql://192.168.31.85:3306/eshop

spring.datasource.username=eshop

spring.datasource.password=eshop

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

说明：我已经在一个虚拟机中，安装好了一个mysql数据库，大家需要自己在自己本地安装一个mysql，配置好对应的url连接串，还有对应的用户名和密码就可以了

怎么安装mysql，大家自己网上查一下吧，java工程师，大学的学生，自己在本地安装一个mysql还是可以搞定的吧

mybatis/UserMappper.xml

templates/hello.html

3、Application

@EnableAutoConfiguration

@SpringBootApplication

@ComponentScan

@MapperScan("com.roncoo.eshop.cache.mapper")

public class Application {

@Bean

@ConfigurationProperties(prefix="spring.datasource")

public DataSource dataSource() {

return new org.apache.tomcat.jdbc.pool.DataSource();

}

@Bean

public SqlSessionFactory sqlSessionFactoryBean() throws Exception {

SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();

sqlSessionFactoryBean.setDataSource(dataSource());

PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();

sqlSessionFactoryBean.setMapperLocations(resolver.getResources("classpath:/mybatis/\*.xml"));

return sqlSessionFactoryBean.getObject();

}

@Bean

public PlatformTransactionManager transactionManager() {

return new DataSourceTransactionManager(dataSource());

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

4、HelloController

5、完成两个服务的构建

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

快速将核心的功能流程，用代码来实现

从下一讲开始，然后我们其实就针对这里面的一些东西，来给大家讲解哪些地方可能会有可用性的问题，如何用hystrix来解决这些可用性的问题

1、接收数据变更的消息，订阅一个MQ的topic，但是我们这里就简化一下，采取提供一个http接口

2、往http接口发送一条消息，就认为是通知缓存服务，有一个商品的数据变更了

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.4</version>

</dependency>

/\*\*

\* HttpClient工具类

\* @author lixuerui

\*

\*/

@SuppressWarnings("deprecation")

public class HttpClientUtils {

/\*\*

\* 发送GET请求

\* @param url 请求URL

\* @return 响应结果

\*/

@SuppressWarnings("resource")

public static String sendGetRequest(String url) {

String httpResponse = null;

HttpClient httpclient = null;

InputStream is = null;

BufferedReader br = null;

try {

// 发送GET请求

httpclient = new DefaultHttpClient();

HttpGet httpget = new HttpGet(url);

HttpResponse response = httpclient.execute(httpget);

// 处理响应

HttpEntity entity = response.getEntity();

if (entity != null) {

is = entity.getContent();

br = new BufferedReader(new InputStreamReader(is));

StringBuffer buffer = new StringBuffer("");

String line = null;

while ((line = br.readLine()) != null) {

buffer.append(line + "\n");

}

httpResponse = buffer.toString();

}

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

if(br != null) {

br.close();

}

if(is != null) {

is.close();

}

} catch (Exception e2) {

e2.printStackTrace();

}

}

return httpResponse;

}

/\*\*

\* 发送post请求

\* @param url URL

\* @param map 参数Map

\* @return

\*/

@SuppressWarnings({ "rawtypes", "unchecked", "resource" })

public static String sendPostRequest(String url, Map<String,String> map){

HttpClient httpClient = null;

HttpPost httpPost = null;

String result = null;

try{

httpClient = new DefaultHttpClient();

httpPost = new HttpPost(url);

//设置参数

List<NameValuePair> list = new ArrayList<NameValuePair>();

Iterator iterator = map.entrySet().iterator();

while(iterator.hasNext()){

Entry<String,String> elem = (Entry<String, String>) iterator.next();

list.add(new BasicNameValuePair(elem.getKey(), elem.getValue()));

}

if(list.size() > 0){

UrlEncodedFormEntity entity = new UrlEncodedFormEntity(list, "utf-8");

httpPost.setEntity(entity);

}

HttpResponse response = httpClient.execute(httpPost);

if(response != null){

HttpEntity resEntity = response.getEntity();

if(resEntity != null){

result = EntityUtils.toString(resEntity, "utf-8");

}

}

} catch(Exception ex){

ex.printStackTrace();

} finally {

}

return result;

}

}

3、缓存服务接收到这条消息之后，就会去通过http调用商品服务的一个接口，获取到商品变更后的最新数据

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、pom.xml

<dependency>

<groupId>com.netflix.hystrix</groupId>

<artifactId>hystrix-core</artifactId>

<version>1.5.12</version>

</dependency>

2、将商品服务接口调用的逻辑进行封装

hystrix进行资源隔离，其实是提供了一个抽象，叫做command，就是说，你如果要把对某一个依赖服务的所有调用请求，全部隔离在同一份资源池内

对这个依赖服务的所有调用请求，全部走这个资源池内的资源，不会去用其他的资源了，这个就叫做资源隔离

hystrix最最基本的资源隔离的技术，线程池隔离技术

对某一个依赖服务，商品服务，所有的调用请求，全部隔离到一个线程池内，对商品服务的每次调用请求都封装在一个command里面

每个command（每次服务调用请求）都是使用线程池内的一个线程去执行的

所以哪怕是对这个依赖服务，商品服务，现在同时发起的调用量已经到了1000了，但是线程池内就10个线程，最多就只会用这10个线程去执行

不会说，对商品服务的请求，因为接口调用延迟，将tomcat内部所有的线程资源全部耗尽，不会出现了

public class CommandHelloWorld extends HystrixCommand<String> {

private final String name;

public CommandHelloWorld(String name) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.name = name;

}

@Override

protected String run() {

return "Hello " + name + "!";

}

}

不让超出这个量的请求去执行了，保护说，不要因为某一个依赖服务的故障，导致耗尽了缓存服务中的所有的线程资源去执行

3、开发一个支持批量商品变更的接口

HystrixCommand：是用来获取一条数据的

HystrixObservableCommand：是设计用来获取多条数据的

public class ObservableCommandHelloWorld extends HystrixObservableCommand<String> {

private final String name;

public ObservableCommandHelloWorld(String name) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.name = name;

}

@Override

protected Observable<String> construct() {

return Observable.create(new Observable.OnSubscribe<String>() {

@Override

public void call(Subscriber<? super String> observer) {

try {

if (!observer.isUnsubscribed()) {

observer.onNext("Hello " + name + "!");

observer.onNext("Hi " + name + "!");

observer.onCompleted();

}

} catch (Exception e) {

observer.onError(e);

}

}

} ).subscribeOn(Schedulers.io());

