#### 01\_课程介绍以及高并发高可用复杂系统中的缓存架构有哪些东西？

第一讲，简单来跟大家说一下，4块

**1、现在常见的java工程师/架构师对缓存技术的了解和掌握程度**

我常年在一些国内最大的那些互联网公司里吧，负责招人，java这块，我们也会招，各种各样的人，我都见过

大型的互联网公司的人，传统行业的一些人，初级的人，高阶的架构师，高级工程师，技术经理，技术总监，带几十个人

缓存技术，复杂的场景的时候，很复杂的缓存架构

工作中都会用到一些缓存技术，redis/memcached基础使用，初步的集群知识

我面试过的人里，能掌握到很少的缓存架构的人，屈指可数，个位数，而且都是在大公司有过类似的大型复杂系统架构经验的人

**2、缓存架构/技术掌握的不够，对你的发展带来了哪些阻碍？**

工作中

如果你这块技术掌握不够，然后你的公司的项目遇到了一些相关的难题，高并发+高性能的场景，hold不住类似的这种高并发的系统

因为缓存架构做得不好，不到位，实际在公司的项目里，出了一些大case，导致系统崩溃，巨大的经济损失

职业发展中

redis，memcached，activemq，zookeeper，kafka，lucene，activiti，爬虫，或者等等，各种技术，写了都几十种技术

没有一样是精通的，redis就会简单的操作，memcached操作，activemq，zookeeper，爬虫，全都是简单的操作

架构设计思路，有没有一些考量的点，高并发的中场景，高可用的场景，说不出来

不可能做到更高级的一个职位了，因为很多公司的人也不傻，技术一看就平平庸庸，怎么给你一个很好的职位呢？职业发展怎么做上去呢？

亮点，技术亮点，高人一筹

java高工，java资深工，java架构，技术亮点，造诣

如果你的技术很牛，各种技术都有深度，架构面临过一些复杂的场景，别人搞不定的高并发高可用的系统架构，你都能搞定，职业发展就会做的很好

**3、课程的一个简单的介绍**

亿级流量电商网站的商品详情页系统，项目实战，业务背景，简化，贯穿起来，学习到亿级流量的电商网站，商品详情页的整体架构设计，学到的

复杂的缓存架构：才是我们最真实要讲解的东西，支撑高并发，高可用

缓存架构过程中：我们会讲解各种高并发场景下的各种难题，怎么去解决这些难题，缓存架构的过程，各种技术和解决方案，高可用性，解决缓存架构中面临的一些高可用的问题，包括怎么去解决，技术，解决方案

亿级流量电商网站的商品详情页系统，架构讲解，学到，作为项目背景贯穿，项目实战；缓存架构，支撑高并发，高可用的系统架构；缓存架构观察的过程中，高并发以及高可用相关的各种技术点和知识点，解决方案，串在一起，讲解了

这套课程，学到很多的全网独家的技术

大型电商网站的商品详情页系统的架构

复杂的缓存架构

如何用复杂的缓存架构去支撑高并发

利用将缓存架构做成高可用机会，也可以学到高可用系统架构构建的技术

**4、真正能支撑高并发以及高可用的复杂系统中的缓存架构有哪些东西？**

（1）如何让redis集群支撑几十万QPS高并发+99.99%高可用+TB级海量数据+企业级数据备份与恢复？：redis企业级集群架构

（2）如何支撑高性能以及高并发到极致？同时给缓存架构最后的安全保护层？：(nginx+lua)+redis+ehcache的三级缓存架构

（3）高并发场景下，如何解决数据库与缓存双写的时候数据不一致的情况？：企业级的完美的数据库+缓存双写一致性解决方案

（4）如何解决大value缓存的全量更新效率低下问题？：缓存维度化拆分解决方案

（5）如何将缓存命中率提升到极致？：双层nginx部署架构，以及lua脚本实现的一致性hash流量分发策略

（6）如何解决高并发场景下，缓存重建时的分布式并发重建的冲突问题？：基于zookeeper分布式锁的缓存并发重建解决方案

（7）如何解决高并发场景下，缓存冷启动MySQL瞬间被打死的问题？：基于storm实时统计热数据的分布式快速缓存预热解决方案

（8）如何解决热点缓存导致单机器负载瞬间超高？：基于storm的实时热点发现，以及毫秒级的实时热点缓存负载均衡降级

（9）如何解决分布式系统中的服务高可用问题？避免多层服务依赖因为少量故障导致系统崩溃？：基于hystrix的高可用缓存服务，资源隔离+限流+降级+熔断+超时控制

（10）如何应用分布式系统中的高可用服务的高阶技术？：基于hystrix的容错+多级降级+手动降级+生产环境参数优化经验+可视化运维与监控

（11）如何解决恐怖的缓存雪崩问题？避免给公司带来巨大的经济损失？：独家的事前+事中+事后三层次完美解决方案

（12）如何解决高并发场景下的缓存穿透问题？避免给MySQL带来过大的压力？：缓存穿透解决方案

（13）如何解决高并发场景下的缓存失效问题？避免给redis集群带来过大的压力？：缓存失效解决方案

**5、学会了这套课程，能给你带来些什么？工作中以及职业发展中？**

工作中：如果你遇到了类似的缓存架构的一些问题，你可以立刻将学到的东西结合你的项目业务融入到架构中去；系统架构重构，抵抗各种更加复杂的场景的架构

职业发展中：缓存，redis，复杂的缓存架构，解决的复杂场景，技术亮点，青睐，拿到更好的职位

java架构师，学完这套课程行不行？绝对不行

java架构师必备的一项架构技能，缓存架构，高并发（缓存架构，异步队列架构，复杂的分库分表），高可用架构（hystrix分布式系统服务的高可用），微服务的架构

让你积累了成长为java架构师过程中，必备的一项缓存架构的技能

5、笔记编辑器

sublime

#### 02\_基于大型电商网站中的商品详情页系统贯穿的授课思路介绍

我授课的一个思路，龙果另外两个课程，es的快速入门+es高手进阶

讲解一个技术，一个课程，就是纯讲技术，按照一个一个的技术点去讲解

决定拿一个从真实的系统中抽离出来的，简化过后的一个项目，去贯穿整个课程

项目去讲解，提供了连续而且仿真的一个业务场景

各种各样的业务场景，以及在业务场景中面临的难题和问题，去学习一个又一个的技术或者解决方案，或者架构设计思想

亿级流量电商网站的商品详情页系统

最最核心的架构就是缓存架构，商品详情页系统整体有自己整体的架构

一步一步的去实现商品详情页系统中的一些核心的部分，涉及到最最主要的就是缓存架构，高并发

缓存架构，一步一步讲解各种各样支撑高并发场景的缓存技术，解决方案，架构设计

如何将缓存架构本身做成高可用的架构，缓存架构本身面临的可用性的问题

基于hystrix去讲解，缓存架构本身做成高可用的，高可用架构的设计以及相关的技术

商品详情页系统架构 -> 缓存架构 -> 高并发技术+解决方案+架构 -> 高可用技术+解决方案+架构

#### 03\_小型电商网站的商品详情页的页面静态化架构以及其缺陷

商品详情页的系统架构 -> 缓存架构 -> 高并发 -> 高可用

电商网站里，大概可以说分成两种，第一种小型电商，简单的一种架构方案，页面静态化的方案；大型电商，复杂的一套架构，大电商，国内排名前几的电商，用得应该咱们这里讲解的这套大型的详情页架构

页面静态化，全量的页面静态化

<html>

<title></title>

<body>

商品名称：#{productName}

商品价格：#{productPrice}

商品描述：#{productDesc}

</body>

</html>

->

<html>

<title>

<style css>

<javascript>

</title>

<body>

商品名称：#{productName}

商品价格：#{productPrice}

商品描述：#{productDesc}

</body>

</html>

product1.html

假设是放在一个数据库里的

product表

product\_name product\_price product\_desc

iphon7 plus 5599.50 这是最好的手机

->

iphon7 plus（玫瑰） 5299.50 这是最好的手机，大降价了

......1万行数据，1万个页面

模板的渲染

<html>

<title>

<style css>

<javascript>

</title>

<body>

商品名称：iphon7 plus（玫瑰）

商品价格：5299.50

商品描述：这是最好的手机，大降价了

</body>

</html>

iphone7\_plus.html

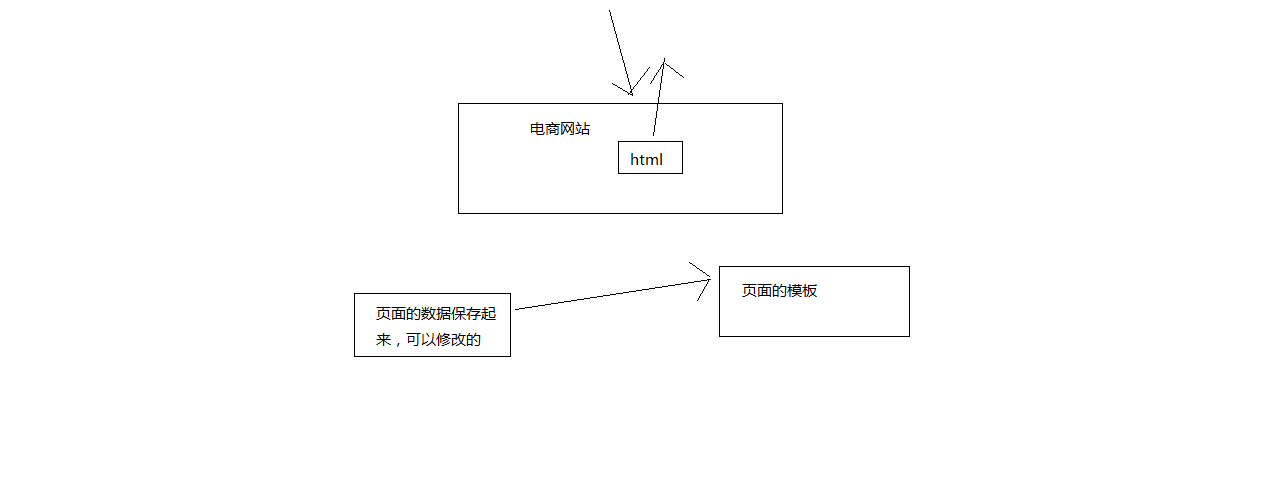
如果模板改变了，那么这个模板对应的所有数据，1万个数据，全部重新渲染一遍，填充到模板中，生成最终的静态化html页面

对于小网站，页面很少，很实用，非常简单，模板引擎，velocity，freemarker，页面数据管理的cms系统，内容管理系统

点击一个按钮，做成系统自动化，重新全量渲染

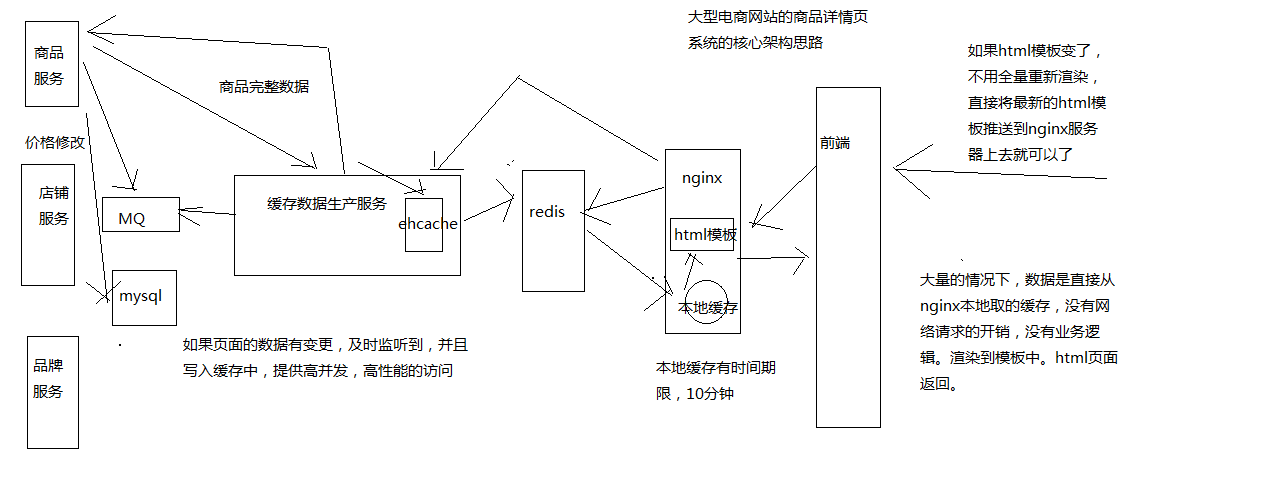
html --> 几百个页面，推送到nginx服务器上面，直接走html

页面太多，上亿，一个模板修改了，重新渲染一亿的商品，靠谱，几天



#### 04\_大型电商网站的异步多级缓存构建+nginx数据本地化动态渲染的架构

模板，可能所有的页面要重新渲染，很坑，大网站



#### 05\_能够支撑高并发+高可用+海量数据+备份恢复的redis的重要性

一块儿一块儿的去讲解，商品详情页的架构实现

缓存架构

第一块儿，要掌握的很好的，就是redis架构

高并发，高可用，海量数据，备份，随时可以恢复，缓存架构如果要支撑这些要点，首先呢，redis就得支撑

redis架构，每秒钟几十万的访问量QPS，99.99%的高可用性，TB级的海量的数据，备份和恢复，缓存架构就成功了一半了

最最简单的模式，无非就是存取redis，存数据，取数据

支撑你的缓存架构，最基础的就是redis架构

解决各种各样高并发场景下的缓存面临的难题，缓存架构中不断的引入各种解决方案和技术，解决高并发的问题

解决各种各样缓存架构本身面临的高可用的问题，缓存架构中引入各种解决方案和技术，解决高可用的问题

#### 06.从零开始在虚拟机中一步一步搭建一个4个节点的CentOS集群

课程大纲

1、在虚拟机中安装CentOS

2、在每个CentOS中都安装Java和Perl

3、在4个虚拟机中安装CentOS集群

4、配置4台CentOS为ssh免密码互相通信

从零开始，纯手工，一步一步搭建出一个4个节点的CentOS集群

为我们后面的课程做准备，后面会讲解大型的分布式的redis集群架构，一步一步纯手工搭建redis集群，集群部署，主从架构，分布式集群架构

我们后面的课程，会讲解一些实时计算技术的应用，包括storm，讲解一下storm的基础知识，对于java工程师来说，会用就可以了，用一些storm最基本的分布式实时计算的feature就ok了，搭建一个storm的集群

部署我们整套的系统，nginx，tomcat+java web应用，mysql

尽量以真实的网络拓扑的环境，去给大家演示一下整套系统的部署，不要所有东西，redis集群+storm集群+nginx+tomcat+mysql，全部放在一个节点上玩儿，也可以去试一试，但是作为课程来说，效果不是太理想

redis集群，独立的一套机器

storm集群，独立的一套机器

nginx，独立部署

tomcat + java web应用，独立部署

mysql，独立部署

十几个机器，去部署整套系统，我在自己的笔记本电脑上来讲课的，这么玩儿撑不住的

i5，12G

4台虚拟机，每台虚拟机是1G的内存，电脑基本还能撑住

电脑本身就6个G内存的话，学习这种大型的系统架构的课程，是有点吃力，给大家建议，几个G的内存条，也就几百块钱，给自己最好加个内存条，至少到8G以上

16G凑合

纯手工，从零开始

很多视频课程，里面讲师都是现成的虚拟机，自己都装好了，包括各种必要的软件

讲课的时候直接基于自己的虚拟机就开始讲解了

很多同学就会发现，想要做到跟讲师一样的环境都很难，自己可能照着样子装了个环境，但是发现，各种问题，各种报错，环境起不来

学习课程的过程很艰难

学视频课程，肯定是要跟着视频的所有的东西自己去做一做，练一练，结果你却因为环境问题，做不了，连不了，那就太惨了

从centos的镜像文件，到所有的需要使用的软件，全都给你，在自己电脑上，下载一个虚拟机管理软件，virtual box，就可以跟着玩儿了

如果你一步一步跟着视频做，搭建起整个环境，应该问题不大

环境问题，给大家弄成傻瓜式的

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

1、在虚拟机中安装CentOS

启动一个virtual box虚拟机管理软件（vmware，我早些年，发现不太稳定，主要是当时搭建一个hadoop大数据的集群，发现每次休眠以后再重启，集群就挂掉了）

virtual box，发现很稳定，集群从来不会随便乱挂，所以就一直用virtual box了

（1）使用课程提供的CentOS 6.5镜像即可，CentOS-6.5-i386-minimal.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为eshop-cache01，选择操作系统为Linux，选择版本为Red Hat，分配1024MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。

（4）安装虚拟机中的CentOS 6.5操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 6.5镜像文件），选择第一项开始安装-Skip-欢迎界面Next-选择默认语言-Baisc Storage Devices-Yes, discard any data-主机名:spark2upgrade01-选择时区-设置初始密码为hadoop-Replace Existing Linux System-Write changes to disk-CentOS 6.5自己开始安装。

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

vi /etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=eth0

TYPE=Ethernet

ONBOOT=yes

BOOTPROTO=dhcp

service network restart

ifconfig

BOOTPROTO=static

IPADDR=192.168.0.X

NETMASK=255.255.255.0

GATEWAY=192.168.0.1

service network restart

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

比如后面我们要安装很多其他的一些东西，perl，java，redis，storm，复制一些命令直接去执行

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

service iptables stop

service ip6tables stop

chkconfig iptables off

chkconfig ip6tables off

vi /etc/selinux/config

SELINUX=disabled

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

iyum clean all

yum makecache

yum install wget

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

2、在每个CentOS中都安装Java和Perl

WinSCP，就是在windows宿主机和linux虚拟机之间互相传递文件的一个工具

（1）安装JDK

1、将jdk-7u60-linux-i586.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-7u65-linux-i586.rpm