}

}

4、command的四种调用方式

同步：new CommandHelloWorld("World").execute()，new ObservableCommandHelloWorld("World").toBlocking().toFuture().get()

如果你认为observable command只会返回一条数据，那么可以调用上面的模式，去同步执行，返回一条数据

异步：new CommandHelloWorld("World").queue()，new ObservableCommandHelloWorld("World").toBlocking().toFuture()

对command调用queue()，仅仅将command放入线程池的一个等待队列，就立即返回，拿到一个Future对象，后面可以做一些其他的事情，然后过一段时间对future调用get()方法获取数据

// observe()：hot，已经执行过了

// toObservable(): cold，还没执行过

Observable<String> fWorld = new CommandHelloWorld("World").observe();

assertEquals("Hello World!", fWorld.toBlocking().single());

fWorld.subscribe(new Observer<String>() {

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

e.printStackTrace();

}

@Override

public void onNext(String v) {

System.out.println("onNext: " + v);

}

});

Observable<String> fWorld = new ObservableCommandHelloWorld("World").toObservable();

assertEquals("Hello World!", fWorld.toBlocking().single());

fWorld.subscribe(new Observer<String>() {

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

e.printStackTrace();

}

@Override

public void onNext(String v) {

System.out.println("onNext: " + v);

}

});

5、如何解决刚才的问题

画图讲解资源隔离后的效果

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、线程池隔离技术与信号量隔离技术的区别

hystrix里面，核心的一项功能，其实就是所谓的资源隔离，要解决的最最核心的问题，就是将多个依赖服务的调用分别隔离到各自自己的资源池内

避免说对某一个依赖服务的调用，因为依赖服务的接口调用的延迟或者失败，导致服务所有的线程资源全部耗费在这个服务的接口调用上

一旦说某个服务的线程资源全部耗尽的话，可能就导致服务就会崩溃，甚至说这种故障会不断蔓延

hystrix，资源隔离，两种技术，线程池的资源隔离，信号量的资源隔离

信号量，semaphore

信号量跟线程池，两种资源隔离的技术，区别到底在哪儿呢？

2、线程池隔离技术和信号量隔离技术，分别在什么样的场景下去使用呢？？

线程池：适合绝大多数的场景，99%的，线程池，对依赖服务的网络请求的调用和访问，timeout这种问题

信号量：适合，你的访问不是对外部依赖的访问，而是对内部的一些比较复杂的业务逻辑的访问，但是像这种访问，系统内部的代码，其实不涉及任何的网络请求，那么只要做信号量的普通限流就可以了，因为不需要去捕获timeout类似的问题，算法+数据结构的效率不是太高，并发量突然太高，因为这里稍微耗时一些，导致很多线程卡在这里的话，不太好，所以进行一个基本的资源隔离和访问，避免内部复杂的低效率的代码，导致大量的线程被hang住

3、在代码中加入从本地内存获取地理位置数据的逻辑

业务背景里面， 比较适合信号量的是什么场景呢？

比如说，我们一般来说，缓存服务，可能会将部分量特别少，访问又特别频繁的一些数据，放在自己的纯内存中

一般我们在获取到商品数据之后，都要去获取商品是属于哪个地理位置，省，市，卖家的，可能在自己的纯内存中，比如就一个Map去获取

对于这种直接访问本地内存的逻辑，比较适合用信号量做一下简单的隔离

优点在于，不用自己管理线程池拉，不用care timeout超时了，信号量做隔离的话，性能会相对来说高一些

4、采用信号量技术对地理位置获取逻辑进行资源隔离与限流

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandPropertiesDefaults(HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)));

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

资源隔离，两种策略，线程池隔离，信号量隔离

对资源隔离这一块东西，做稍微更加深入一些的讲解，告诉你，除了可以选择隔离策略以外，对你选择的隔离策略，可以做一定的细粒度的一些控制

1、execution.isolation.strategy

指定了HystrixCommand.run()的资源隔离策略，THREAD或者SEMAPHORE，一种是基于线程池，一种是信号量

线程池机制，每个command运行在一个线程中，限流是通过线程池的大小来控制的

信号量机制，command是运行在调用线程中，但是通过信号量的容量来进行限流

如何在线程池和信号量之间做选择？

默认的策略就是线程池

线程池其实最大的好处就是对于网络访问请求，如果有超时的话，可以避免调用线程阻塞住

而使用信号量的场景，通常是针对超大并发量的场景下，每个服务实例每秒都几百的QPS，那么此时你用线程池的话，线程一般不会太多，可能撑不住那么高的并发，如果要撑住，可能要耗费大量的线程资源，那么就是用信号量，来进行限流保护

一般用信号量常见于那种基于纯内存的一些业务逻辑服务，而不涉及到任何网络访问请求

netflix有100+的command运行在40+的线程池中，只有少数command是不运行在线程池中的，就是从纯内存中获取一些元数据，或者是对多个command包装起来的facacde command，是用信号量限流的

// to use thread isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.THREAD)

// to use semaphore isolation

HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)

2、command名称和command组

线程池隔离，依赖服务->接口->线程池，如何来划分

你的每个command，都可以设置一个自己的名称，同时可以设置一个自己的组

private static final Setter cachedSetter =

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"));

public CommandHelloWorld(String name) {

super(cachedSetter);

this.name = name;

}

command group，是一个非常重要的概念，默认情况下，因为就是通过command group来定义一个线程池的，而且还会通过command group来聚合一些监控和报警信息

同一个command group中的请求，都会进入同一个线程池中

3、command线程池

threadpool key代表了一个HystrixThreadPool，用来进行统一监控，统计，缓存

默认的threadpool key就是command group名称

每个command都会跟它的threadpool key对应的thread pool绑定在一起

如果不想直接用command group，也可以手动设置thread pool name

public CommandHelloWorld(String name) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));

this.name = name;

}

command threadpool -> command group -> command key

command key，代表了一类command，一般来说，代表了底层的依赖服务的一个接口

command group，代表了某一个底层的依赖服务，合理，一个依赖服务可能会暴露出来多个接口，每个接口就是一个command key

command group，在逻辑上去组织起来一堆command key的调用，统计信息，成功次数，timeout超时次数，失败次数，可以看到某一个服务整体的一些访问情况

command group，一般来说，推荐是根据一个服务去划分出一个线程池，command key默认都是属于同一个线程池的

比如说你以一个服务为粒度，估算出来这个服务每秒的所有接口加起来的整体QPS在100左右

你调用那个服务的当前服务，部署了10个服务实例，每个服务实例上，其实用这个command group对应这个服务，给一个线程池，量大概在10个左右，就可以了，你对整个服务的整体的访问QPS大概在每秒100左右