3、配置jdk相关的环境变量

vi .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

（2）安装Perl

很多讲师，拿着自己之前花了很多时间调试好的虚拟机环境，去讲课，这个很不负责任

yum install -y gcc

wget http://www.cpan.org/src/5.0/perl-5.16.1.tar.gz

tar -xzf perl-5.16.1.tar.gz

cd perl-5.16.1

./Configure -des -Dprefix=/usr/local/perl

make && make test && make install

perl -v

为什么要装perl？我们整个大型电商网站的详情页系统，复杂。java+nginx+lua，需要perl。

perl，是一个基础的编程语言的安装，tomcat，跑java web应用

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

3、在4个虚拟机中安装CentOS集群

（1）按照上述步骤，再安装三台一模一样环境的linux机器

（2）另外三台机器的hostname分别设置为eshop-cache02，eshop-cache03，eshop-cache04

（3）安装好之后，在每台机器的hosts文件里面，配置好所有的机器的ip地址到hostname的映射关系

比如说，在eshop-cache01的hosts里面

192.168.60.5 eshop-cache01

192.168.60.6 eshop-cache02

192.168.60.7 eshop-cache03

192.168.60.8 eshop-cache04

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

4、配置4台CentOS为ssh免密码互相通信

（1）首先在三台机器上配置对本机的ssh免密码登录

ssh-keygen -t rsa

生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下

cd /root/.ssh

cp id\_rsa.pub authorized\_keys

将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了

（2）接着配置三台机器互相之间的ssh免密码登录

使用ssh-copy-id -i hostname命令将本机的公钥拷贝到指定机器的authorized\_keys文件中

java，在公司里做项目，有几个人是自己去维护linux集群的啊？？？？？

几乎没有，很少很少，类似这一讲要做的事情，其实都是SRE，运维的同学，去做的

但是对于课程来说，我们只能自己一步一步做，才有环境去学习啊！！！

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

基于虚拟机的linux集群环境，都准备好了，手上有4台机器，后面玩儿各种redis、kafka、storm、tomcat、nginx，都有机器了

#### 07.单机版redis的安装以及redis生产环境启动方案

课程大纲

1、安装单机版redis

2、redis的生产环境启动方案

3、redis cli的使用

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

1、安装单机版redis

大家可以自己去官网下载，当然也可以用课程提供的压缩包

wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz

tar -xzvf tcl8.6.1-src.tar.gz

cd /usr/local/tcl8.6.1/unix/

./configure

make && make install

使用redis-3.2.8.tar.gz（截止2017年4月的最新稳定版）

tar -zxvf redis-3.2.8.tar.gz

cd redis-3.2.8

make && make test && make install

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

2、redis的生产环境启动方案

如果一般的学习课程，你就随便用redis-server启动一下redis，做一些实验，这样的话，没什么意义

要把redis作为一个系统的daemon进程去运行的，每次系统启动，redis进程一起启动

（1）redis utils目录下，有个redis\_init\_script脚本

（2）将redis\_init\_script脚本拷贝到linux的/etc/init.d目录中，将redis\_init\_script重命名为redis\_6379，6379是我们希望这个redis实例监听的端口号

（3）修改redis\_6379脚本的第6行的REDISPORT，设置为相同的端口号（默认就是6379）

（4）创建两个目录：/etc/redis（存放redis的配置文件），/var/redis/6379（存放redis的持久化文件）

（5）修改redis配置文件（默认在根目录下，redis.conf），拷贝到/etc/redis目录中，修改名称为6379.conf

（6）修改redis.conf中的部分配置为生产环境

daemonize yes 让redis以daemon进程运行

pidfile /var/run/redis\_6379.pid 设置redis的pid文件位置

port 6379 设置redis的监听端口号

dir /var/redis/6379 设置持久化文件的存储位置

（7）启动redis，执行cd /etc/init.d, chmod 777 redis\_6379，./redis\_6379 start

（8）确认redis进程是否启动，ps -ef | grep redis

（9）让redis跟随系统启动自动启动

在redis\_6379脚本中，最上面，加入两行注释

# chkconfig: 2345 90 10

# description: Redis is a persistent key-value database

chkconfig redis\_6379 on

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

3、redis cli的使用

redis-cli SHUTDOWN，连接本机的6379端口停止redis进程

redis-cli -h 192.168.60.5 -a redis-pass -p 6379 SHUTDOWN，制定要连接的ip和端口号

redis-cli PING，ping redis的端口，看是否正常

redis-cli，进入交互式命令行

SET k1 v1

GET k1

redis的技术，包括4块

redis各种数据结构和命令的使用，包括java api的使用

redis一些特殊的解决方案的使用，pub/sub消息系统，分布式锁，输入的自动完成，等等

redis日常的管理相关的命令

redis企业级的集群部署和架构

我们这套课程，实际上是针对企业级的大型缓存架构，用得项目，真实的大型电商网站的详情页系统（缓存）

我们首先讲解的第一块，其实就是企业级的大型缓存架构中的，redis集群架构（海量数据、高并发、高可用），最最流行，最最常用的分布式缓存系统

后面我们做商品详情页系统的业务开发的时候，当然也会去用redis的一些命令

redis基础知识：教程，书籍，视频

redis持久化、主从架构、复制原理、集群架构、数据分布式存储原理、哨兵原理、高可用架构

#### 08.redis持久化机对于生产环境中的灾难恢复的意义

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**

redis的持久化，RDB，AOF，区别，各自的特点是什么，适合什么场景

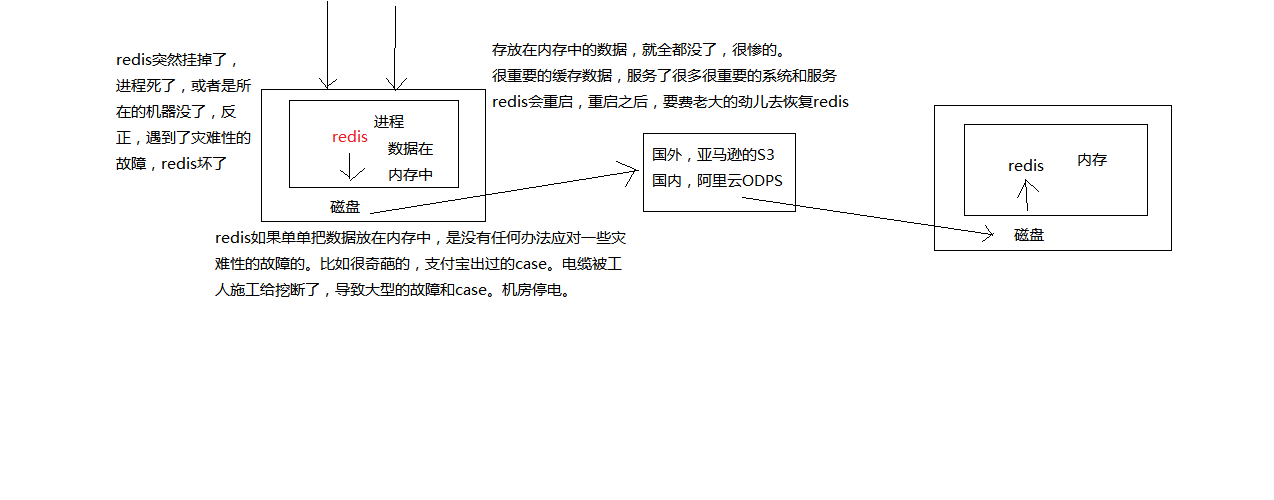
redis的企业级的持久化方案是什么，是用来跟哪些企业级的场景结合起来使用的？？？

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



#### 09.图解分析redis的RDB和AOF两种持久化机制的工作原理

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

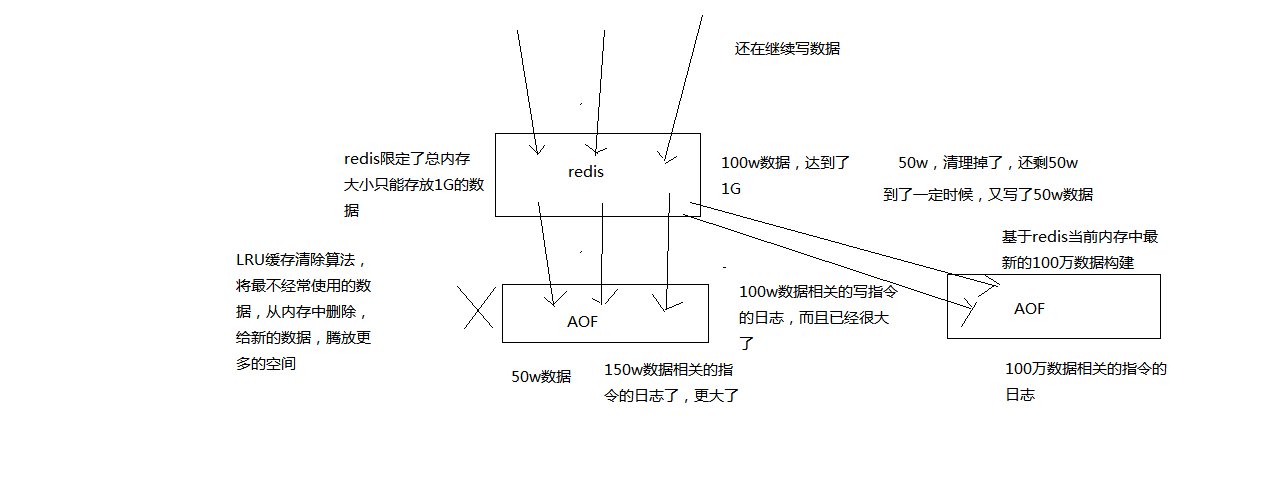
RDB持久化机制，对redis中的数据执行周期性的持久化

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。

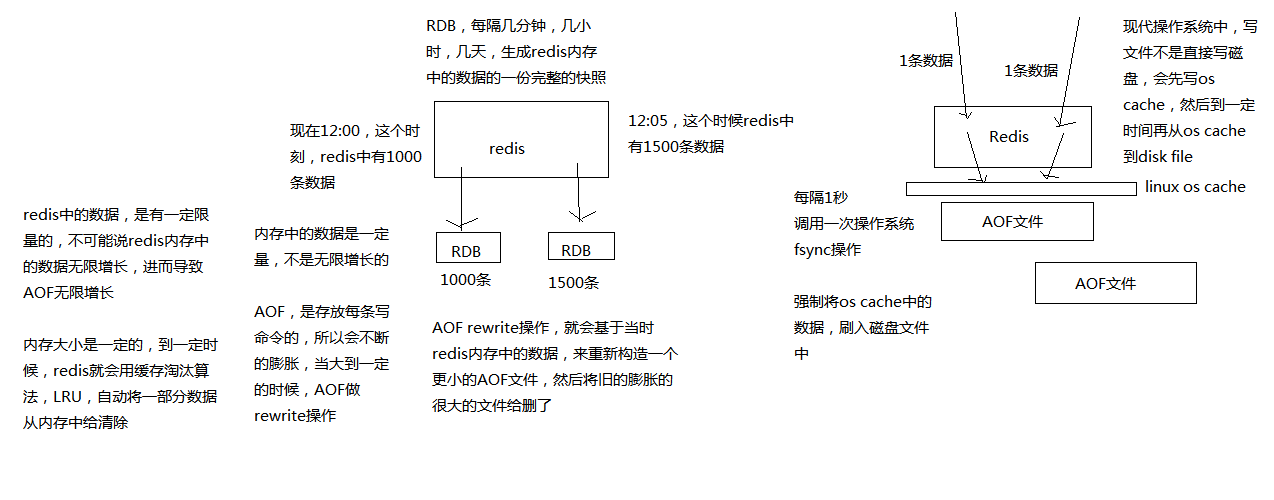
-------------------------------------------------------------------------------------

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



#### redis的RDB和AOF两种持久化机制的优劣势对比

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

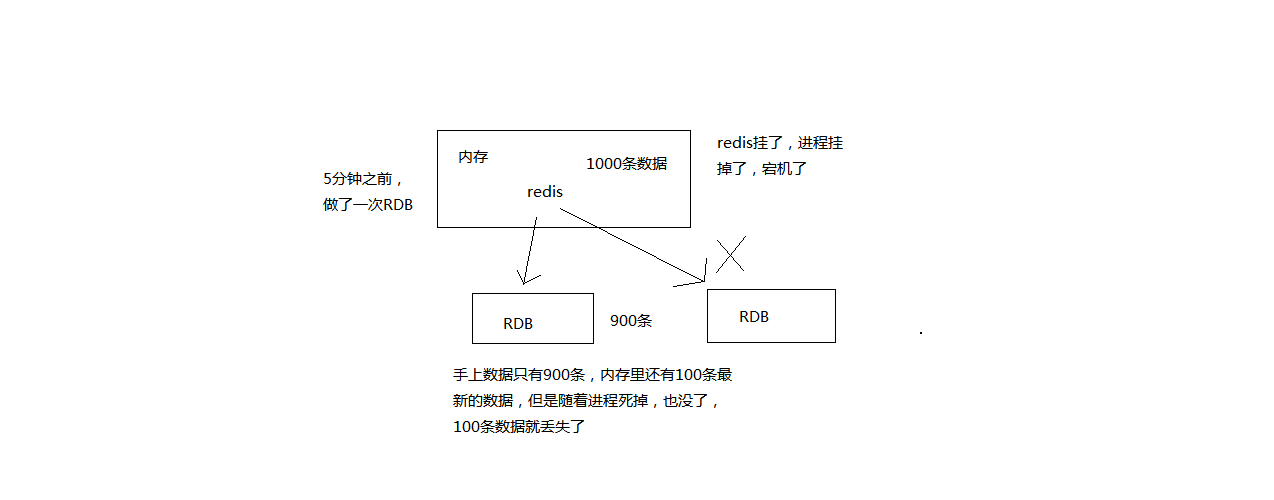
-------------------------------------------------------------------------------------

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



#### 11.redis的RDB持久化配置以及数据恢复实验

课程大纲

1、如何配置RDB持久化机制

2、RDB持久化机制的工作流程

3、基于RDB持久化机制的数据恢复实验

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

1、如何配置RDB持久化机制

redis.conf文件，也就是/etc/redis/6379.conf，去配置持久化

save 60 1000

每隔60s，如果有超过1000个key发生了变更，那么就生成一个新的dump.rdb文件，就是当前redis内存中完整的数据快照，这个操作也被称之为snapshotting，快照

也可以手动调用save或者bgsave命令，同步或异步执行rdb快照生成

save可以设置多个，就是多个snapshotting检查点，每到一个检查点，就会去check一下，是否有指定的key数量发生了变更，如果有，就生成一个新的dump.rdb文件

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

2、RDB持久化机制的工作流程

（1）redis根据配置自己尝试去生成rdb快照文件

（2）fork一个子进程出来

（3）子进程尝试将数据dump到临时的rdb快照文件中

（4）完成rdb快照文件的生成之后，就替换之前的旧的快照文件

dump.rdb，每次生成一个新的快照，都会覆盖之前的老快照

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

3、基于RDB持久化机制的数据恢复实验

（1）在redis中保存几条数据，立即停掉redis进程，然后重启redis，看看刚才插入的数据还在不在

数据还在，为什么？

带出来一个知识点，通过redis-cli SHUTDOWN这种方式去停掉redis，其实是一种安全退出的模式，redis在退出的时候会将内存中的数据立即生成一份完整的rdb快照

/var/redis/6379/dump.rdb

（2）在redis中再保存几条新的数据，用kill -9粗暴杀死redis进程，模拟redis故障异常退出，导致内存数据丢失的场景

这次就发现，redis进程异常被杀掉，数据没有进dump文件，几条最新的数据就丢失了

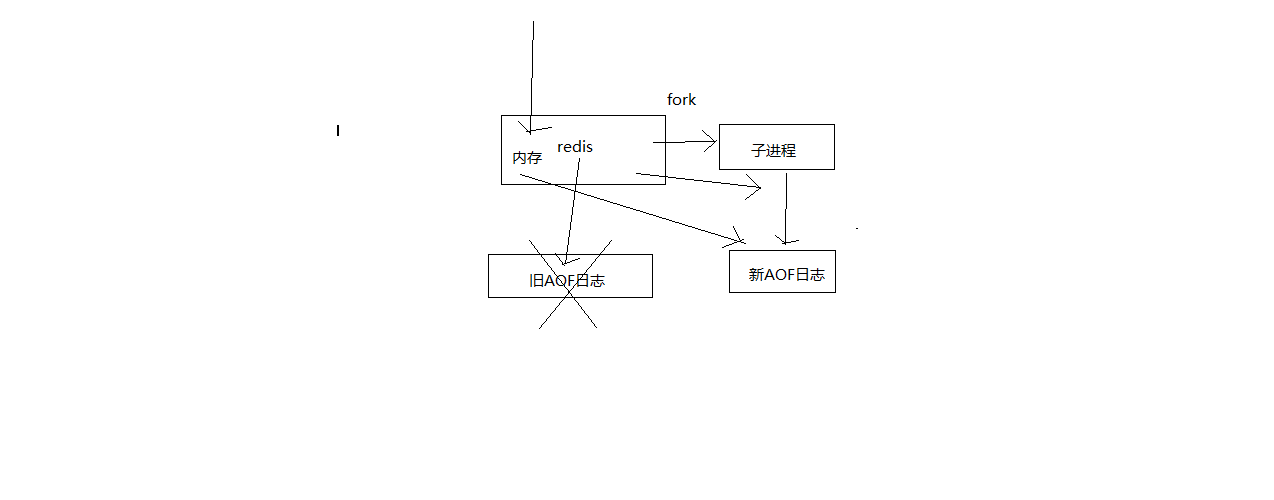
（2）手动设置一个save检查点，save 5 1

（3）写入几条数据，等待5秒钟，会发现自动进行了一次dump rdb快照，在dump.rdb中发现了数据

（4）异常停掉redis进程，再重新启动redis，看刚才插入的数据还在

rdb的手动配置检查点，以及rdb快照的生成，包括数据的丢失和恢复，全都演示过了

#### 12\_redis的AOF持久化深入讲解各种操作和相关实验



课程大纲

1、AOF持久化的配置

2、AOF持久化的数据恢复实验

3、AOF rewrite

4、AOF破损文件的修复

5、AOF和RDB同时工作

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

1、AOF持久化的配置

AOF持久化，默认是关闭的，默认是打开RDB持久化

**appendonly yes**，可以打开AOF持久化机制，在生产环境里面，一般来说AOF都是要打开的，除非你说随便丢个几分钟的数据也无所谓

打开AOF持久化机制之后，redis每次接收到一条写命令，就会写入日志文件中，当然是先写入os cache的，然后每隔一定时间再fsync一下

而且即使AOF和RDB都开启了，redis重启的时候，也是优先通过AOF进行数据恢复的，因为aof数据比较完整

可以配置AOF的fsync策略，有三种策略可以选择，一种是每次写入一条数据就执行一次fsync; 一种是每隔一秒执行一次fsync; 一种是不主动执行fsync

always: 每次写入一条数据，立即将这个数据对应的写日志fsync到磁盘上去，性能非常非常差，吞吐量很低; 确保说redis里的数据一条都不丢，那就只能这样了

mysql -> 内存策略，大量磁盘，QPS到多少，一两k。QPS，每秒钟的请求数量

redis -> 内存，磁盘持久化，QPS到多少，单机，一般来说，上万QPS没问题

everysec: 每秒将os cache中的数据fsync到磁盘，这个最常用的，生产环境一般都这么配置，性能很高，QPS还是可以上万的

no: 仅仅redis负责将数据写入os cache就撒手不管了，然后后面os自己会时不时有自己的策略将数据刷入磁盘，不可控了

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

2、AOF持久化的数据恢复实验

（1）先仅仅打开RDB，写入一些数据，然后kill -9杀掉redis进程，接着重启redis，发现数据没了，因为RDB快照还没生成

（2）打开AOF的开关，启用AOF持久化

（3）写入一些数据，观察AOF文件中的日志内容

其实你在appendonly.aof文件中，可以看到刚写的日志，它们其实就是先写入os cache的，然后1秒后才fsync到磁盘中，只有fsync到磁盘中了，才是安全的，要不然光是在os cache中，机器只要重启，就什么都没了