一般来说，command group是用来在逻辑上组合一堆command的

举个例子，对于一个服务中的某个功能模块来说，希望将这个功能模块内的所有command放在一个group中，那么在监控和报警的时候可以放一起看

command group，对应了一个服务，但是这个服务暴露出来的几个接口，访问量很不一样，差异非常之大

你可能就希望在这个服务command group内部，包含的对应多个接口的command key，做一些细粒度的资源隔离

对同一个服务的不同接口，都使用不同的线程池

command key -> command group

command key -> 自己的threadpool key

逻辑上来说，多个command key属于一个command group，在做统计的时候，会放在一起统计

每个command key有自己的线程池，每个接口有自己的线程池，去做资源隔离和限流

但是对于thread pool资源隔离来说，可能是希望能够拆分的更加一致一些，比如在一个功能模块内，对不同的请求可以使用不同的thread pool

command group一般来说，可以是对应一个服务，多个command key对应这个服务的多个接口，多个接口的调用共享同一个线程池

如果说你的command key，要用自己的线程池，可以定义自己的threadpool key，就ok了

4、coreSize

设置线程池的大小，默认是10

HystrixThreadPoolProperties.Setter()

.withCoreSize(int value)

一般来说，用这个默认的10个线程大小就够了

5、queueSizeRejectionThreshold

控制queue满后reject的threshold，因为maxQueueSize不允许热修改，因此提供这个参数可以热修改，控制队列的最大大小

HystrixCommand在提交到线程池之前，其实会先进入一个队列中，这个队列满了之后，才会reject

默认值是5

HystrixThreadPoolProperties.Setter()

.withQueueSizeRejectionThreshold(int value)

6、execution.isolation.semaphore.maxConcurrentRequests

设置使用SEMAPHORE隔离策略的时候，允许访问的最大并发量，超过这个最大并发量，请求直接被reject

这个并发量的设置，跟线程池大小的设置，应该是类似的，但是基于信号量的话，性能会好很多，而且hystrix框架本身的开销会小很多

默认值是10，设置的小一些，否则因为信号量是基于调用线程去执行command的，而且不能从timeout中抽离，因此一旦设置的太大，而且有延时发生，可能瞬间导致tomcat本身的线程资源本占满

HystrixCommandProperties.Setter()

.withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

之前几讲，我们用实际的业务背景给了一些可用性的问题

然后借着那些最最基础的可用性的问题，然后讲解了hystrix最基本的支持高可用的技术，资源隔离+限流

创建command，执行这个command，配置这个command对应的group和线程池，以及线程池/信号量的容量和大小

我们要去讲解一下，你开始执行这个command，调用了这个command的execute()方法以后，hystrix内部的底层的执行流程和步骤以及原理是什么呢？

在讲解这个流程的过程中，我们会带出来hystrix其他的一些核心以及重要的功能

画图分析整个8大步骤的流程，然后再对每个步骤进行细致的讲解

1、构建一个HystrixCommand或者HystrixObservableCommand

一个HystrixCommand或一个HystrixObservableCommand对象，代表了对某个依赖服务发起的一次请求或者调用

构造的时候，可以在构造函数中传入任何需要的参数

HystrixCommand主要用于仅仅会返回一个结果的调用

HystrixObservableCommand主要用于可能会返回多条结果的调用

HystrixCommand command = new HystrixCommand(arg1, arg2);

HystrixObservableCommand command = new HystrixObservableCommand(arg1, arg2);

2、调用command的执行方法

执行Command就可以发起一次对依赖服务的调用

要执行Command，需要在4个方法中选择其中的一个：execute()，queue()，observe()，toObservable()

其中execute()和queue()仅仅对HystrixCommand适用

execute()：调用后直接block住，属于同步调用，直到依赖服务返回单条结果，或者抛出异常

queue()：返回一个Future，属于异步调用，后面可以通过Future获取单条结果

observe()：订阅一个Observable对象，Observable代表的是依赖服务返回的结果，获取到一个那个代表结果的Observable对象的拷贝对象

toObservable()：返回一个Observable对象，如果我们订阅这个对象，就会执行command并且获取返回结果

K value = command.execute();

Future<K> fValue = command.queue();

Observable<K> ohValue = command.observe();

Observable<K> ocValue = command.toObservable();

execute()实际上会调用queue().get().queue()，接着会调用toObservable().toBlocking().toFuture()

也就是说，无论是哪种执行command的方式，最终都是依赖toObservable()去执行的

3、检查是否开启缓存

从这一步开始，进入我们的底层的运行原理啦，了解hysrix的一些更加高级的功能和特性

如果这个command开启了请求缓存，request cache，而且这个调用的结果在缓存中存在，那么直接从缓存中返回结果

4、检查是否开启了短路器

检查这个command对应的依赖服务是否开启了短路器

如果断路器被打开了，那么hystrix就不会执行这个command，而是直接去执行fallback降级机制

5、检查线程池/队列/semaphore是否已经满了

如果command对应的线程池/队列/semaphore已经满了，那么也不会执行command，而是直接去调用fallback降级机制

6、执行command

调用HystrixObservableCommand.construct()或HystrixCommand.run()来实际执行这个command

HystrixCommand.run()是返回一个单条结果，或者抛出一个异常

HystrixObservableCommand.construct()是返回一个Observable对象，可以获取多条结果

如果HystrixCommand.run()或HystrixObservableCommand.construct()的执行，超过了timeout时长的话，那么command所在的线程就会抛出一个TimeoutException

如果timeout了，也会去执行fallback降级机制，而且就不会管run()或construct()返回的值了

这里要注意的一点是，我们是不可能终止掉一个调用严重延迟的依赖服务的线程的，只能说给你抛出来一个TimeoutException，但是还是可能会因为严重延迟的调用线程占满整个线程池的

即使这个时候新来的流量都被限流了。。。

如果没有timeout的话，那么就会拿到一些调用依赖服务获取到的结果，然后hystrix会做一些logging记录和metric统计

7、短路健康检查

Hystrix会将每一个依赖服务的调用成功，失败，拒绝，超时，等事件，都会发送给circuit breaker断路器

短路器就会对调用成功/失败/拒绝/超时等事件的次数进行统计

短路器会根据这些统计次数来决定，是否要进行短路，如果打开了短路器，那么在一段时间内就会直接短路，然后如果在之后第一次检查发现调用成功了，就关闭断路器

8、调用fallback降级机制

在以下几种情况中，hystrix会调用fallback降级机制：run()或construct()抛出一个异常，短路器打开，线程池/队列/semaphore满了，command执行超时了

一般在降级机制中，都建议给出一些默认的返回值，比如静态的一些代码逻辑，或者从内存中的缓存中提取一些数据，尽量在这里不要再进行网络请求了

即使在降级中，一定要进行网络调用，也应该将那个调用放在一个HystrixCommand中，进行隔离

在HystrixCommand中，上线getFallback()方法，可以提供降级机制

在HystirxObservableCommand中，实现一个resumeWithFallback()方法，返回一个Observable对象，可以提供降级结果

如果fallback返回了结果，那么hystrix就会返回这个结果

对于HystrixCommand，会返回一个Observable对象，其中会发返回对应的结果

对于HystrixObservableCommand，会返回一个原始的Observable对象

如果没有实现fallback，或者是fallback抛出了异常，Hystrix会返回一个Observable，但是不会返回任何数据

不同的command执行方式，其fallback为空或者异常时的返回结果不同

对于execute()，直接抛出异常

对于queue()，返回一个Future，调用get()时抛出异常

对于observe()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

对于toObservable()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

9、不同的执行方式

execute()，获取一个Future.get()，然后拿到单个结果

queue()，返回一个Future

observer()，立即订阅Observable，然后启动8大执行步骤，返回一个拷贝的Observable，订阅时理解回调给你结果

toObservable()，返回一个原始的Observable，必须手动订阅才会去执行8大步骤

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

之前几讲，我们用实际的业务背景给了一些可用性的问题

然后借着那些最最基础的可用性的问题，然后讲解了hystrix最基本的支持高可用的技术，资源隔离+限流

创建command，执行这个command，配置这个command对应的group和线程池，以及线程池/信号量的容量和大小

我们要去讲解一下，你开始执行这个command，调用了这个command的execute()方法以后，hystrix内部的底层的执行流程和步骤以及原理是什么呢？

在讲解这个流程的过程中，我们会带出来hystrix其他的一些核心以及重要的功能

画图分析整个8大步骤的流程，然后再对每个步骤进行细致的讲解

1、构建一个HystrixCommand或者HystrixObservableCommand

一个HystrixCommand或一个HystrixObservableCommand对象，代表了对某个依赖服务发起的一次请求或者调用

构造的时候，可以在构造函数中传入任何需要的参数

HystrixCommand主要用于仅仅会返回一个结果的调用

HystrixObservableCommand主要用于可能会返回多条结果的调用

HystrixCommand command = new HystrixCommand(arg1, arg2);

HystrixObservableCommand command = new HystrixObservableCommand(arg1, arg2);

2、调用command的执行方法

执行Command就可以发起一次对依赖服务的调用

要执行Command，需要在4个方法中选择其中的一个：execute()，queue()，observe()，toObservable()

其中execute()和queue()仅仅对HystrixCommand适用

execute()：调用后直接block住，属于同步调用，直到依赖服务返回单条结果，或者抛出异常

queue()：返回一个Future，属于异步调用，后面可以通过Future获取单条结果

observe()：订阅一个Observable对象，Observable代表的是依赖服务返回的结果，获取到一个那个代表结果的Observable对象的拷贝对象

toObservable()：返回一个Observable对象，如果我们订阅这个对象，就会执行command并且获取返回结果

K value = command.execute();

Future<K> fValue = command.queue();

Observable<K> ohValue = command.observe();

Observable<K> ocValue = command.toObservable();

execute()实际上会调用queue().get().queue()，接着会调用toObservable().toBlocking().toFuture()

也就是说，无论是哪种执行command的方式，最终都是依赖toObservable()去执行的

3、检查是否开启缓存

从这一步开始，进入我们的底层的运行原理啦，了解hysrix的一些更加高级的功能和特性

如果这个command开启了请求缓存，request cache，而且这个调用的结果在缓存中存在，那么直接从缓存中返回结果

4、检查是否开启了短路器

检查这个command对应的依赖服务是否开启了短路器

如果断路器被打开了，那么hystrix就不会执行这个command，而是直接去执行fallback降级机制

5、检查线程池/队列/semaphore是否已经满了

如果command对应的线程池/队列/semaphore已经满了，那么也不会执行command，而是直接去调用fallback降级机制

6、执行command

调用HystrixObservableCommand.construct()或HystrixCommand.run()来实际执行这个command

HystrixCommand.run()是返回一个单条结果，或者抛出一个异常

HystrixObservableCommand.construct()是返回一个Observable对象，可以获取多条结果

如果HystrixCommand.run()或HystrixObservableCommand.construct()的执行，超过了timeout时长的话，那么command所在的线程就会抛出一个TimeoutException

如果timeout了，也会去执行fallback降级机制，而且就不会管run()或construct()返回的值了

这里要注意的一点是，我们是不可能终止掉一个调用严重延迟的依赖服务的线程的，只能说给你抛出来一个TimeoutException，但是还是可能会因为严重延迟的调用线程占满整个线程池的

即使这个时候新来的流量都被限流了。。。

如果没有timeout的话，那么就会拿到一些调用依赖服务获取到的结果，然后hystrix会做一些logging记录和metric统计

7、短路健康检查

Hystrix会将每一个依赖服务的调用成功，失败，拒绝，超时，等事件，都会发送给circuit breaker断路器

短路器就会对调用成功/失败/拒绝/超时等事件的次数进行统计

短路器会根据这些统计次数来决定，是否要进行短路，如果打开了短路器，那么在一段时间内就会直接短路，然后如果在之后第一次检查发现调用成功了，就关闭断路器

8、调用fallback降级机制

在以下几种情况中，hystrix会调用fallback降级机制：run()或construct()抛出一个异常，短路器打开，线程池/队列/semaphore满了，command执行超时了