（4）kill -9杀掉redis进程，重新启动redis进程，发现数据被恢复回来了，就是从AOF文件中恢复回来的

redis进程启动的时候，直接就会从appendonly.aof中加载所有的日志，把内存中的数据恢复回来

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

3、AOF rewrite

redis中的数据其实有限的，很多数据可能会自动过期，可能会被用户删除，可能会被redis用缓存清除的算法清理掉

redis中的数据会不断淘汰掉旧的，就一部分常用的数据会被自动保留在redis内存中

所以可能很多之前的已经被清理掉的数据，对应的写日志还停留在AOF中，AOF日志文件就一个，会不断的膨胀，到很大很大

所以AOF会自动在后台每隔一定时间做rewrite操作，比如日志里已经存放了针对100w数据的写日志了; redis内存只剩下10万; 基于内存中当前的10万数据构建一套最新的日志，到AOF中; 覆盖之前的老日志; 确保AOF日志文件不会过大，保持跟redis内存数据量一致

redis 2.4之前，还需要手动，开发一些脚本，crontab，通过BGREWRITEAOF命令去执行AOF rewrite，但是redis 2.4之后，会自动进行rewrite操作

在redis.conf中，可以配置rewrite策略

auto-aof-rewrite-percentage 100

auto-aof-rewrite-min-size 64mb

比如说上一次AOF rewrite之后，是128mb

然后就会接着128mb继续写AOF的日志，如果发现增长的比例，超过了之前的100%，256mb，就可能会去触发一次rewrite

但是此时还要去跟min-size，64mb去比较，256mb > 64mb，才会去触发rewrite

（1）redis fork一个子进程

（2）子进程基于当前内存中的数据，构建日志，开始往一个新的临时的AOF文件中写入日志

（3）redis主进程，接收到client新的写操作之后，在内存中写入日志，同时新的日志也继续写入旧的AOF文件

（4）子进程写完新的日志文件之后，redis主进程将内存中的新日志再次追加到新的AOF文件中

（5）用新的日志文件替换掉旧的日志文件

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

4、AOF破损文件的修复

如果redis在append数据到AOF文件时，机器宕机了，可能会导致AOF文件破损

用redis-check-aof --fix命令来修复破损的AOF文件

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

5、AOF和RDB同时工作

（1）如果RDB在执行snapshotting操作，那么redis不会执行AOF rewrite; 如果redis再执行AOF rewrite，那么就不会执行RDB snapshotting

（2）如果RDB在执行snapshotting，此时用户执行BGREWRITEAOF命令，那么等RDB快照生成之后，才会去执行AOF rewrite

（3）同时有RDB snapshot文件和AOF日志文件，那么redis重启的时候，会优先使用AOF进行数据恢复，因为其中的日志更完整

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

6、最后一个小实验，让大家对redis的数据恢复有更加深刻的体会

（1）在有rdb的dump和aof的appendonly的同时，rdb里也有部分数据，aof里也有部分数据，这个时候其实会发现，rdb的数据不会恢复到内存中

（2）我们模拟让aof破损，然后fix，有一条数据会被fix删除

（3）再次用fix得aof文件去重启redis，发现数据只剩下一条了

数据恢复完全是依赖于底层的磁盘的持久化的，主要rdb和aof上都没有数据，那就没了

#### 13\_在项目中部署redis企业级数据备份方案以及各种踩坑的数据恢复容灾演练

到这里为止，其实还是停留在简单学习知识的程度，学会了redis的持久化的原理和操作，但是在企业中，持久化到底是怎么去用得呢？

企业级的数据备份和各种灾难下的数据恢复，是怎么做得呢？

1、企业级的持久化的配置策略

在企业中，RDB的生成策略，用默认的也差不多

save 60 10000：如果你希望尽可能确保说，RDB最多丢1分钟的数据，那么尽量就是每隔1分钟都生成一个快照，低峰期，数据量很少，也没必要

10000->生成RDB，1000->RDB，这个根据你自己的应用和业务的数据量，你自己去决定

**AOF一定要打开，fsync，everysec**

auto-aof-rewrite-percentage 100: 就是当前AOF大小膨胀到超过上次100%，上次的两倍

auto-aof-rewrite-min-size 64mb: 根据你的数据量来定，16mb，32mb

2、企业级的数据备份方案

**RDB非常适合做冷备，每次生成之后，就不会再有修改了**

数据备份方案

**（1）写crontab定时调度脚本去做数据备份**

**（2）每小时都copy一份rdb的备份，到一个目录中去，仅仅保留最近48小时的备份**

**（3）每天都保留一份当日的rdb的备份，到一个目录中去，仅仅保留最近1个月的备份**

**（4）每次copy备份的时候，都把太旧的备份给删了**

**（5）每天晚上将当前服务器上所有的数据备份，发送一份到远程的云服务上去**

**bin/sh^M: bad interpreter: No such file or directory解决 系统编辑器问题**

（1）使用vi工具

      vi test.sh

    （2）利用如下命令查看文件格式

     :set ff 或 :set fileformat

     可以看到如下信息

     fileformat=[dos](http://www.2cto.com/os/dos/" \o "dos) 或 fileformat=unix

     （3） 利用如下命令修改文件格式

     :set ff=unix 或 :set fileformat=unix

     :wq (存盘退出)

/usr/local/redis

每小时copy一次备份，删除48小时前的数据

crontab -e

0 \* \* \* \* sh /usr/local/redis/copy/redis\_rdb\_copy\_hourly.sh

redis\_rdb\_copy\_hourly.sh

#!/bin/sh

cur\_date=`date +%Y%m%d%k`

rm -rf /usr/local/redis/snapshotting/$cur\_date

mkdir /usr/local/redis/snapshotting/$cur\_date

cp /var/redis/6379/dump.rdb /usr/local/redis/snapshotting/$cur\_date

del\_date=`date -d -48hour +%Y%m%d%k`

rm -rf /usr/local/redis/snapshotting/$del\_date

每天copy一次备份

crontab -e

0 0 \* \* \* sh /usr/local/redis/copy/redis\_rdb\_copy\_daily.sh

redis\_rdb\_copy\_daily.sh

#!/bin/sh

cur\_date=`date +%Y%m%d`

rm -rf /usr/local/redis/snapshotting/$cur\_date

mkdir /usr/local/redis/snapshotting/$cur\_date

cp /var/redis/6379/dump.rdb /usr/local/redis/snapshotting/$cur\_date

del\_date=`date -d -1month +%Y%m%d`

rm -rf /usr/local/redis/snapshotting/$del\_date

snapshotting

每天一次将所有数据上传一次到远程的云服务器上去

lsof | grep redis\_rdb\_copy\_daily.sh 查看文件被哪个进程访问中

3、数据恢复方案

（1）如果是redis进程挂掉，那么重启redis进程即可，直接基于AOF日志文件恢复数据

不演示了，在AOF数据恢复那一块，演示了，fsync everysec，最多就丢一秒的数

（2）如果是redis进程所在机器挂掉，那么重启机器后，尝试重启redis进程，尝试直接基于AOF日志文件进行数据恢复

AOF没有破损，也是可以直接基于AOF恢复的

AOF append-only，顺序写入，如果AOF文件破损，那么用redis-check-aof fix

（3）如果redis当前最新的AOF和RDB文件出现了丢失/损坏，那么可以尝试基于该机器上当前的某个最新的RDB数据副本进行数据恢复

当前最新的AOF和RDB文件都出现了丢失/损坏到无法恢复，一般不是机器的故障，人为

大数据系统，hadoop，有人不小心就把hadoop中存储的大量的数据文件对应的目录，rm -rf一下，我朋友的一个小公司，运维不太靠谱，权限也弄的不太好

/var/redis/6379下的文件给删除了

找到RDB最新的一份备份，小时级的备份可以了，小时级的肯定是最新的，copy到redis里面去，就可以恢复到某一个小时的数据

容灾演练

appendonly.aof + dump.rdb，优先用appendonly.aof去恢复数据，但是我们发现redis自动生成的appendonly.aof是没有数据的

然后我们自己的dump.rdb是有数据的，但是明显没用我们的数据

redis启动的时候，自动重新基于内存的数据，生成了一份最新的rdb快照，直接用空的数据，覆盖掉了我们有数据的，拷贝过去的那份dump.rdb

**你停止redis之后，其实应该先删除appendonly.aof，然后将我们的dump.rdb拷贝过去，然后再重启redis**

很简单，就是虽然你删除了appendonly.aof，但是因为打开了aof持久化，redis就一定会优先基于aof去恢复，即使文件不在，那就创建一个新的空的aof文件

停止redis，暂时在配置中关闭aof，然后拷贝一份rdb过来，再重启redis，数据能不能恢复过来，可以恢复过来

脑子一热，再关掉redis，手动修改配置文件，打开aof，再重启redis，数据又没了，空的aof文件，所有数据又没了

在数据安全丢失的情况下，基于rdb冷备，如何完美的恢复数据，同时还保持aof和rdb的双开

停止redis，关闭aof，拷贝rdb备份，重启redis，确认数据恢复，直接在命令行热修改redis配置，打开aof，这个redis就会将内存中的数据对应的日志，写入aof文件中

此时aof和rdb两份数据文件的数据就同步了

redis config set热修改配置参数，可能配置文件中的实际的参数没有被持久化的修改，再次停止redis，手动修改配置文件，打开aof的命令，再次重启redis

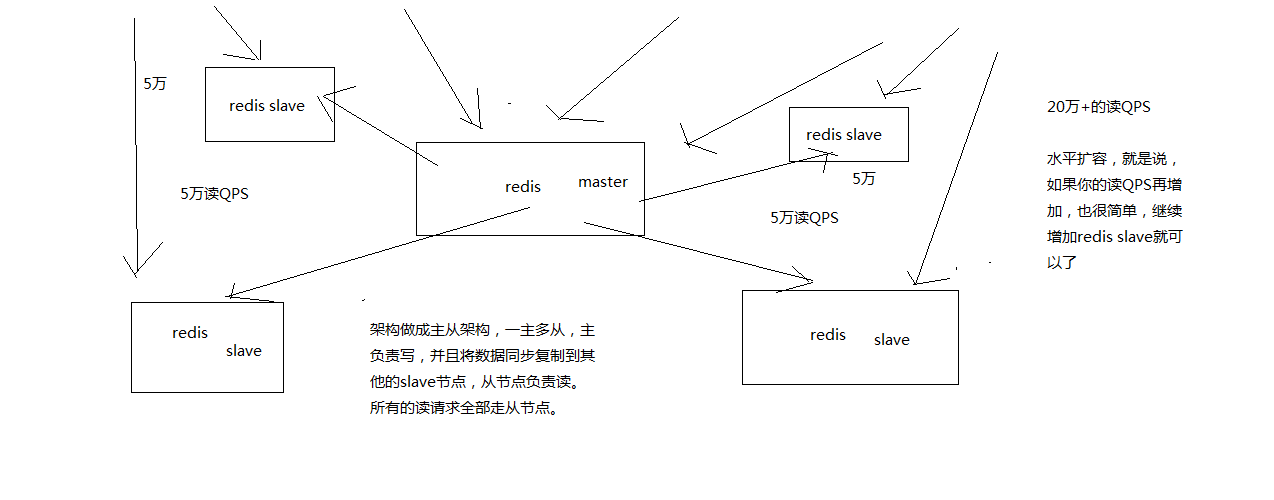
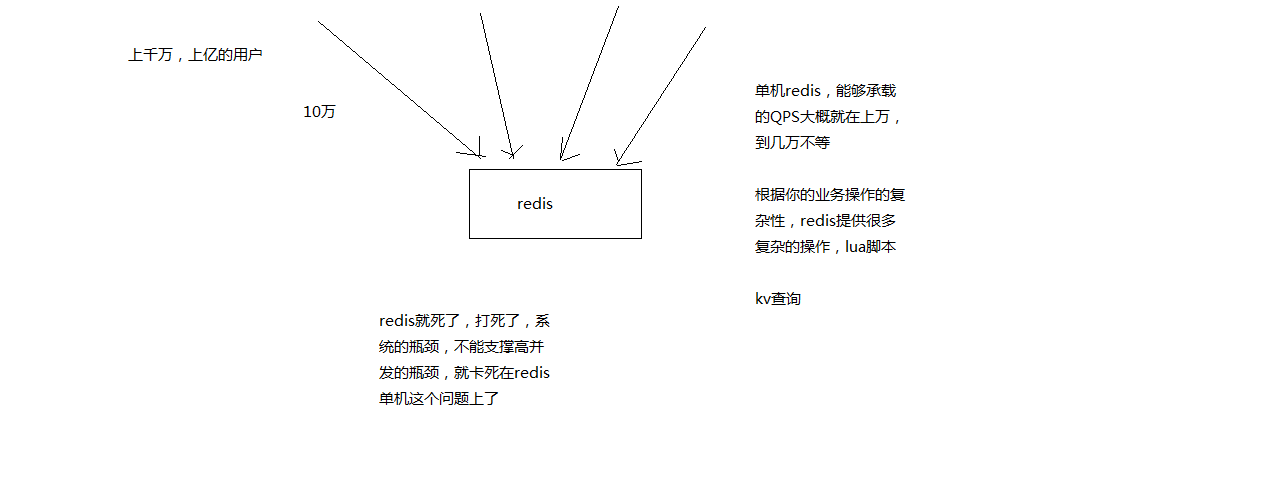
（4）如果当前机器上的所有RDB文件全部损坏，那么从远程的云服务上拉取最新的RDB快照回来恢复数据

（5）如果是发现有重大的数据错误，比如某个小时上线的程序一下子将数据全部污染了，数据全错了，那么可以选择某个更早的时间点，对数据进行恢复

举个例子，12点上线了代码，发现代码有bug，导致代码生成的所有的缓存数据，写入redis，全部错了

找到一份11点的rdb的冷备，然后按照上面的步骤，去恢复到11点的数据，不就可以了吗

#### 14\_redis如何通过读写分离来承载读请求QPS超过10万+？



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

#### 15\_redis replication以及master持久化对主从架构的安全意义

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

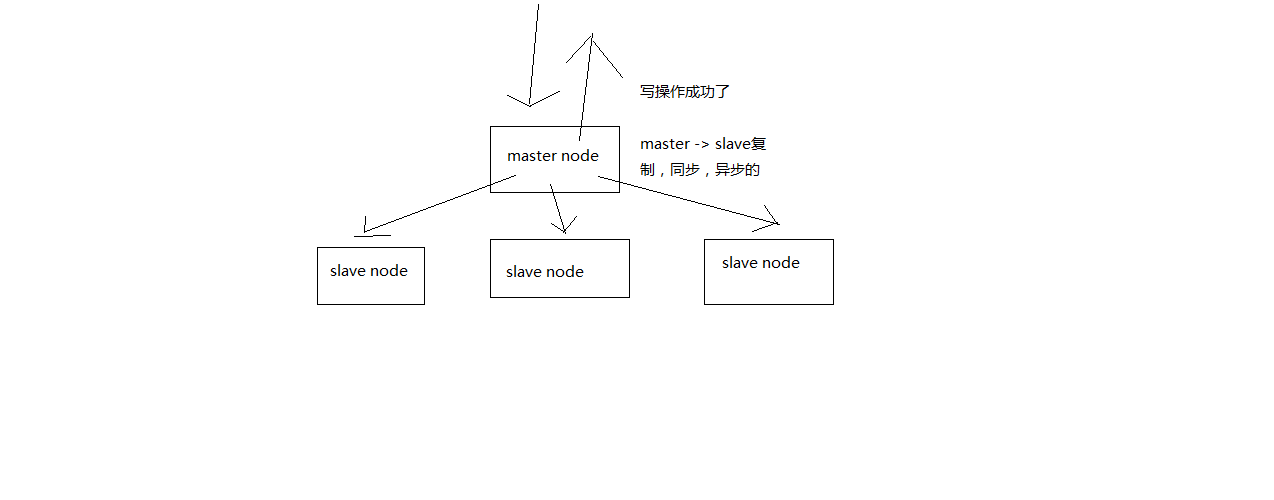
100%的数据丢失

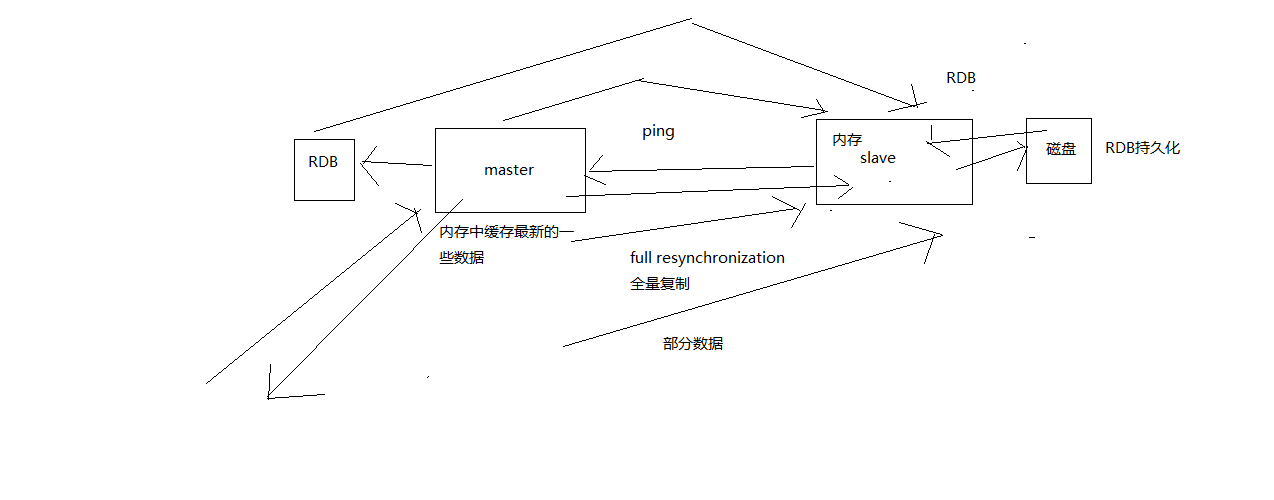
master节点，必须要使用持久化机制

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

#### 16\_redis主从复制原理、断点续传、无磁盘化复制、过期key处理





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

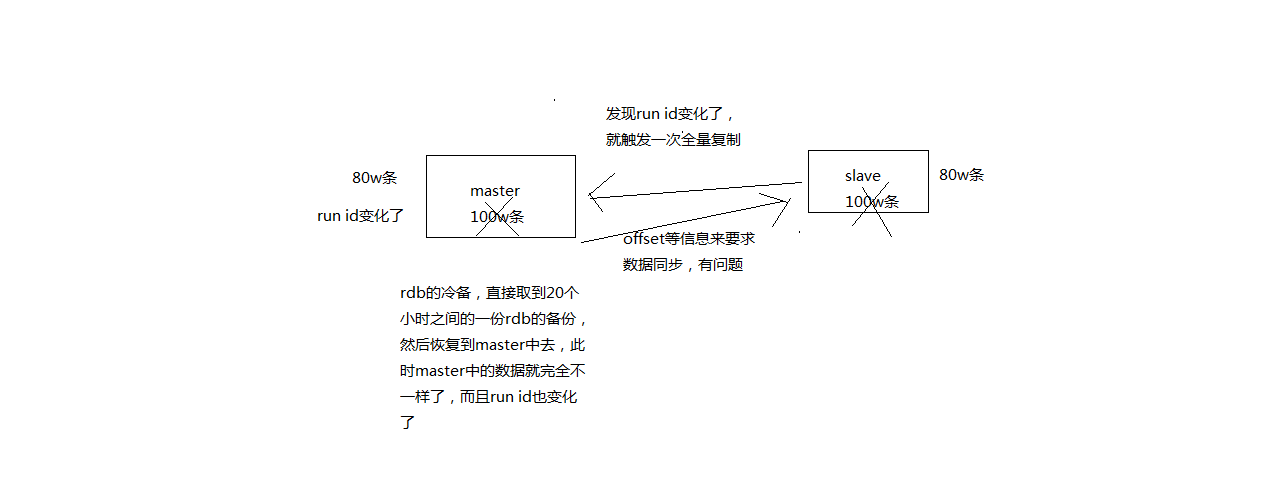
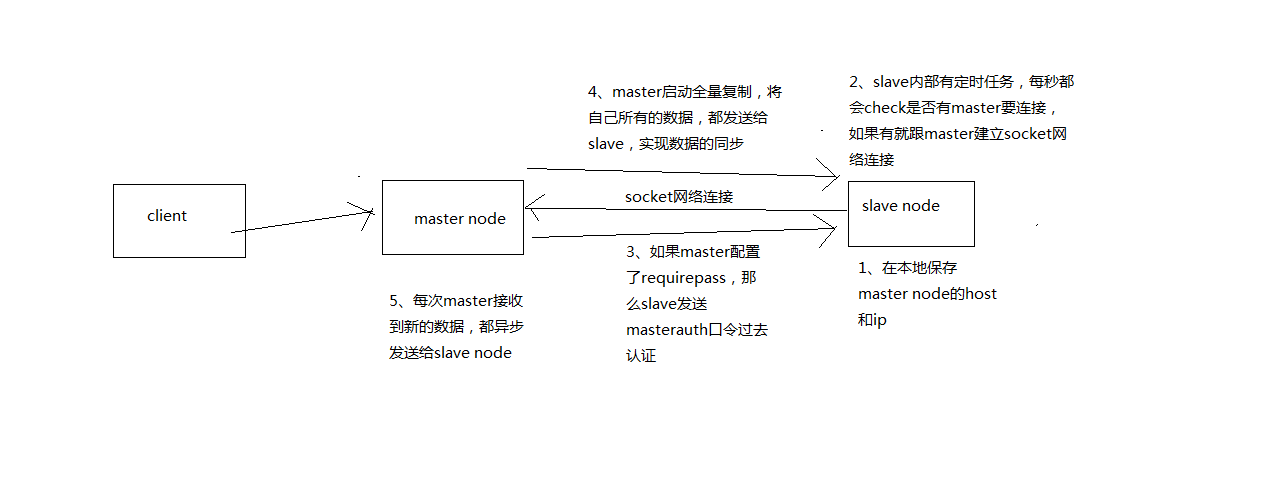
repl-diskless-sync

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。

#### 17\_redis replication的完整流运行程和原理的再次深入剖析



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

#### 18\_在项目中部署redis的读写分离架构（包含节点间认证口令）

之前几讲都是在铺垫各种redis replication的原理，和知识，主从，读写分离，画图

一主一从，往主节点去写，在从节点去读，可以读到，主从架构就搭建成功了

1、启用复制，部署slave node

wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz

tar -xzvf tcl8.6.1-src.tar.gz

cd /usr/local/tcl8.6.1/unix/

./configure

make && make install

使用redis-3.2.8.tar.gz（截止2017年4月的最新稳定版）

tar -zxvf redis-3.2.8.tar.gz

cd redis-3.2.8

make && make test && make install

（1）redis utils目录下，有个redis\_init\_script脚本

（2）将redis\_init\_script脚本拷贝到linux的/etc/init.d目录中，将redis\_init\_script重命名为redis\_6379，6379是我们希望这个redis实例监听的端口号

（3）修改redis\_6379脚本的第6行的REDISPORT，设置为相同的端口号（默认就是6379）

（4）创建两个目录：/etc/redis（存放redis的配置文件），/var/redis/6379（存放redis的持久化文件）

（5）修改redis配置文件（默认在根目录下，redis.conf），拷贝到/etc/redis目录中，修改名称为6379.conf

（6）修改redis.conf中的部分配置为生产环境

daemonize yes 让redis以daemon进程运行

pidfile /var/run/redis\_6379.pid 设置redis的pid文件位置

port 6379 设置redis的监听端口号

dir /var/redis/6379 设置持久化文件的存储位置

（7）让redis跟随系统启动自动启动

在redis\_6379脚本中，最上面，加入两行注释

# chkconfig: 2345 90 10

# description: Redis is a persistent key-value database

chkconfig redis\_6379 on

在slave node上配置：slaveof 192.168.1.1 6379，即可

也可以使用slaveof命令

2、强制读写分离

基于主从复制架构，实现读写分离

redis slave node只读，默认开启，slave-read-only

开启了只读的redis slave node，会拒绝所有的写操作，这样可以强制搭建成读写分离的架构

3、集群安全认证

Slave-node上的masterauth redis-pass

master上启用安全认证，requirepass

master连接口令，masterauth

4、读写分离架构的测试

先启动主节点，eshop-cache01上的redis实例

再启动从节点，eshop-cache02上的redis实例

刚才我调试了一下，redis slave node一直说没法连接到主节点的6379的端口

在搭建生产环境的集群的时候，不要忘记修改一个配置，bind

bind 127.0.0.1 -> 本地的开发调试的模式，就只能127.0.0.1本地才能访问到6379的端口

每个redis.conf中的bind 127.0.0.1 -> bind自己的ip地址

在每个节点上都: iptables -A INPUT -ptcp --dport 6379 -j ACCEPT

redis-cli -h ipaddr -a redis-pass

info replication

在主上写，在从上读

#### 19\_对项目的主从redis架构进行QPS压测以及水平扩容支撑更高QPS

你如果要对自己刚刚搭建好的redis做一个基准的压测，测一下你的redis的性能和QPS（query per second）

redis自己提供的redis-benchmark压测工具，是最快捷最方便的，当然啦，这个工具比较简单，用一些简单的操作和场景去压测

1、对redis读写分离架构进行压测，单实例写QPS+单实例读QPS

**redis-3.2.8/src**

**./redis-benchmark -h 192.168.60.5**

-c <clients> Number of parallel connections (default 50)

-n <requests> Total number of requests (default 100000)

-d <size> Data size of SET/GET value in bytes (default 2)

根据你自己的高峰期的访问量，在高峰期，瞬时最大用户量会达到10万+，-c 100000，-n 10000000，-d 50

各种基准测试，直接出来

1核1G，虚拟机

====== SET ======

100000 requests completed in 1.40 seconds

50 parallel clients

3 bytes payload

keep alive: 1

99.73% <= 1 milliseconds

100.00% <= 1 milliseconds

71174.38 requests per second

====== GET ======

100000 requests completed in 1.42 seconds

50 parallel clients

3 bytes payload

keep alive: 1

99.20% <= 1 milliseconds

100.00% <= 1 milliseconds

70372.98 requests per second

搭建一些集群，专门为某个项目，搭建的专用集群，4核4G内存，比较复杂的操作，数据比较大

redis提供的高并发，至少到上万，没问题

生产环境，大量的网络请求的调用，网络本身就有开销，你的redis的吞吐量就不一定那么高了

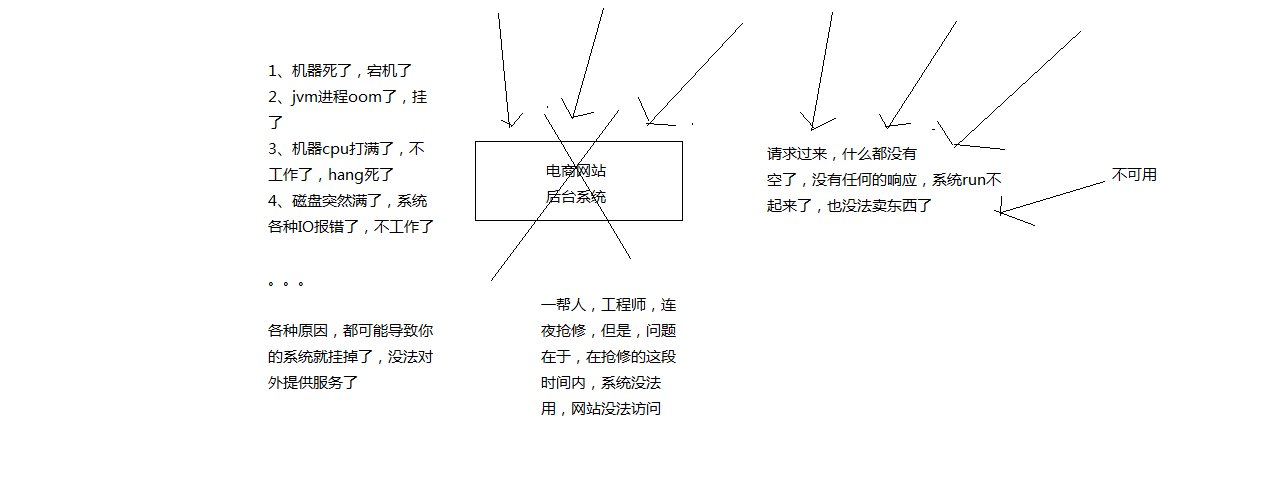
QPS的两个杀手：一个是复杂操作，lrange，挺多的; value很大，2 byte，我之前用redis做大规模的缓存

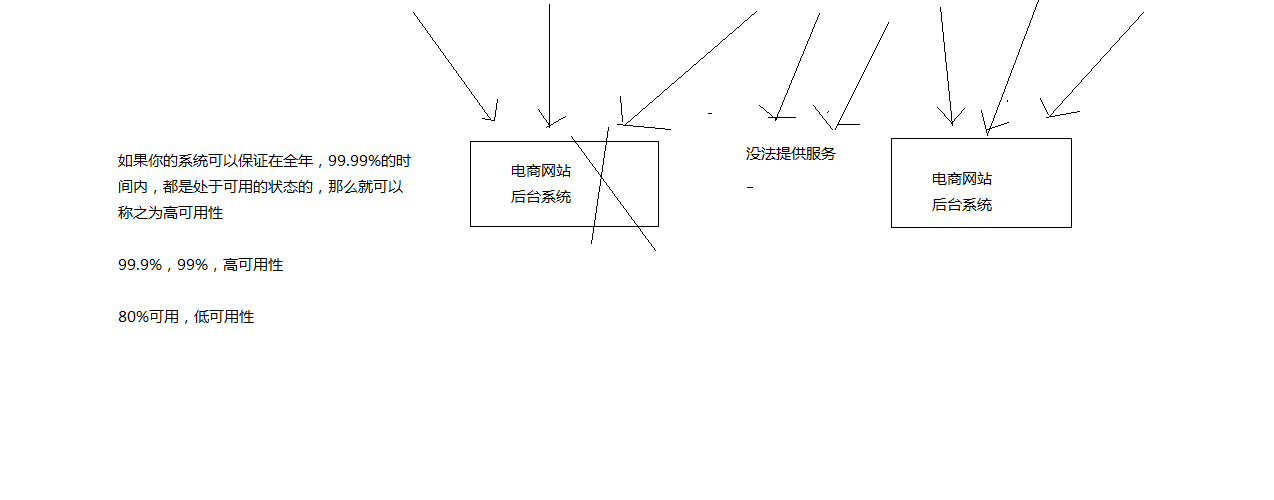
做商品详情页的cache，可能是需要把大串数据，拼接在一起，作为一个json串，大小可能都几k，几个byte

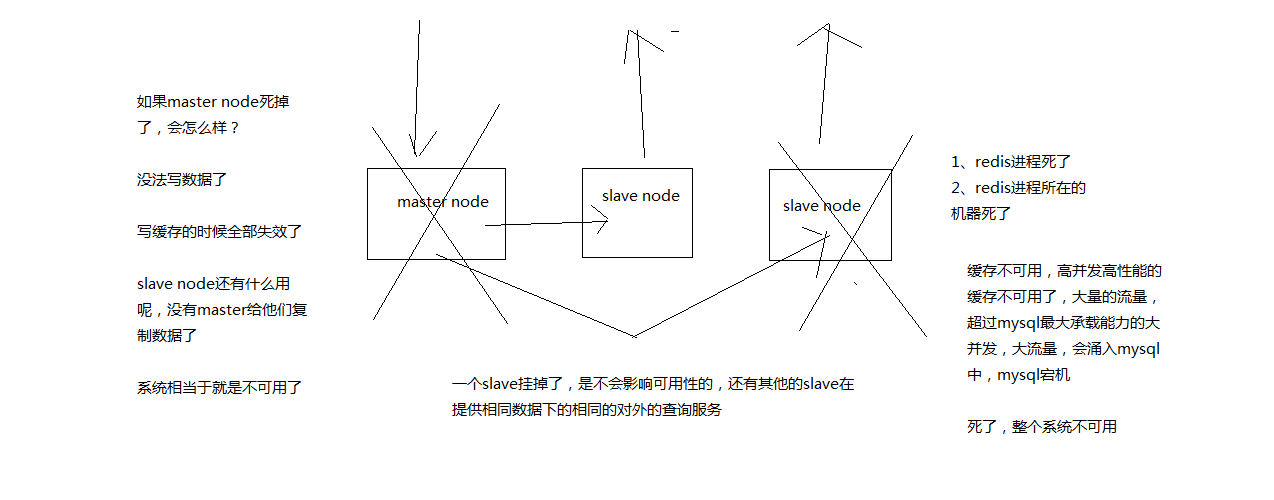
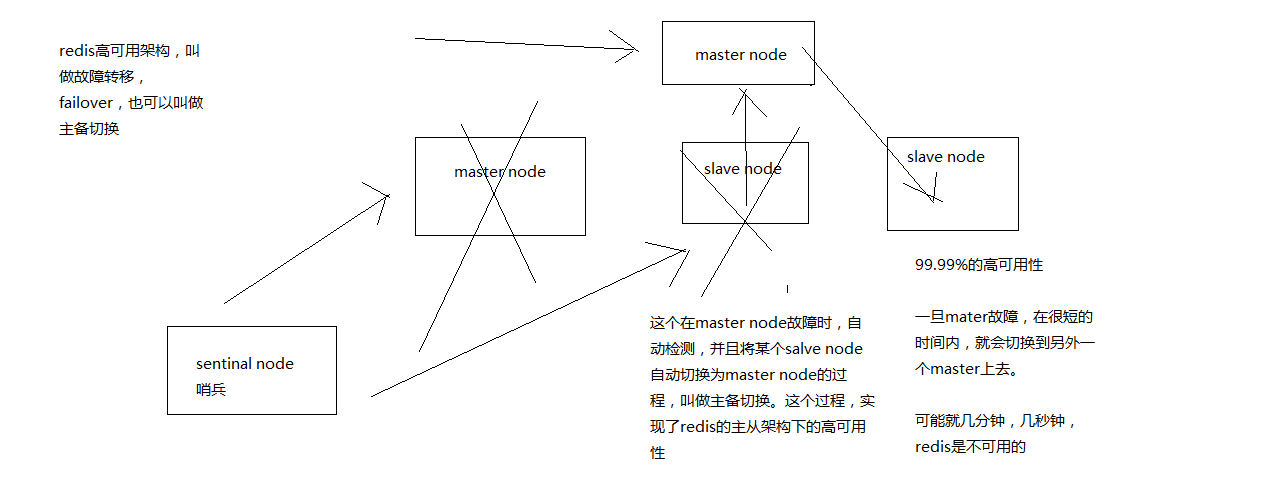
2、水平扩容redis读节点，提升度吞吐量