一般在降级机制中，都建议给出一些默认的返回值，比如静态的一些代码逻辑，或者从内存中的缓存中提取一些数据，尽量在这里不要再进行网络请求了

即使在降级中，一定要进行网络调用，也应该将那个调用放在一个HystrixCommand中，进行隔离

在HystrixCommand中，上线getFallback()方法，可以提供降级机制

在HystirxObservableCommand中，实现一个resumeWithFallback()方法，返回一个Observable对象，可以提供降级结果

如果fallback返回了结果，那么hystrix就会返回这个结果

对于HystrixCommand，会返回一个Observable对象，其中会发返回对应的结果

对于HystrixObservableCommand，会返回一个原始的Observable对象

如果没有实现fallback，或者是fallback抛出了异常，Hystrix会返回一个Observable，但是不会返回任何数据

不同的command执行方式，其fallback为空或者异常时的返回结果不同

对于execute()，直接抛出异常

对于queue()，返回一个Future，调用get()时抛出异常

对于observe()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

对于toObservable()，返回一个Observable对象，但是调用subscribe()方法订阅它时，理解抛出调用者的onError方法

9、不同的执行方式

execute()，获取一个Future.get()，然后拿到单个结果

queue()，返回一个Future

observer()，立即订阅Observable，然后启动8大执行步骤，返回一个拷贝的Observable，订阅时理解回调给你结果

toObservable()，返回一个原始的Observable，必须手动订阅才会去执行8大步骤

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

我们上一讲讲解的那个图片，顺着那个图片的流程，来一个一个的讲解hystrix的核心技术

1、创建command，2种command类型

2、执行command，4种执行方式

3、查找是否开启了request cache，是否有请求缓存，如果有缓存，直接取用缓存，返回结果

首先，有一个概念，叫做reqeust context，请求上下文，一般来说，在一个web应用中，hystrix

我们会在一个filter里面，对每一个请求都施加一个请求上下文，就是说，tomcat容器内，每一次请求，就是一次请求上下文

然后在这次请求上下文中，我们会去执行N多代码，调用N多依赖服务，有的依赖服务可能还会调用好几次

在一次请求上下文中，如果有多个command，参数都是一样的，调用的接口也是一样的，其实结果可以认为也是一样的

那么这个时候，我们就可以让第一次command执行，返回的结果，被缓存在内存中，然后这个请求上下文中，后续的其他对这个依赖的调用全部从内存中取用缓存结果就可以了

不用在一次请求上下文中反复多次的执行一样的command，提升整个请求的性能

HystrixCommand和HystrixObservableCommand都可以指定一个缓存key，然后hystrix会自动进行缓存，接着在同一个request context内，再次访问的时候，就会直接取用缓存

用请求缓存，可以避免重复执行网络请求

多次调用一个command，那么只会执行一次，后面都是直接取缓存

对于请求缓存（request caching），请求合并（request collapsing），请求日志（request log），等等技术，都必须自己管理HystrixReuqestContext的声明周期

在一个请求执行之前，都必须先初始化一个request context

HystrixRequestContext context = HystrixRequestContext.initializeContext();

然后在请求结束之后，需要关闭request context

context.shutdown();

一般来说，在java web来的应用中，都是通过filter过滤器来实现的

public class HystrixRequestContextServletFilter implements Filter {

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HystrixRequestContext context = HystrixRequestContext.initializeContext();

try {

chain.doFilter(request, response);

} finally {

context.shutdown();

}

}

}

@Bean

public FilterRegistrationBean indexFilterRegistration() {

FilterRegistrationBean registration = new FilterRegistrationBean(new IndexFilter());

registration.addUrlPatterns("/");

return registration;

}

结合咱们的业务背景，我们做了一个批量查询商品数据的接口，在这个里面，我们其实通过HystrixObservableCommand一次性批量查询多个商品id的数据

但是这里有个问题，如果说nginx在本地缓存失效了，重新获取一批缓存，传递过来的productId都没有进行去重，1,1,2,2,5,6,7

那么可能说，商品id出现了重复，如果按照我们之前的业务逻辑，可能就会重复对productId=1的商品查询两次，productId=2的商品查询两次

我们对批量查询商品数据的接口，可以用request cache做一个优化，就是说一次请求，就是一次request context，对相同的商品查询只能执行一次，其余的都走request cache

public class CommandUsingRequestCache extends HystrixCommand<Boolean> {

private final int value;

protected CommandUsingRequestCache(int value) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.value = value;

}

@Override

protected Boolean run() {

return value == 0 || value % 2 == 0;

}

@Override

protected String getCacheKey() {

return String.valueOf(value);

}

}

@Test

public void testWithCacheHits() {

HystrixRequestContext context = HystrixRequestContext.initializeContext();

try {

CommandUsingRequestCache command2a = new CommandUsingRequestCache(2);

CommandUsingRequestCache command2b = new CommandUsingRequestCache(2);

assertTrue(command2a.execute());

// this is the first time we've executed this command with

// the value of "2" so it should not be from cache

assertFalse(command2a.isResponseFromCache());

assertTrue(command2b.execute());

// this is the second time we've executed this command with

// the same value so it should return from cache

assertTrue(command2b.isResponseFromCache());

} finally {

context.shutdown();

}

// start a new request context

context = HystrixRequestContext.initializeContext();

try {

CommandUsingRequestCache command3b = new CommandUsingRequestCache(2);

assertTrue(command3b.execute());

// this is a new request context so this

// should not come from cache

assertFalse(command3b.isResponseFromCache());

} finally {

context.shutdown();

}

}

缓存的手动清理

public static class GetterCommand extends HystrixCommand<String> {

private static final HystrixCommandKey GETTER\_KEY = HystrixCommandKey.Factory.asKey("GetterCommand");

private final int id;

public GetterCommand(int id) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GetSetGet"))

.andCommandKey(GETTER\_KEY));

this.id = id;

}

@Override

protected String run() {

return prefixStoredOnRemoteDataStore + id;

}

@Override

protected String getCacheKey() {

return String.valueOf(id);

}

/\*\*

\* Allow the cache to be flushed for this object.