就按照上一节课讲解的，再在其他服务器上搭建redis从节点，单个从节点读请QPS在5万左右，两个redis从节点，所有的读请求打到两台机器上去，承载整个集群读QPS在10万+

#### 20\_redis主从架构下如何才能做到99.99%的高可用性？







1、什么是99.99%高可用？

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？

#### 21\_redis哨兵架构的相关基础知识的讲解

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 |

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

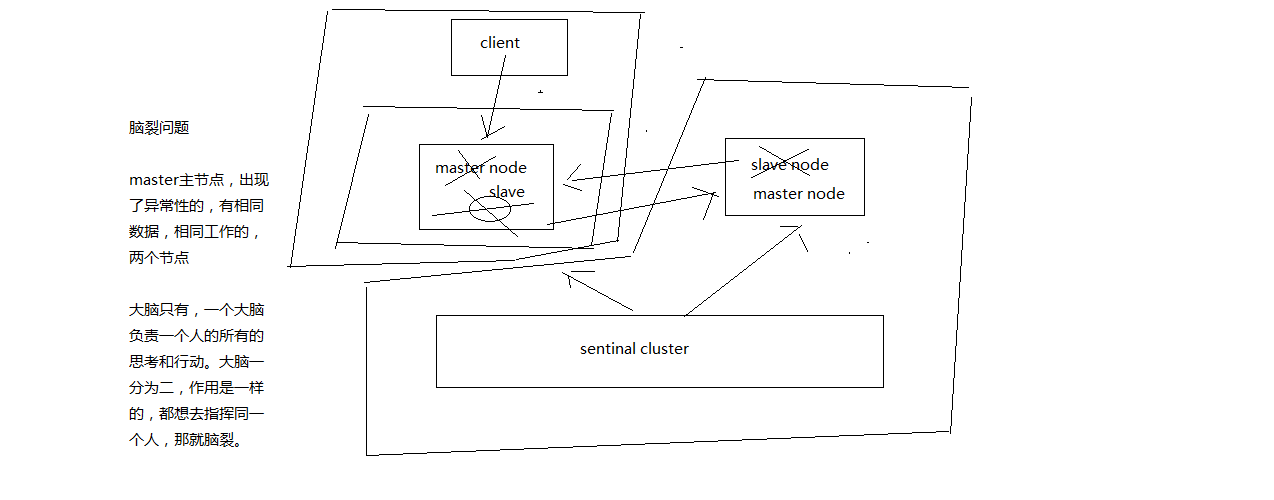
Configuration: quorum = 2，majority

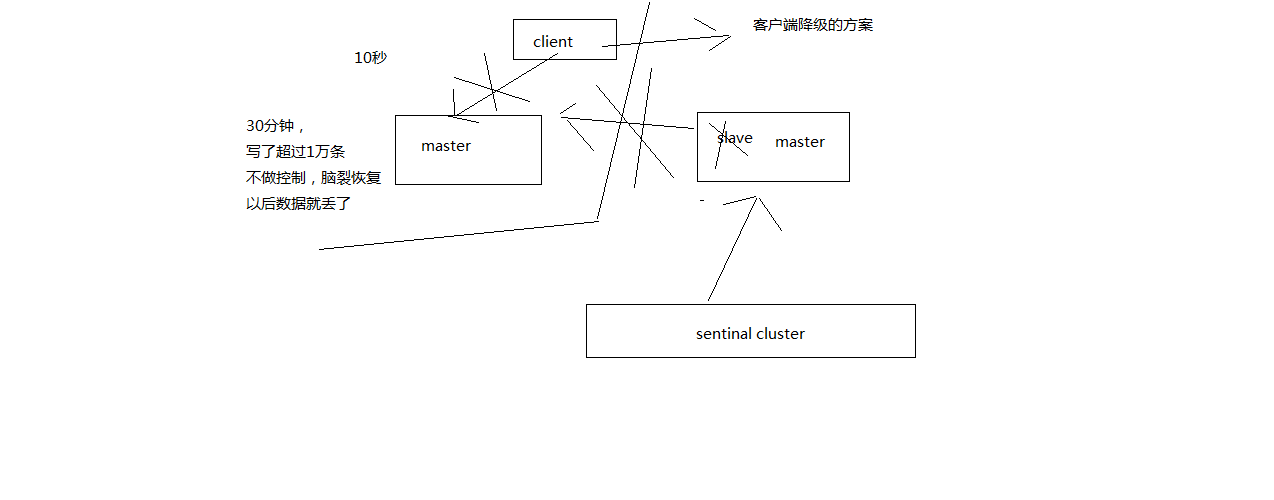
如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移

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

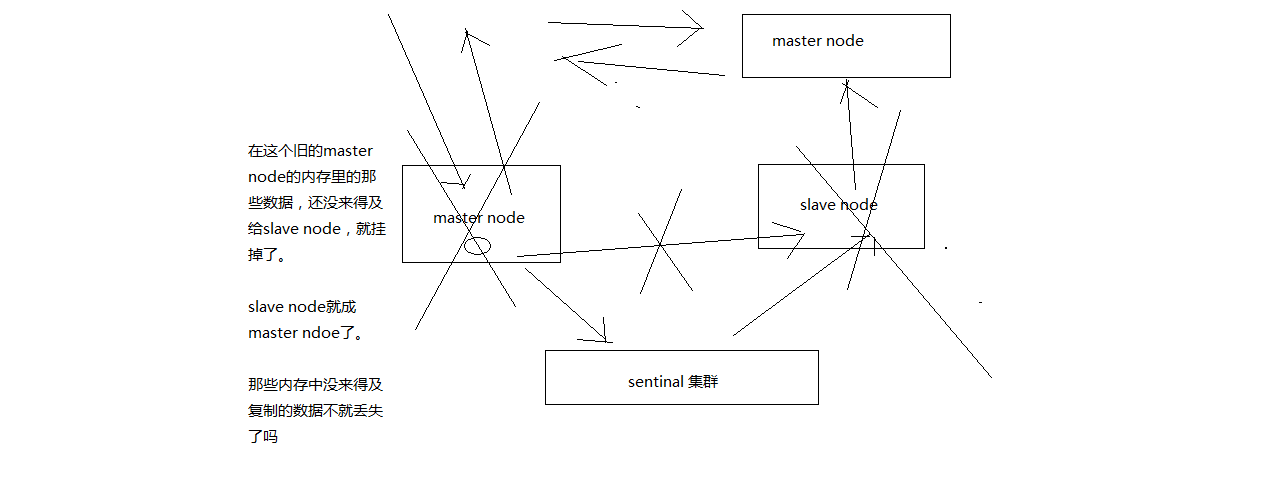
#### 22\_redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂

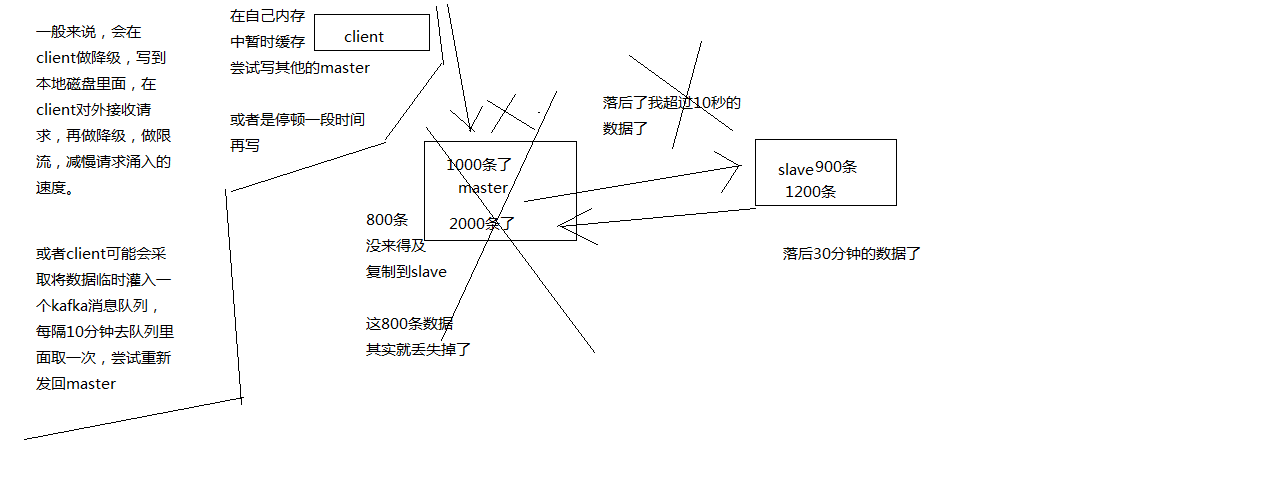
集群脑裂：





异步复制导致的脑裂问题：





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

#### 23\_redis哨兵的多个核心底层原理的深入解析（包含slave选举算法）

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

#### 24\_在项目中以经典的3节点方式部署哨兵集群

动手实操，练习如何操作部署哨兵集群，如何基于哨兵进行故障转移，还有一些企业级的配置方案

**1、哨兵的配置文件**

sentinel.conf

最小的配置

每一个哨兵都可以去监控多个maser-slaves的主从架构

因为可能你的公司里，为不同的项目，部署了多个master-slaves的redis主从集群

相同的一套哨兵集群，就可以去监控不同的多个redis主从集群

**你自己给每个redis主从集群分配一个逻辑的名称**

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel down-after-milliseconds mymaster 60000

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4

sentinel down-after-milliseconds resque 10000

sentinel failover-timeout resque 180000

sentinel parallel-syncs resque 5

sentinel monitor mymaster 127.0.0.1 6379

类似这种配置，来指定对一个master的监控，给监控的master指定的一个名称，因为后面分布式集群架构里会讲解，可以配置多个master做数据拆分

sentinel down-after-milliseconds mymaster 60000

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

上面的三个配置，都是针对某个监控的master配置的，给其指定上面分配的名称即可

上面这段配置，就监控了两个master node

这是最小的哨兵配置，如果发生了master-slave故障转移，或者新的哨兵进程加入哨兵集群，那么哨兵会自动更新自己的配置文件

sentinel monitor master-group-name hostname port quorum

quorum的解释如下：

（1）至少多少个哨兵要一致同意，master进程挂掉了，或者slave进程挂掉了，或者要启动一个故障转移操作

（2）quorum是用来识别故障的，真正执行故障转移的时候，还是要在哨兵集群执行选举，选举一个哨兵进程出来执行故障转移操作

（3）假设有5个哨兵，quorum设置了2，那么如果5个哨兵中的2个都认为master挂掉了; 2个哨兵中的一个就会做一个选举，选举一个哨兵出来，执行故障转移; 如果5个哨兵中有3个哨兵都是运行的，那么故障转移就会被允许执行

down-after-milliseconds，超过多少毫秒跟一个redis实例断了连接，哨兵就可能认为这个redis实例挂了

parallel-syncs，新的master别切换之后，同时有多少个slave被切换到去连接新master，重新做同步，数字越低，花费的时间越多

假设你的redis是1个master，4个slave

然后master宕机了，4个slave中有1个切换成了master，剩下3个slave就要挂到新的master上面去

这个时候，如果parallel-syncs是1，那么3个slave，一个一个地挂接到新的master上面去，1个挂接完，而且从新的master sync完数据之后，再挂接下一个

如果parallel-syncs是3，那么一次性就会把所有slave挂接到新的master上去

failover-timeout，执行故障转移的timeout超时时长

2、在eshop-cache03上再部署一个redis

只要安装redis就可以了，不需要去部署redis实例的启动

2、正式的配置

哨兵默认用26379端口，默认不能跟其他机器在指定端口连通，只能在本地访问

mkdir /etc/sentinal

mkdir -p /var/sentinal/5000

/etc/sentinel/5000.conf

port 5000

bind 192.168.60.5

dir /var/sentinal/5000

sentinel monitor mymaster 192.168.60.5 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel failover-timeout mymaster 60000

sentinel parallel-syncs mymaster 1

port 5000

bind 192.168.60.6

dir /var/sentinal/5000

sentinel monitor mymaster 192.168.60.5 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel failover-timeout mymaster 60000

sentinel parallel-syncs mymaster 1

port 5000

bind 192.168.60.7

dir /var/sentinal/5000

sentinel monitor mymaster 192.168.60.5 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel failover-timeout mymaster 60000

sentinel parallel-syncs mymaster 1

sentinel auth-pass mymaster redis-pass

3、启动哨兵进程

在eshop-cache01、eshop-cache02、eshop-cache03三台机器上，分别启动三个哨兵进程，组成一个集群，观察一下日志的输出

**redis-sentinel /etc/sentinal/5000.conf**

redis-server /etc/sentinal/5000.conf --sentinel

日志里会显示出来，每个哨兵都能去监控到对应的redis master，并能够自动发现对应的slave

哨兵之间，互相会自动进行发现，用的就是之前说的pub/sub，消息发布和订阅channel消息系统和机制

4、检查哨兵状态

redis-cli -h 192.168.60.5 -p 5000

sentinel master mymaster

SENTINEL slaves mymaster

SENTINEL sentinels mymaster

SENTINEL get-master-addr-by-name mymaster

#### 25\_对项目中的哨兵节点进行管理以及高可用redis集群的容灾演练

1、哨兵节点的增加和删除

增加sentinal，会自动发现

删除sentinal的步骤

（1）停止sentinal进程

（2）SENTINEL RESET \*，在所有sentinal上执行，清理所有的master状态

（3）SENTINEL MASTER mastername，在所有sentinal上执行，查看所有sentinal对数量是否达成了一致

2、slave的永久下线

让master摘除某个已经下线的slave：SENTINEL RESET mastername，在所有的哨兵上面执行

3、slave切换为Master的优先级

slave->master选举优先级：slave-priority，值越小优先级越高

4、基于哨兵集群架构下的安全认证

每个slave都有可能切换成master，所以每个实例都要配置两个指令

master上启用安全认证，requirepass

master连接口令，masterauth

sentinal，sentinel auth-pass <master-group-name> <pass>

5、容灾演练

通过哨兵看一下当前的master：SENTINEL get-master-addr-by-name mymaster

把master节点kill -9掉，pid文件也删除掉

查看sentinal的日志，是否出现+sdown字样，识别出了master的宕机问题; 然后出现+odown字样，就是指定的quorum哨兵数量，都认为master宕机了

（1）三个哨兵进程都认为master是sdown了

（2）超过quorum指定的哨兵进程都认为sdown之后，就变为odown

（3）哨兵1是被选举为要执行后续的主备切换的那个哨兵

（4）哨兵1去新的master（slave）获取了一个新的config version

（5）尝试执行failover

（6）投票选举出一个slave区切换成master，每隔哨兵都会执行一次投票

（7）让salve，slaveof noone，不让它去做任何节点的slave了; 把slave提拔成master; 旧的master认为不再是master了

（8）哨兵就自动认为之前的187:6379变成了slave了，19:6379变成了master了

（9）哨兵去探查了一下187:6379这个salve的状态，认为它sdown了

所有哨兵选举出了一个，来执行主备切换操作

如果哨兵的majority都存活着，那么就会执行主备切换操作

再通过哨兵看一下master：SENTINEL get-master-addr-by-name mymaster

尝试连接一下新的master

故障恢复，再将旧的master重新启动，查看是否被哨兵自动切换成slave节点

（1）手动杀掉master

（2）哨兵能否执行主备切换，将slave切换为master

（3）哨兵完成主备切换后，新的master能否使用

（4）故障恢复，将旧的master重新启动

（5）哨兵能否自动将旧的master变为slave，挂接到新的master上面去，而且也是可以使用的

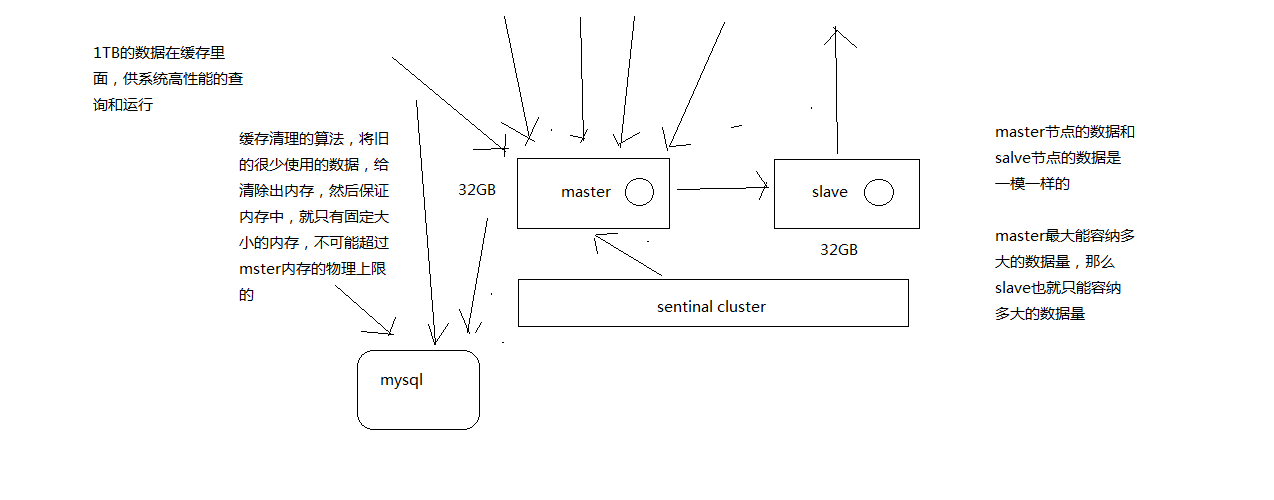
6、哨兵的生产环境部署

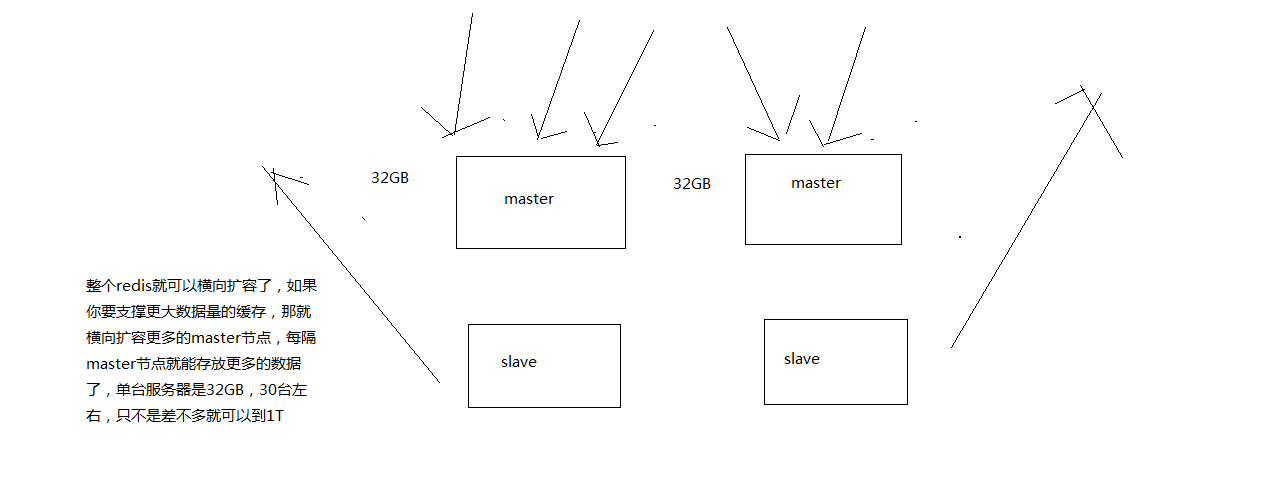
daemonize yes

logfile /var/log/sentinal/5000

mkdir -p /var/log/sentinal/5000

#### 26\_redis如何在保持读写分离+高可用的架构下，还能横向扩容支撑1T+海量数据





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

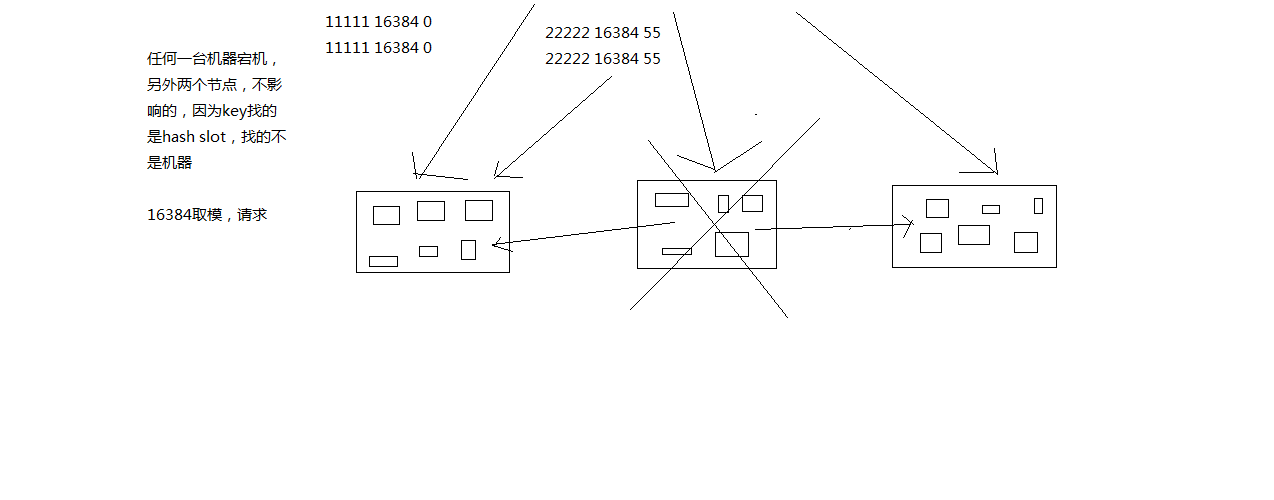
如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个G，单机足够了

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

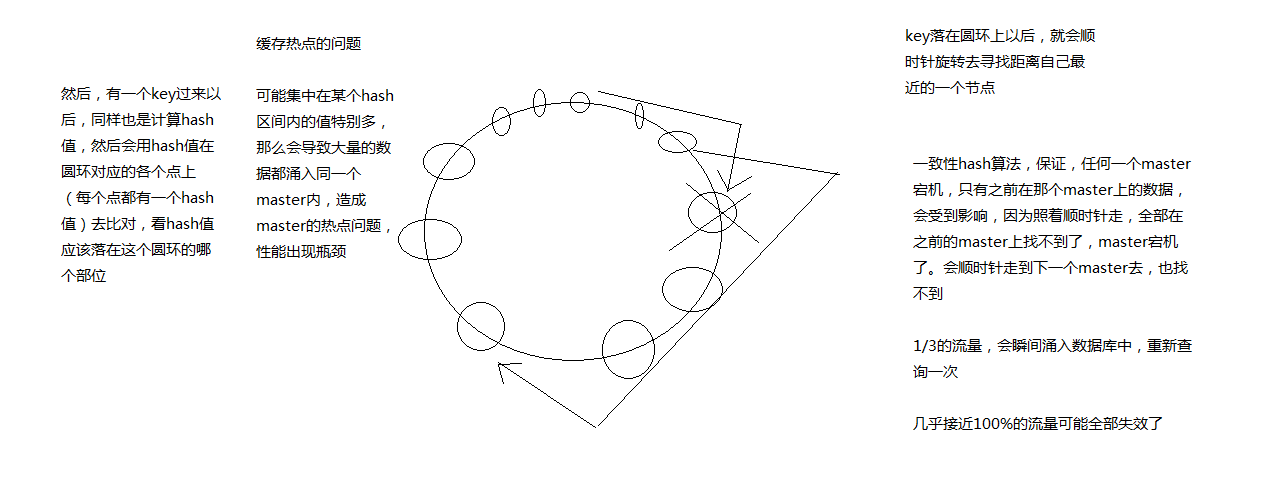
redis cluster，主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用redis cluster

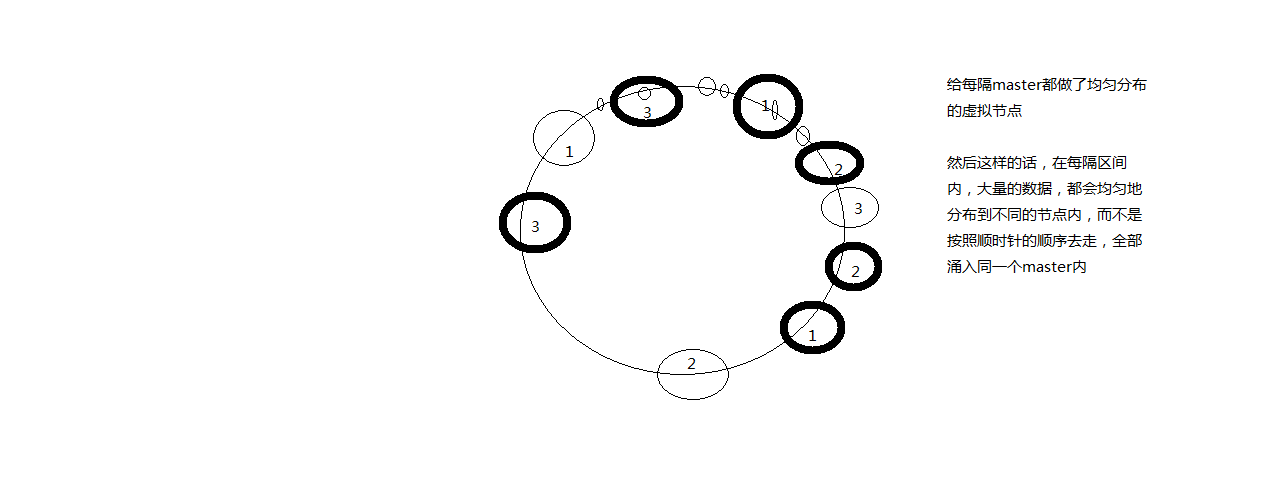
#### 27\_数据分布算法：hash+一致性hash+redis cluster的hash slot

**Hash slot原理**

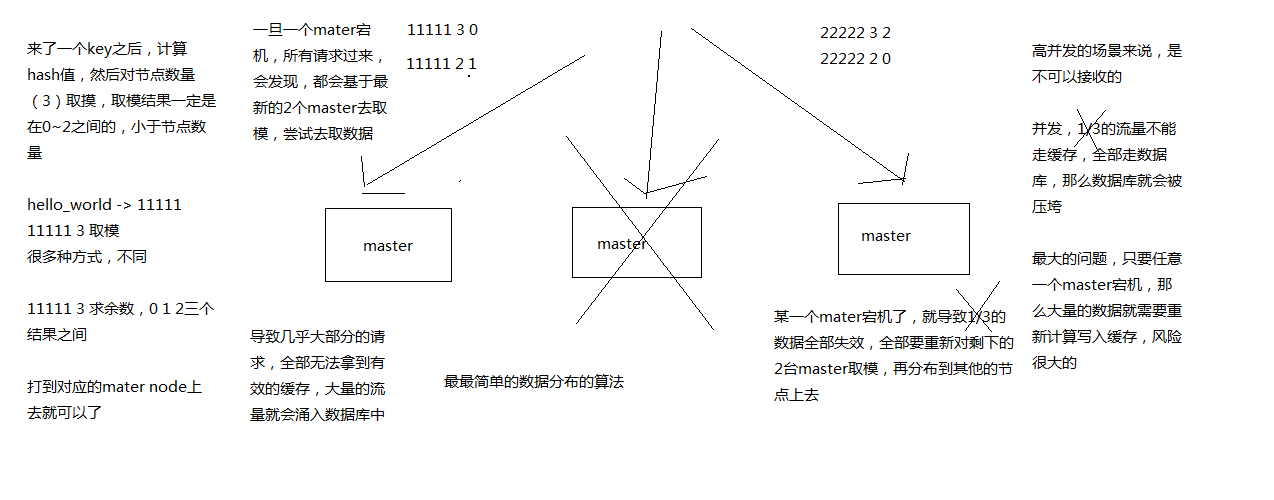


**一致性哈希原理**





**普通哈希原理**



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

#### 28\_在项目中重新搭建一套读写分离+高可用+多master的redis cluster集群

redis cluster最最基础的一些知识

**redis cluster: 自动，master+slave复制和读写分离，master+slave高可用和主备切换，支持多个master的hash slot支持数据分布式存储**

停止之前所有的实例，包括redis主从和哨兵集群

1、redis cluster的重要配置

cluster-enabled <yes/no>

cluster-config-file <filename>：这是指定一个文件，供cluster模式下的redis实例将集群状态保存在那里，包括集群中其他机器的信息，比如节点的上线和下限，故障转移，不是我们去维护的，给它指定一个文件，让redis自己去维护的

cluster-node-timeout <milliseconds>：节点存活超时时长，超过一定时长，认为节点宕机，master宕机的话就会触发主备切换，slave宕机就不会提供服务

2、在三台机器上启动6个redis实例

（1）在eshop-cache03上部署目录

/etc/redis（存放redis的配置文件），/var/redis/6379（存放redis的持久化文件）

（2）编写配置文件

redis cluster集群，要求至少3个master，去组成一个高可用，健壮的分布式的集群，每个master都建议至少给一个slave，3个master，3个slave，最少的要求

正式环境下，建议都是说在6台机器上去搭建，至少3台机器

保证，每个master都跟自己的slave不在同一台机器上，如果是6台自然更好，一个master+一个slave就死了

3台机器去搭建6个redis实例的redis cluster

mkdir -p /etc/redis-cluster

mkdir -p /var/log/redis

mkdir -p /var/redis/7001

port 7001

cluster-enabled yes

cluster-config-file /etc/redis-cluster/node-7001.conf

cluster-node-timeout 15000

daemonize yes

pidfile /var/run/redis\_7001.pid

dir /var/redis/7001

logfile /var/log/redis/7001.log

bind 192.168.31.187

appendonly yes

至少要用3个master节点启动，每个master加一个slave节点，先选择6个节点，启动6个实例

将上面的配置文件，在/etc/redis下放6个，分别为: 7001.conf，7002.conf，7003.conf，7004.conf，7005.conf，7006.conf

（3）准备生产环境的启动脚本

在/etc/init.d下，放6个启动脚本，分别为: redis\_7001, redis\_7002, redis\_7003, redis\_7004, redis\_7005, redis\_7006

每个启动脚本内，都修改对应的端口号

（4）分别在3台机器上，启动6个redis实例

将每个配置文件中的slaveof给删除

3、创建集群

下面方框内的内容废弃掉

=======================================================================

wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.gz

tar -zxvf ruby-2.3.1.tar.gz

./configure -prefix=/usr/local/ruby

make && make install

cd /usr/local/ruby

cp bin/ruby /usr/local/bin

cp bin/gem /usr/local/bin

wget http://rubygems.org/downloads/redis-3.3.0.gem

gem install -l ./redis-3.3.0.gem

gem list --check redis gem

=======================================================================

因为，以前比如公司里面搭建集群，公司里的机器的环境，运维会帮你做好很多事情

在讲课的话，我们手工用从零开始装的linux虚拟机去搭建，那肯定会碰到各种各样的问题

yum install -y ruby

yum install -y rubygems

gem install redis

cp /usr/local/redis-3.2.8/src/redis-trib.rb /usr/local/bin

redis-trib.rb create --replicas 1 192.168.31.187:7001 192.168.31.187:7002 192.168.31.19:7003 192.168.31.19:7004 192.168.31.227:7005 192.168.31.227:7006

上句改为

redis-trib.rb create --replicas 1 192.168.60.5:7001 192.168.60.5:7002 192.168.60.6:7003 192.168.60.6:7004 192.168.60.7:7005 192.168.60.7:7006

--replicas: 每个master有几个slave

6台机器，3个master，3个slave，尽量自己让master和slave不在一台机器上

yes

redis-trib.rb check 192.168.60.5:7001

4、读写分离+高可用+多master

读写分离：每个master都有一个slave

高可用：master宕机，slave自动被切换过去

多master：横向扩容支持更大数据量

#### 29\_对项目的redis cluster实验多master写入、读写分离、高可用性

redis cluster搭建起来了

redis cluster，提供了多个master，数据可以分布式存储在多个master上; 每个master都带着slave，自动就做读写分离; 每个master如果故障，那么久会自动将slave切换成master，高可用

redis cluster的基本功能，来测试一下

1、实验多master写入 -> 海量数据的分布式存储

你在redis cluster写入数据的时候，其实是你可以将请求发送到任意一个master上去执行

但是，每个master都会计算这个key对应的CRC16值，然后对16384个hashslot取模，找到key对应的hashslot，找到hashslot对应的master

如果对应的master就在自己本地的话，set mykey1 v1，mykey1这个key对应的hashslot就在自己本地，那么自己就处理掉了

但是如果计算出来的hashslot在其他master上，那么就会给客户端返回一个moved error，告诉你，你得到哪个master上去执行这条写入的命令

什么叫做多master的写入，就是每条数据只能存在于一个master上，不同的master负责存储不同的数据，分布式的数据存储

100w条数据，5个master，每个master就负责存储20w条数据，分布式数据存储

大型的java系统架构，还专注在大数据系统架构，分布式，分布式存储，hadoop hdfs，分布式资源调度，hadoop yarn，分布式计算，hadoop mapreduce/hive

分布式的nosql数据库，hbase，分布式的协调，zookeeper，分布式通用计算引擎，spark，分布式的实时计算引擎，storm

如果你要处理海量数据，就涉及到了一个名词，叫做大数据，只要涉及到大数据，那么其实就会涉及到分布式

redis cluster，分布式

大数据相关的系统，也涉及很多的java系统架构，高并发、高可用、高性能、可扩展、分布式系统

redis，高并发、高性能、每日上亿流量的大型电商网站的商品详情页系统的缓存架构，来讲解的，redis是作为大规模缓存架构中的底层的核心存储的支持

高并发、高性能、每日上亿流量，redis持久化 -> 灾难的时候，做数据恢复，复制 -> 读写分离，扩容slave，支撑更高的读吞吐，redis怎么支撑读QPS超过10万，几十万; 哨兵，在redis主从，一主多从，怎么保证99.99%可用性; redis cluster，海量数据

java架构课，架构思路和设计是很重要的，但是另外一点，我希望能够带着大家用真正java架构师的角度去看待一些技术，而不是仅仅停留在技术的一些细节的点

给大家从一些大数据的角度，去分析一下我们java架构领域中的一些技术

天下武功，都出自一脉，研究过各种大数据的系统，redis cluster讲解了很多原理，跟elasticsearch，很多底层的分布式原理，都是类似的

redis AOF，fsync

elasticsearch建立索引的时候，先写内存缓存，每秒钟把数据刷入os cache，接下来再每隔一定时间fsync到磁盘上去

redis cluster，写可以到任意master，任意master计算key的hashslot以后，告诉client，重定向，路由到其他mater去执行，分布式存储的一个经典的做法

elasticsearch，建立索引的时候，也会根据doc id/routing value，做路由，路由到某个其他节点，重定向到其他节点去执行

分布式的一些，hadoop，spark，storm里面很多核心的思想都是类似的

后面，马上把redis架构给讲完之后，就开始讲解业务系统的开发，包括高并发的商品详情页系统的大型的缓存架构，jedis cluster相关api去封装和测试对redis cluster的访问

jedis cluster api，就可以自动针对多个master进行写入和读取

2、实验不同master各自的slave读取 -> 读写分离

在这个redis cluster中，如果你要在slave读取数据，那么需要带上readonly指令，get mykey1

redis-cli -c启动，就会自动进行各种底层的重定向的操作

实验redis cluster的读写分离的时候，会发现有一定的限制性，默认情况下，redis cluster的核心的理念，主要是用slave做高可用的，每个master挂一两个slave，主要是做数据的热备，还有master故障时的主备切换，实现高可用的

redis cluster默认是不支持slave节点读或者写的，跟我们手动基于replication搭建的主从架构不一样的

slave node，readonly，get，这个时候才能在slave node进行读取

redis cluster，主从架构是出来，读写分离，复杂了点，也可以做，jedis客户端，对redis cluster的读写分离支持不太好的

默认的话就是读和写都到master上去执行的

如果你要让最流行的jedis做redis cluster的读写分离的访问，那可能还得自己修改一点jedis的源码，成本比较高

要不然你就是自己基于jedis，封装一下，自己做一个redis cluster的读写分离的访问api

核心的思路，就是说，redis cluster的时候，就没有所谓的读写分离的概念了

读写分离，是为了什么，主要是因为要建立一主多从的架构，才能横向任意扩展slave node去支撑更大的读吞吐量

redis cluster的架构下，实际上本身master就是可以任意扩展的，你如果要支撑更大的读吞吐量，或者写吞吐量，或者数据量，都可以直接对master进行横向扩展就可以了