\*

\* @param id

\* argument that would normally be passed to the command

\*/

public static void flushCache(int id) {

HystrixRequestCache.getInstance(GETTER\_KEY,

HystrixConcurrencyStrategyDefault.getInstance()).clear(String.valueOf(id));

}

}

public static class SetterCommand extends HystrixCommand<Void> {

private final int id;

private final String prefix;

public SetterCommand(int id, String prefix) {

super(HystrixCommandGroupKey.Factory.asKey("GetSetGet"));

this.id = id;

this.prefix = prefix;

}

@Override

protected Void run() {

// persist the value against the datastore

prefixStoredOnRemoteDataStore = prefix;

// flush the cache

GetterCommand.flushCache(id);

// no return value

return null;

}

}

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、创建command

2、执行command

3、request cache

4、短路器，如果打开了，fallback降级机制

1、fallback降级机制

hystrix调用各种接口，或者访问外部依赖，mysql，redis，zookeeper，kafka，等等，如果出现了任何异常的情况

比如说报错了，访问mysql报错，redis报错，zookeeper报错，kafka报错，error

对每个外部依赖，无论是服务接口，中间件，资源隔离，对外部依赖只能用一定量的资源去访问，线程池/信号量，如果资源池已满，reject

访问外部依赖的时候，访问时间过长，可能就会导致超时，报一个TimeoutException异常，timeout

上述三种情况，都是我们说的异常情况，对外部依赖的东西访问的时候出现了异常，发送异常事件到短路器中去进行统计

如果短路器发现异常事件的占比达到了一定的比例，直接开启短路，circuit breaker

上述四种情况，都会去调用fallback降级机制

fallback，降级机制，你之前都是必须去调用外部的依赖接口，或者从mysql中去查询数据的，但是为了避免说可能外部依赖会有故障

比如，你可以再内存中维护一个ehcache，作为一个纯内存的基于LRU自动清理的缓存，数据也可以放入缓存内

如果说外部依赖有异常，fallback这里，直接尝试从ehcache中获取数据

比如说，本来你是从mysql，redis，或者其他任何地方去获取数据的，获取调用其他服务的接口的，结果人家故障了，人家挂了，fallback，可以返回一个默认值

两种最经典的降级机制：纯内存数据，默认值

run()抛出异常，超时，线程池或信号量满了，或短路了，都会调用fallback机制

给大家举个例子，比如说我们现在有个商品数据，brandId，品牌，一般来说，假设，正常的逻辑，拿到了一个商品数据以后，用brandId再调用一次请求，到其他的服务去获取品牌的最新名称

假如说，那个品牌服务挂掉了，那么我们可以尝试本地内存中，会保留一份时间比较过期的一份品牌数据，有些品牌没有，有些品牌的名称过期了，Nike++，Nike

调用品牌服务失败了，fallback降级就从本地内存中获取一份过期的数据，先凑合着用着

public class CommandHelloFailure extends HystrixCommand<String> {

private final String name;

public CommandHelloFailure(String name) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.name = name;

}

@Override

protected String run() {

throw new RuntimeException("this command always fails");

}

@Override

protected String getFallback() {

return "Hello Failure " + name + "!";

}

}

@Test

public void testSynchronous() {

assertEquals("Hello Failure World!", new CommandHelloFailure("World").execute());

}

HystrixObservableCommand，是实现resumeWithFallback方法

2、fallback.isolation.semaphore.maxConcurrentRequests

这个参数设置了HystrixCommand.getFallback()最大允许的并发请求数量，默认值是10，也是通过semaphore信号量的机制去限流

如果超出了这个最大值，那么直接被reject

HystrixCommandProperties.Setter()

.withFallbackIsolationSemaphoreMaxConcurrentRequests(int value)

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

短路器深入的工作原理

1、如果经过短路器的流量超过了一定的阈值，HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()

举个例子，可能看起来是这样子的，要求在10s内，经过短路器的流量必须达到20个；在10s内，经过短路器的流量才10个，那么根本不会去判断要不要短路

2、如果断路器统计到的异常调用的占比超过了一定的阈值，HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()

如果达到了上面的要求，比如说在10s内，经过短路器的流量（你，只要执行一个command，这个请求就一定会经过短路器），达到了30个；同时其中异常的访问数量，占到了一定的比例，比如说60%的请求都是异常（报错，timeout，reject），会开启短路

3、然后断路器从close状态转换到open状态

4、断路器打开的时候，所有经过该断路器的请求全部被短路，不调用后端服务，直接走fallback降级

5、经过了一段时间之后，HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()，会half-open，让一条请求经过短路器，看能不能正常调用。如果调用成功了，那么就自动恢复，转到close状态

短路器，会自动恢复的，half-open，半开状态

6、circuit breaker短路器的配置

（1）circuitBreaker.enabled

控制短路器是否允许工作，包括跟踪依赖服务调用的健康状况，以及对异常情况过多时是否允许触发短路，默认是true

HystrixCommandProperties.Setter()

.withCircuitBreakerEnabled(boolean value)

（2）circuitBreaker.requestVolumeThreshold

设置一个rolling window，滑动窗口中，最少要有多少个请求时，才触发开启短路

举例来说，如果设置为20（默认值），那么在一个10秒的滑动窗口内，如果只有19个请求，即使这19个请求都是异常的，也是不会触发开启短路器的

HystrixCommandProperties.Setter()

.withCircuitBreakerRequestVolumeThreshold(int value)

（3）circuitBreaker.sleepWindowInMilliseconds

设置在短路之后，需要在多长时间内直接reject请求，然后在这段时间之后，再重新导holf-open状态，尝试允许请求通过以及自动恢复，默认值是5000毫秒

HystrixCommandProperties.Setter()

.withCircuitBreakerSleepWindowInMilliseconds(int value)

（4）circuitBreaker.errorThresholdPercentage

设置异常请求量的百分比，当异常请求达到这个百分比时，就触发打开短路器，默认是50，也就是50%

HystrixCommandProperties.Setter()

.withCircuitBreakerErrorThresholdPercentage(int value)

（5）circuitBreaker.forceOpen

如果设置为true的话，直接强迫打开短路器，相当于是手动短路了，手动降级，默认false

HystrixCommandProperties.Setter()

.withCircuitBreakerForceOpen(boolean value)

（6）circuitBreaker.forceClosed

如果设置为ture的话，直接强迫关闭短路器，相当于是手动停止短路了，手动升级，默认false

HystrixCommandProperties.Setter()

.withCircuitBreakerForceClosed(boolean value)

7、实战演练

配置一个断路器，流量要求是20，异常比例是50%，短路时间是5s

在command内加入一个判断，如果是productId=-1，那么就直接报错，触发异常执行

写一个client测试程序，写入50个请求，前20个是正常的，但是后30个是productId=-1，然后继续请求，会发现

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、command的创建和执行：资源隔离

2、request cache：请求缓存

3、fallback：优雅降级

4、circuit breaker：短路器，快速熔断（一旦后端服务故障，立刻熔断，阻止对其的访问）

把一个分布式系统中的某一个服务，打造成一个高可用的服务

资源隔离，优雅降级，熔断

5、判断，线程池或者信号量的容量是否已满，reject，限流

限流，限制对后端的服务的访问量，比如说你对mysql，redis，zookeeper，各种后端的中间件的资源，访问，其实为了避免过大的流浪打死后端的服务，线程池，信号量，限流