也可以实现支撑更高的读吞吐的效果

不会去跟大家直接讲解的，很多东西都要带着一些疑问，未知，实际经过一些实验和操作之后，让你体会的更加深刻一些

redis cluster，主从架构，读写分离，没说错，没有撒谎

redis cluster，不太好，server层面，jedis client层面，对master做扩容，所以说扩容master，跟之前扩容slave，效果是一样的

3、实验自动故障切换 -> 高可用性

redis-trib.rb check 192.168.31.187:7001

比如把master1，187:7001，杀掉，看看它对应的19:7004能不能自动切换成master，可以自动切换

切换成master后的19:7004，可以直接读取数据

再试着把187:7001给重新启动，恢复过来，自动作为slave挂载到了19:7004上面去

#### 30\_redis cluster通过master水平扩容来支撑更高的读写吞吐+海量数据

实验，演示过了

redis cluster模式下，不建议做物理的读写分离了

我们建议通过master的水平扩容，来横向扩展读写吞吐量，还有支撑更多的海量数据

redis单机，读吞吐是5w/s，写吞吐2w/s

扩展redis更多master，那么如果有5台master，不就读吞吐可以达到总量25/s QPS，写可以达到10w/s QPS

redis单机，内存，6G，8G，fork类操作的时候很耗时，会导致请求延时的问题

扩容到5台master，能支撑的总的缓存数据量就是30G，40G -> 100台，600G，800G，甚至1T+，海量数据

redis是怎么扩容的

1、加入新master

mkdir -p /var/redis/7007

port 7007

cluster-enabled yes

cluster-config-file /etc/redis-cluster/node-7007.conf

cluster-node-timeout 15000

daemonize yes

pidfile /var/run/redis\_7007.pid

dir /var/redis/7007

logfile /var/log/redis/7007.log

bind 192.168.31.227

appendonly yes

搞一个7007.conf，再搞一个redis\_7007启动脚本

手动启动一个新的redis实例，在7007端口上

redis-trib.rb add-node 192.168.60.8:7008 192.168.60.5:7001

redis-trib.rb check 192.168.31.187:7001

连接到新的redis实例上，cluster nodes，确认自己是否加入了集群，作为了一个新的master

2、reshard一些数据过去

resharding的意思就是把一部分hash slot从一些node上迁移到另外一些node上

redis-trib.rb reshard 192.168.31.187:7001

要把之前3个master上，总共4096个hashslot迁移到新的第四个master上去

How many slots do you want to move (from 1 to 16384)?

1000

3、添加node作为slave

eshop-cache03

mkdir -p /var/redis/7008

port 7008

cluster-enabled yes

cluster-config-file /etc/redis-cluster/node-7008.conf

cluster-node-timeout 15000

daemonize yes

pidfile /var/run/redis\_7008.pid

dir /var/redis/7008

logfile /var/log/redis/7008.log

bind 192.168.31.227

appendonly yes

redis-trib.rb add-node --slave --master-id 28927912ea0d59f6b790a50cf606602a5ee48108 192.168.31.227:7008 192.168.31.187:7001

4、删除node

先用resharding将数据都移除到其他节点，确保node为空之后，才能执行remove操作

redis-trib.rb reshard 192.168.60.5:7001

redis-trib.rb del-node 192.168.60.5:7001 fc834b2be406e0e8f8906fef6af1c19110df97b9

2个是1365，1个是1366

当你清空了一个master的hashslot时，redis cluster就会自动将其slave挂载到其他master上去

这个时候就只要删除掉master就可以了

#### 31\_redis cluster的自动化slave迁移实现更强的高可用架构的部署方案

slave的自动迁移

比如现在有10个master，每个有1个slave，然后新增了3个slave作为冗余，有的master就有2个slave了，有的master出现了salve冗余

如果某个master的slave挂了，那么redis cluster会自动迁移一个冗余的slave给那个master

只要多加一些冗余的slave就可以了

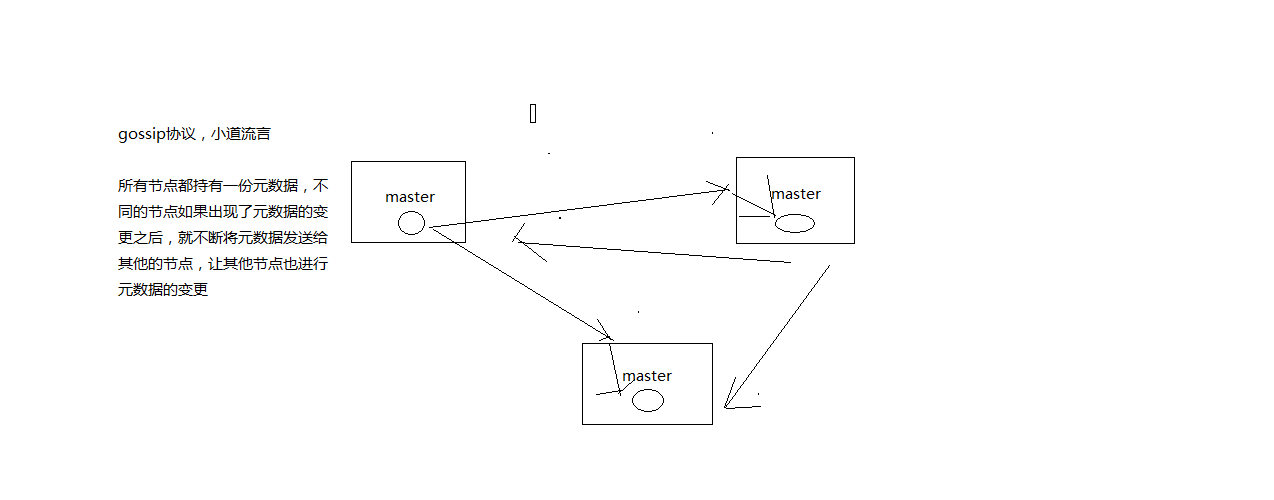
为了避免的场景，就是说，如果你每个master只有一个slave，万一说一个slave死了，然后很快，master也死了，那可用性还是降低了

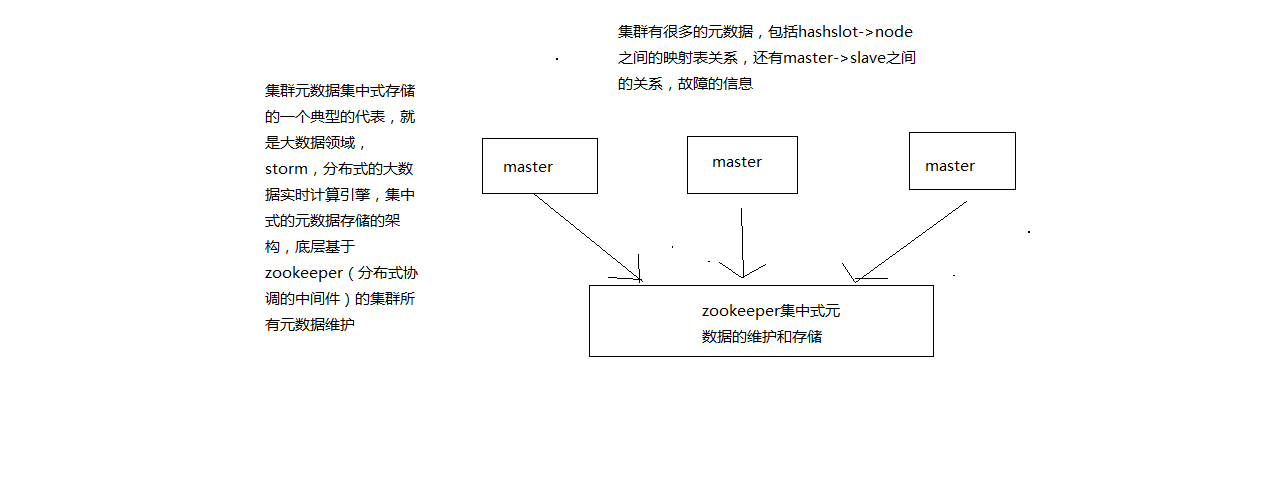
但是如果你给整个集群挂载了一些冗余slave，那么某个master的slave死了，冗余的slave会被自动迁移过去，作为master的新slave，此时即使那个master也死了

还是有一个slave会切换成master的

之前有一个master是有冗余slave的，直接让其他master其中的一个slave死掉，然后看有冗余slave会不会自动挂载到那个master

#### 32\_redis cluster的核心原理分析：gossip通信、jedis smart定位、主备切换





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

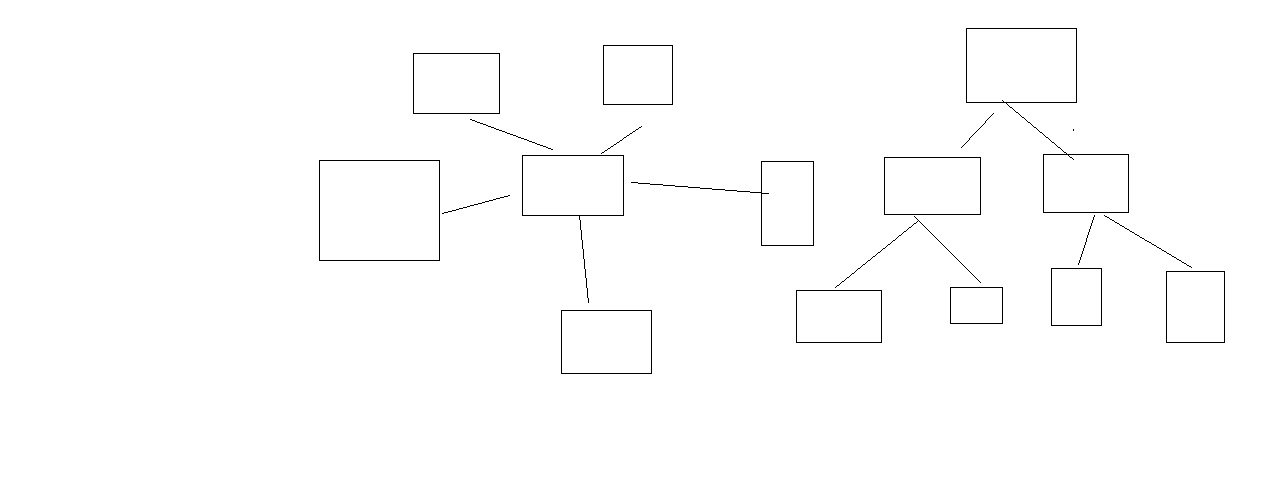
4、与哨兵比较

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

#### 33\_redis在实践中的一些常见问题以及优化思路（包含linux内核参数优化）



做各种压力测试，性能，redis-benchmark，并发，QPS，高可用的演练，每台机器最大能存储多少数据量，横向扩容支撑更多数据

基于测试环境还有测试数据，做各种演练，去摸索一些最适合自己的一些细节的东西

你说你靠一套课程，搞定一个技术100%的所有的东西，几乎是不可能的

spark等等课程

实际学了课程去做项目，100%会遇到大量自己没想到的问题，遇到了首先就自己尝试去解决，遇到问题，才是你的经验积累

spark，elasticsearch，java架构课程

1、fork耗时导致高并发请求延时

RDB和AOF的时候，其实会有生成RDB快照，AOF rewrite，耗费磁盘IO的过程，主进程fork子进程

fork的时候，子进程是需要拷贝父进程的空间内存页表的，也是会耗费一定的时间的

一般来说，如果父进程内存有1个G的数据，那么fork可能会耗费在20ms左右，如果是10G~30G，那么就会耗费20 \* 10，甚至20 \* 30，也就是几百毫秒的时间

info stats中的latest\_fork\_usec，可以看到最近一次form的时长

redis单机QPS一般在几万，fork可能一下子就会拖慢几万条操作的请求时长，从几毫秒变成1秒

优化思路

fork耗时跟redis主进程的内存有关系，一般控制redis的内存在10GB以内，slave -> master，全量复制

2、AOF的阻塞问题

redis将数据写入AOF缓冲区，单独开一个现场做fsync操作，每秒一次

但是redis主线程会检查两次fsync的时间，如果距离上次fsync时间超过了2秒，那么写请求就会阻塞

everysec，最多丢失2秒的数据

一旦fsync超过2秒的延时，整个redis就被拖慢

优化思路

优化硬盘写入速度，建议采用SSD，不要用普通的机械硬盘，SSD，大幅度提升磁盘读写的速度

3、主从复制延迟问题

主从复制可能会超时严重，这个时候需要良好的监控和报警机制

在info replication中，可以看到master和slave复制的offset，做一个差值就可以看到对应的延迟量

如果延迟过多，那么就进行报警

4、主从复制风暴问题

如果一下子让多个slave从master去执行全量复制，一份大的rdb同时发送到多个slave，会导致网络带宽被严重占用

如果一个master真的要挂载多个slave，那尽量用树状结构，不要用星型结构

5、vm.overcommit\_memory

0: 检查有没有足够内存，没有的话申请内存失败

1: 允许使用内存直到用完为止

2: 内存地址空间不能超过swap + 50%

如果是0的话，可能导致类似fork等操作执行失败，申请不到足够的内存空间

cat /proc/sys/vm/overcommit\_memory

echo "vm.overcommit\_memory=1" >> /etc/sysctl.conf

sysctl vm.overcommit\_memory=1

6、swapiness

cat /proc/version，查看linux内核版本

如果linux内核版本<3.5，那么swapiness设置为0，这样系统宁愿swap也不会oom killer（杀掉进程）

如果linux内核版本>=3.5，那么swapiness设置为1，这样系统宁愿swap也不会oom killer

保证redis不会被杀掉

echo 0 > /proc/sys/vm/swappiness

echo vm.swapiness=0 >> /etc/sysctl.conf

7、最大打开文件句柄

ulimit -n 10032 10032

自己去上网搜一下，不同的操作系统，版本，设置的方式都不太一样

8、tcp backlog

cat /proc/sys/net/core/somaxconn

echo 511 > /proc/sys/net/core/somaxconn

#### 34\_redis阶段性总结：1T以上海量数据+10万以上QPS高并发+99.99%高可用

1、讲解redis是为了什么？

topic：高并发、亿级流量、高性能、海量数据的场景，电商网站的商品详情页系统的缓存架构

商品详情页系统，大型电商网站，会有很多部分组成，但是支撑高并发、亿级流量的，主要就是其中的大型的缓存架构

在这个大型的缓存架构中，redis是最最基础的一层

高并发，缓存架构中除了redis，还有其他的组成部分，但是redis至关重要

大量的离散请求，随机请求，各种你未知的用户过来的请求，上千万用户过来访问，每个用户访问10次; 集中式的请求，1个用户过来，一天访问1亿次

支撑商品展示的最重要的，就是redis cluster，去抗住每天上亿的请求流量，支撑高并发的访问

redis cluster在整个缓存架构中，如何跟其他几个部分搭配起来组成一个大型的缓存系统，后面再讲

2、讲解的redis可以实现什么效果？

我之前一直在redis的各个知识点的讲解之前都强调一下，我们要讲解的每个知识点，要解决的问题是什么？？？

redis：持久化、复制（主从架构）、哨兵（高可用，主备切换）、redis cluster（海量数据+横向扩容+高可用/主备切换）

持久化：高可用的一部分，在发生redis集群灾难的情况下（比如说部分master+slave全部死掉了），如何快速进行数据恢复，快速实现服务可用，才能实现整个系统的高可用

复制：主从架构，master -> slave 复制，读写分离的架构，写master，读slave，横向扩容slave支撑更高的读吞吐，读高并发，10万，20万，30万，上百万，QPS，横向扩容

哨兵：高可用，主从架构，在master故障的时候，快速将slave切换成master，实现快速的灾难恢复，实现高可用性

redis cluster：多master读写，数据分布式的存储，横向扩容，水平扩容，快速支撑高达的数据量+更高的读写QPS，自动进行master -> slave的主备切换，高可用

让底层的缓存系统，redis，实现能够任意水平扩容，支撑海量数据（1T+，几十T，10G \* 600 redis = 6T），支撑很高的读写QPS（redis单机在几万QPS，10台，几十万QPS），高可用性（给我们每个redis实例都做好AOF+RDB的备份策略+容灾策略，slave -> master主备切换）