限制服务对后端的资源的访问

1、线程池隔离技术的设计原则

Hystrix采取了bulkhead舱壁隔离技术，来将外部依赖进行资源隔离，进而避免任何外部依赖的故障导致本服务崩溃

线程池隔离，学术名称：bulkhead，舱壁隔离

外部依赖的调用在单独的线程中执行，这样就能跟调用线程隔离开来，避免外部依赖调用timeout耗时过长，导致调用线程被卡死

Hystrix对每个外部依赖用一个单独的线程池，这样的话，如果对那个外部依赖调用延迟很严重，最多就是耗尽那个依赖自己的线程池而已，不会影响其他的依赖调用

Hystrix选择用线程池机制来进行资源隔离，要面对的场景如下：

（1）每个服务都会调用几十个后端依赖服务，那些后端依赖服务通常是由很多不同的团队开发的

（2）每个后端依赖服务都会提供它自己的client调用库，比如说用thrift的话，就会提供对应的thrift依赖

（3）client调用库随时会变更

（4）client调用库随时可能会增加新的网络请求的逻辑

（5）client调用库可能会包含诸如自动重试，数据解析，内存中缓存等逻辑

（6）client调用库一般都对调用者来说是个黑盒，包括实现细节，网络访问，默认配置，等等

（7）在真实的生产环境中，经常会出现调用者，突然间惊讶的发现，client调用库发生了某些变化

（8）即使client调用库没有改变，依赖服务本身可能有会发生逻辑上的变化

（9）有些依赖的client调用库可能还会拉取其他的依赖库，而且可能那些依赖库配置的不正确

（10）大多数网络请求都是同步调用的

（11）调用失败和延迟，也有可能会发生在client调用库本身的代码中，不一定就是发生在网络请求中

简单来说，就是你必须默认client调用库就很不靠谱，而且随时可能各种变化，所以就要用强制隔离的方式来确保任何服务的故障不能影响当前服务

我不知道在学习这个课程的学员里，有多少人，真正参与过一些复杂的分布式系统的开发，不是说一个team，你们五六个人，七八个人，去做的

在一些大公司里，做一些复杂的项目的话，广告计费系统，特别复杂，可能涉及多个团队，总共三四十个人，五六十个人，一起去开发一个系统，每个团队负责一块儿

每个团队里的每个人，负责一个服务，或者几个服务，比较常见的大公司的复杂分布式系统项目的分工合作的一个流程

线程池机制的优点如下：

（1）任何一个依赖服务都可以被隔离在自己的线程池内，即使自己的线程池资源填满了，也不会影响任何其他的服务调用

（2）服务可以随时引入一个新的依赖服务，因为即使这个新的依赖服务有问题，也不会影响其他任何服务的调用

（3）当一个故障的依赖服务重新变好的时候，可以通过清理掉线程池，瞬间恢复该服务的调用，而如果是tomcat线程池被占满，再恢复就很麻烦

（4）如果一个client调用库配置有问题，线程池的健康状况随时会报告，比如成功/失败/拒绝/超时的次数统计，然后可以近实时热修改依赖服务的调用配置，而不用停机

（5）如果一个服务本身发生了修改，需要重新调整配置，此时线程池的健康状况也可以随时发现，比如成功/失败/拒绝/超时的次数统计，然后可以近实时热修改依赖服务的调用配置，而不用停机

（6）基于线程池的异步本质，可以在同步的调用之上，构建一层异步调用层

简单来说，最大的好处，就是资源隔离，确保说，任何一个依赖服务故障，不会拖垮当前的这个服务

线程池机制的缺点：

（1）线程池机制最大的缺点就是增加了cpu的开销

除了tomcat本身的调用线程之外，还有hystrix自己管理的线程池

（2）每个command的执行都依托一个独立的线程，会进行排队，调度，还有上下文切换

（3）Hystrix官方自己做了一个多线程异步带来的额外开销，通过对比多线程异步调用+同步调用得出，Netflix API每天通过hystrix执行10亿次调用，每个服务实例有40个以上的线程池，每个线程池有10个左右的线程

（4）最后发现说，用hystrix的额外开销，就是给请求带来了3ms左右的延时，最多延时在10ms以内，相比于可用性和稳定性的提升，这是可以接受的

我们可以用hystrix semaphore技术来实现对某个依赖服务的并发访问量的限制，而不是通过线程池/队列的大小来限制流量

sempahore技术可以用来限流和削峰，但是不能用来对调研延迟的服务进行timeout和隔离

execution.isolation.strategy，设置为SEMAPHORE，那么hystrix就会用semaphore机制来替代线程池机制，来对依赖服务的访问进行限流

如果通过semaphore调用的时候，底层的网络调用延迟很严重，那么是无法timeout的，只能一直block住

一旦请求数量超过了semephore限定的数量之后，就会立即开启限流

2、接口限流实验

假设，一个线程池，大小是15个，队列大小是10个，timeout时长设置的长一些，5s

模拟发送请求，然后写死代码，在command内部做一个sleep，比如每次sleep 1s，10个请求发送过去以后，直接被hang死，线程池占满

再发送请求，就会堵塞在缓冲队列，queue，10个，20个，10个，后10个应该就直接reject，fallback逻辑

15 + 10 = 25个请求，15在执行，10个缓冲在队列里了，剩下的流量全部被reject，限流，降级

withCoreSize：设置你的线程池的大小

withMaxQueueSize：设置的是你的等待队列，缓冲队列的大小

withQueueSizeRejectionThreshold：如果withMaxQueueSize<withQueueSizeRejectionThreshold，那么取的是withMaxQueueSize，反之，取得是withQueueSizeRejectionThreshold

线程池本身的大小，如果你不设置另外两个queue相关的参数，等待队列是关闭的

queue大小，等待队列的大小，timeout时长

先进去线程池的是10个请求，然后有8个请求进入等待队列，线程池里有空闲，等待队列中的请求如果还没有timeout，那么就进去线程池去执行

10 + 8 = 18个请求之外，7个请求，直接会被reject掉，限流，fallback

withExecutionTimeoutInMilliseconds(20000)：timeout也设置大一些，否则如果请求放等待队列中时间太长了，直接就会timeout，等不到去线程池里执行了

withFallbackIsolationSemaphoreMaxConcurrentRequests(30)：fallback，sempahore限流，30个，避免太多的请求同时调用fallback被拒绝访问

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

一般来说，在调用依赖服务的接口的时候，比较常见的一个问题，就是超时

超时是在一个复杂的分布式系统中，导致不稳定，或者系统抖动，或者出现说大量超时，线程资源hang死，吞吐量大幅度下降，甚至服务崩溃

超时最大的一个问题

你去调用各种各样的依赖服务，特别是在大公司，你甚至都不认识开发一个服务的人，你都不知道那个人的水平怎么样，不了解

比尔盖茨说过一句话，在互联网的另外一头，你都不知道甚至坐着一条狗

分布式系统，大公司，多个团队，大型协作，服务是谁的，不了解，很可能说那个哥儿们，实习生都有可能

在一个复杂的系统里，可能你的依赖接口的性能很不稳定，有时候2ms，200ms，2s

如果你不对各种依赖接口的调用，做超时的控制，来给你的服务提供安全保护措施，那么很可能你的服务就被各种垃圾的依赖服务的性能给拖死了

大量的接口调用很慢，大量线程就卡死了，资源隔离，线程池的线程卡死了，超时的控制

（1）execution.isolation.thread.timeoutInMilliseconds

手动设置timeout时长，一个command运行超出这个时间，就被认为是timeout，然后将hystrix command标识为timeout，同时执行fallback降级逻辑

默认是1000，也就是1000毫秒

HystrixCommandProperties.Setter()

.withExecutionTimeoutInMilliseconds(int value)

（2）execution.timeout.enabled

控制是否要打开timeout机制，默认是true

HystrixCommandProperties.Setter()

.withExecutionTimeoutEnabled(boolean value)

让一个command执行timeout，然后看是否会调用fallback降级

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

已经学到的东西

hystrix的核心知识

1、hystrix内部工作原理：8大执行步骤和流程

2、资源隔离：你如果有很多个依赖服务，高可用性，先做资源隔离，任何一个依赖服务的故障不会导致你的服务的资源耗尽，不会崩溃

3、请求缓存：对于一个request context内的多个相同command，使用request cache，提升性能

4、熔断：基于短路器，采集各种异常事件，报错，超时，reject，短路，熔断，一定时间范围内就不允许访问了，直接降级，自动恢复的机制

5、降级：报错，超时，reject，熔断，降级，服务提供容错的机制

6、限流：在你的服务里面，通过线程池，或者信号量，限制对某个后端的服务或资源的访问量，避免从你的服务这里过去太多的流量，打死某个资源

7、超时：避免某个依赖服务性能过差，导致大量的线程hang住去调用那个服务，会导致你的服务本身性能也比较差

学会了这些东西以后，我们特意设置了大电商背景，商品详情页系统，缓存服务的业务场景，尽量的去结合一些仿真的业务，去学习hystrix的各项技术

这个东西做起来没那么容易，尽量做了，学习效果更好一些，兴趣也会更好一些

已经可以快速利用hystrix给自己开发的服务增加各种高可用的保障措施了，避免你的系统因为各种各样的异常情况导致崩溃，不可用

hystrix的高阶知识

1、request collapser，请求合并技术

2、fail-fast和fail-slient，高阶容错模式

3、static fallback和stubbed fallback，高阶降级模式

4、嵌套command实现的发送网络请求的降级模式

5、基于facade command的多级降级模式

6、request cache的手动清理

7、生产环境中的线程池大小以及timeout配置优化经验

8、线程池的自动化动态扩容与缩容技术

9、hystrix的metric高阶配置

10、基于hystrix dashboard的可视化分布式系统监控

11、生产环境中的hystrix工程运维经验

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、配置优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案

1、创建command

2、执行command

3、request cache

4、短路器，如果打开了，fallback降级机制

1、fallback降级机制

hystrix调用各种接口，或者访问外部依赖，mysql，redis，zookeeper，kafka，等等，如果出现了任何异常的情况

比如说报错了，访问mysql报错，redis报错，zookeeper报错，kafka报错，error

对每个外部依赖，无论是服务接口，中间件，资源隔离，对外部依赖只能用一定量的资源去访问，线程池/信号量，如果资源池已满，reject

访问外部依赖的时候，访问时间过长，可能就会导致超时，报一个TimeoutException异常，timeout

上述三种情况，都是我们说的异常情况，对外部依赖的东西访问的时候出现了异常，发送异常事件到短路器中去进行统计

如果短路器发现异常事件的占比达到了一定的比例，直接开启短路，circuit breaker

上述四种情况，都会去调用fallback降级机制

fallback，降级机制，你之前都是必须去调用外部的依赖接口，或者从mysql中去查询数据的，但是为了避免说可能外部依赖会有故障

比如，你可以再内存中维护一个ehcache，作为一个纯内存的基于LRU自动清理的缓存，数据也可以放入缓存内

如果说外部依赖有异常，fallback这里，直接尝试从ehcache中获取数据

比如说，本来你是从mysql，redis，或者其他任何地方去获取数据的，获取调用其他服务的接口的，结果人家故障了，人家挂了，fallback，可以返回一个默认值

两种最经典的降级机制：纯内存数据，默认值

run()抛出异常，超时，线程池或信号量满了，或短路了，都会调用fallback机制

给大家举个例子，比如说我们现在有个商品数据，brandId，品牌，一般来说，假设，正常的逻辑，拿到了一个商品数据以后，用brandId再调用一次请求，到其他的服务去获取品牌的最新名称

假如说，那个品牌服务挂掉了，那么我们可以尝试本地内存中，会保留一份时间比较过期的一份品牌数据，有些品牌没有，有些品牌的名称过期了，Nike++，Nike

调用品牌服务失败了，fallback降级就从本地内存中获取一份过期的数据，先凑合着用着

public class CommandHelloFailure extends HystrixCommand<String> {

private final String name;

public CommandHelloFailure(String name) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.name = name;

}

@Override

protected String run() {

throw new RuntimeException("this command always fails");

}

@Override

protected String getFallback() {

return "Hello Failure " + name + "!";

}

}

@Test

public void testSynchronous() {

assertEquals("Hello Failure World!", new CommandHelloFailure("World").execute());

}

HystrixObservableCommand，是实现resumeWithFallback方法

2、fallback.isolation.semaphore.maxConcurrentRequests

这个参数设置了HystrixCommand.getFallback()最大允许的并发请求数量，默认值是10，也是通过semaphore信号量的机制去限流

如果超出了这个最大值，那么直接被reject

HystrixCommandProperties.Setter()

.withFallbackIsolationSemaphoreMaxConcurrentRequests(int value)

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案