1T+海量数据、10万+读写QPS、99.99%高可用性

3、redis的第一套企业级的架构

如果你的数据量不大，单master就可以容纳，一般来说你的缓存的总量在10G以内就可以，那么建议按照以下架构去部署redis

redis持久化+备份方案+容灾方案+replication（主从+读写分离）+sentinal（哨兵集群，3个节点，高可用性）

可以支撑的数据量在10G以内，可以支撑的写QPS在几万左右，可以支撑的读QPS可以上10万以上（随你的需求，水平扩容slave节点就可以），可用性在99.99%

4、redis的第二套企业级架构

如果你的数据量很大，比如我们课程的topic，大型电商网站的商品详情页的架构（对标那些国内排名前三的大电商网站，\*宝，\*东，\*宁易购），数据量是很大的

海量数据

redis cluster

多master分布式存储数据，水平扩容

支撑更多的数据量，1T+以上没问题，只要扩容master即可

读写QPS分别都达到几十万都没问题，只要扩容master即可，redis cluster，读写分离，支持不太好，readonly才能去slave上读

支撑99.99%可用性，也没问题，slave -> master的主备切换，冗余slave去进一步提升可用性的方案（每个master挂一个slave，但是整个集群再加个3个slave冗余一下）

我们课程里，两套架构都讲解了，后续的业务系统的开发，主要是基于redis cluster去做

5、我们现在课程讲解的项目进展到哪里了？

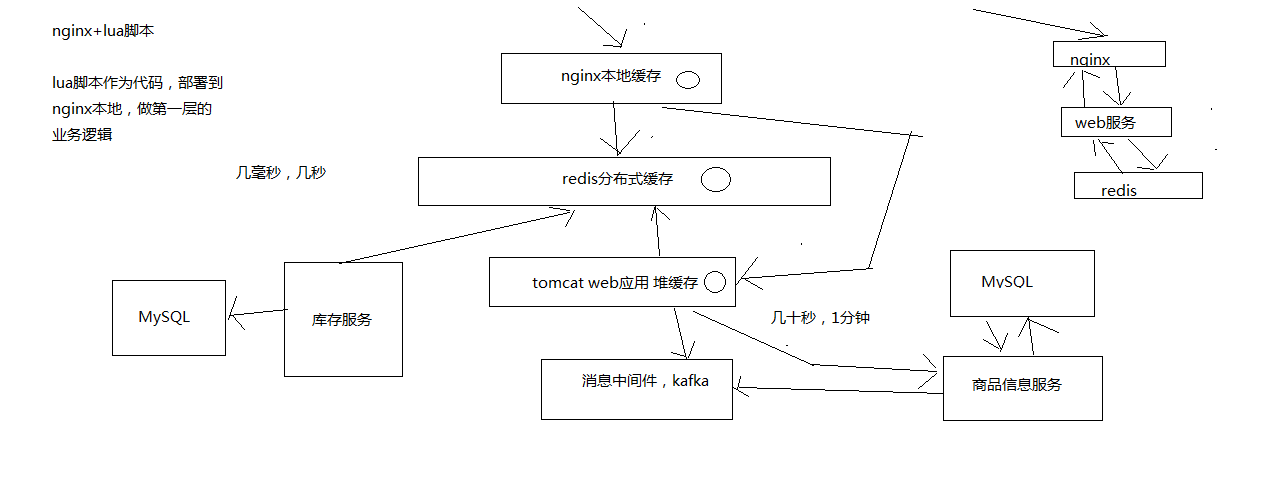
我们要做后续的业务系统的开发，redis的架构部署好，是第一件事情，也是非常重要的，也是你作为一个架构师而言，在对系统进行设计的时候，你必须要考虑到底层的redis的并发、性能、能支撑的数据量、可用性

redis：水平扩容，海量数据，上10万的读写QPS，99.99%高可用性

从架构的角度，我们的redis是可以做到的，水平扩容，只要机器足够，到1T数据量，50万读写QPS，99.99%

正式开始做大型电商网站的商品详情页系统，大规模的缓存架构设计

#### 35\_亿级流量商品详情页的多级缓存架构以及架构中每一层的意义



我们之前的三十讲，主要是在讲解redis如何支撑海量数据、高并发读写、高可用服务的架构，redis架构

redis架构，在我们的真正类似商品详情页读高并发的系统中，redis就是底层的缓存存储的支持

从这一讲开始，我们正式开始做业务系统的开发

亿级流量以上的电商网站的商品详情页的系统，商品详情页系统，大量的业务，十几个人做一两年，堆出来复杂的业务系统

几十个小时的课程，讲解复杂的业务

把整体的架构给大家讲解清楚，然后浓缩和精炼里面的业务，提取部分业务，做一些简化，把整个详情页系统的流程跑出来

架构，骨架，有少量的业务，血和肉，把整个项目串起来，在业务背景下，去学习架构

讲解商品详情页系统，缓存架构，90%大量的业务代码（没有什么技术含量），10%的最优技术含量的就是架构，上亿流量，每秒QPS几万，上十万的，读并发

读并发，缓存架构

1、上亿流量的商品详情页系统的多级缓存架构

采用三级缓存：nginx本地缓存+redis分布式缓存+tomcat堆缓存的多级缓存架构

时效性要求非常高的数据：库存

一般来说，显示的库存，都是时效性要求会相对高一些，因为随着商品的不断的交易，库存会不断的变化

当然，我们就希望当库存变化的时候，尽可能更快将库存显示到页面上去，而不是说等了很长时间，库存才反应到页面上去

时效性要求不高的数据：商品的基本信息（名称、颜色、版本、规格参数，等等）

时效性要求不高的数据，就还好，比如说你现在改变了商品的名称，稍微晚个几分钟反应到商品页面上，也还能接受

商品价格/库存等时效性要求高的数据，而且种类较少，采取相关的服务系统每次发生了变更的时候，直接采取数据库和redis缓存双写的方案，这样缓存的时效性最高

商品基本信息等时效性不高的数据，而且种类繁多，来自多种不同的系统，采取MQ异步通知的方式，写一个数据生产服务，监听MQ消息，然后异步拉取服务的数据，更新tomcat jvm缓存+redis缓存

nginx+lua脚本做页面动态生成的工作，每次请求过来，优先从nginx本地缓存中提取各种数据，结合页面模板，生成需要的页面

如果nginx本地缓存过期了，那么就从nginx到redis中去拉取数据，更新到nginx本地

如果redis中也被LRU算法清理掉了，那么就从nginx走http接口到后端的服务中拉取数据，数据生产服务中，现在本地tomcat里的jvm堆缓存中找，ehcache，如果也被LRU清理掉了，那么就重新发送请求到源头的服务中去拉取数据，然后再次更新tomcat堆内存缓存+redis缓存，并返回数据给nginx，nginx缓存到本地

2、多级缓存架构中每一层的意义

nginx本地缓存，抗的是热数据的高并发访问，一般来说，商品的购买总是有热点的，比如每天购买iphone、nike、海尔等知名品牌的东西的人，总是比较多的

这些热数据，利用nginx本地缓存，由于经常被访问，所以可以被锁定在nginx的本地缓存内

大量的热数据的访问，就是经常会访问的那些数据，就会被保留在nginx本地缓存内，那么对这些热数据的大量访问，就直接走nginx就可以了

那么大量的访问，直接就可以走到nginx就行了，不需要走后续的各种网络开销了

redis分布式大规模缓存，抗的是很高的离散访问，支撑海量的数据，高并发的访问，高可用的服务

redis缓存最大量的数据，最完整的数据和缓存，1T+数据; 支撑高并发的访问，QPS最高到几十万; 可用性，非常好，提供非常稳定的服务

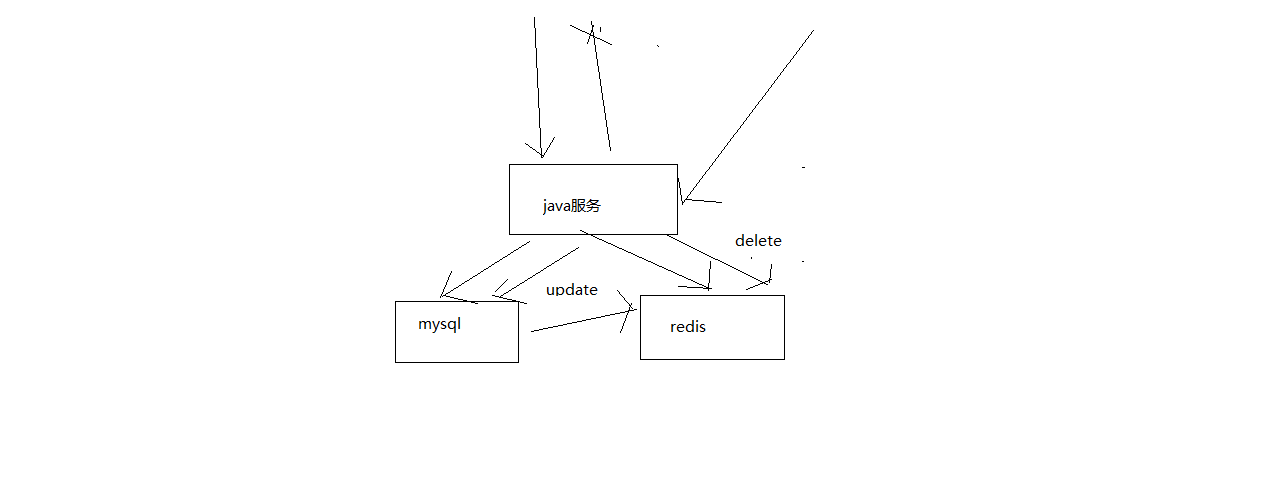
nginx本地内存有限，也就能cache住部分热数据，除了各种iphone、nike等热数据，其他相对不那么热的数据，可能流量会经常走到redis那里

利用redis cluster的多master写入，横向扩容，1T+以上海量数据支持，几十万的读写QPS，99.99%高可用性，那么就可以抗住大量的离散访问请求

tomcat jvm堆内存缓存，主要是抗redis大规模灾难的，如果redis出现了大规模的宕机，导致nginx大量流量直接涌入数据生产服务，那么最后的tomcat堆内存缓存至少可以再抗一下，不至于让数据库直接裸奔

同时tomcat jvm堆内存缓存，也可以抗住redis没有cache住的最后那少量的部分缓存

#### 36\_Cache Aside Pattern缓存+数据库读写模式的分析



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**

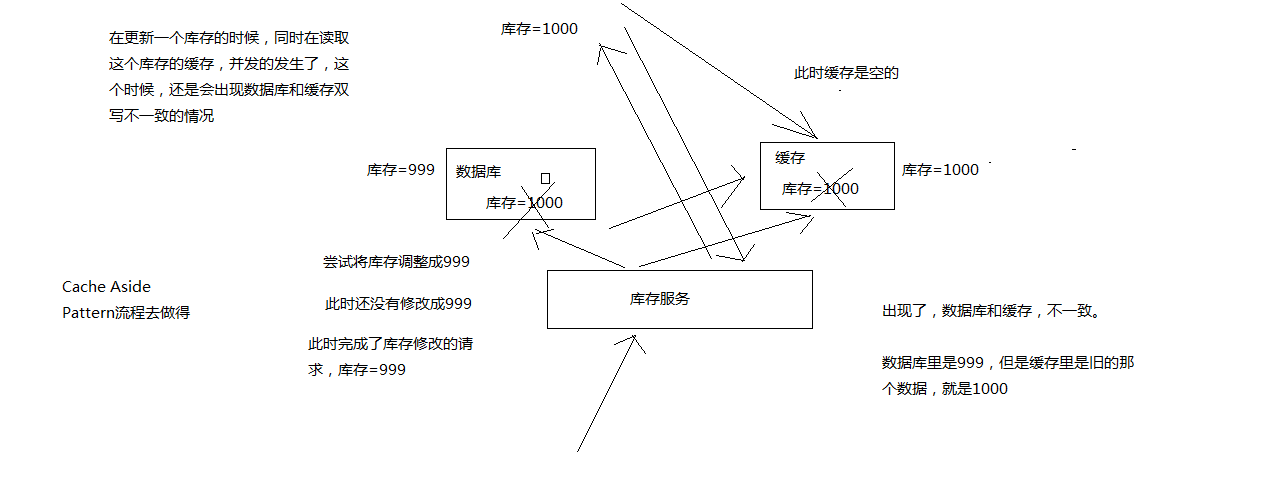
查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊

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

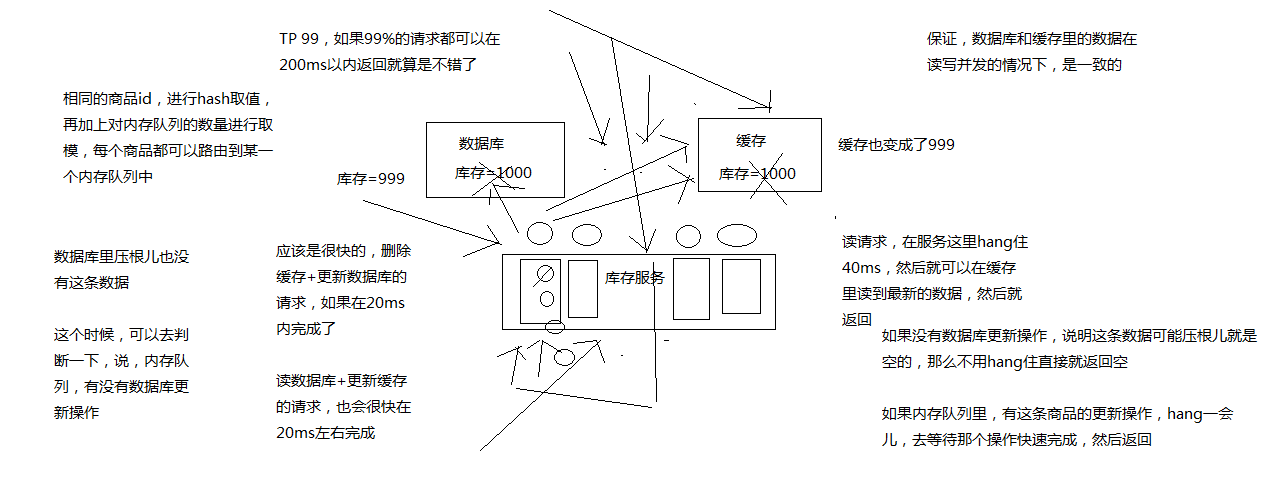
先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工

#### 37\_高并发场景下的缓存+数据库双写不一致问题分析与解决方案设计

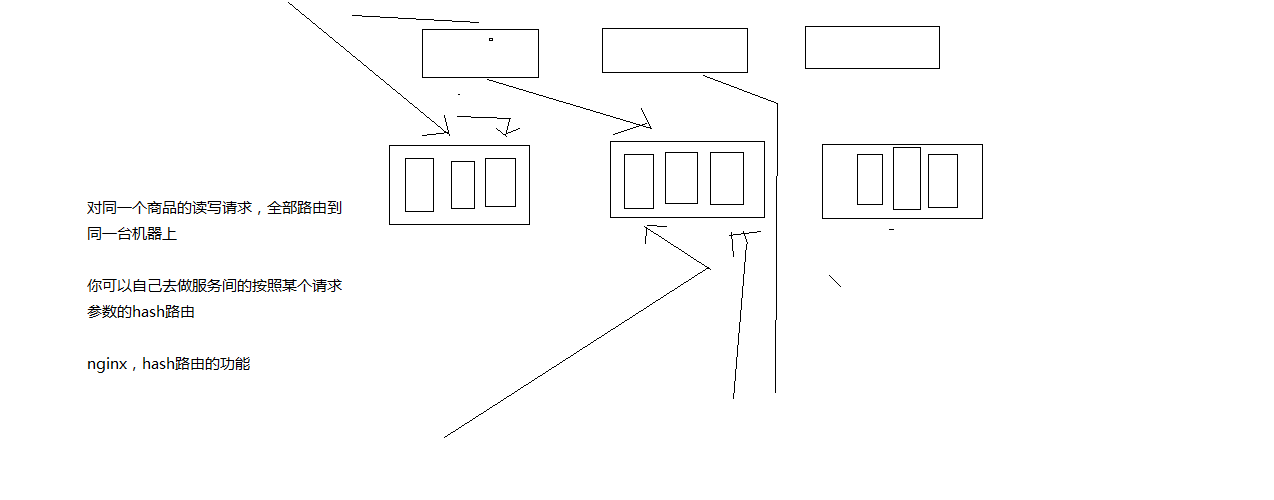
读写并发的时候复杂的数据库+缓存双写不一致的场景



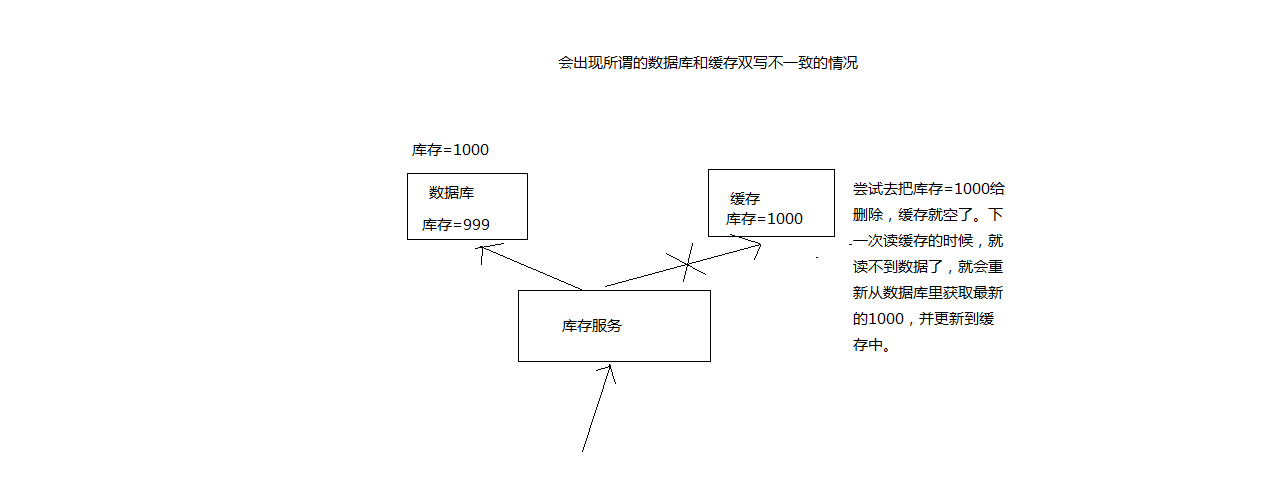
**复杂的数据库+缓存双写一致保障方案**



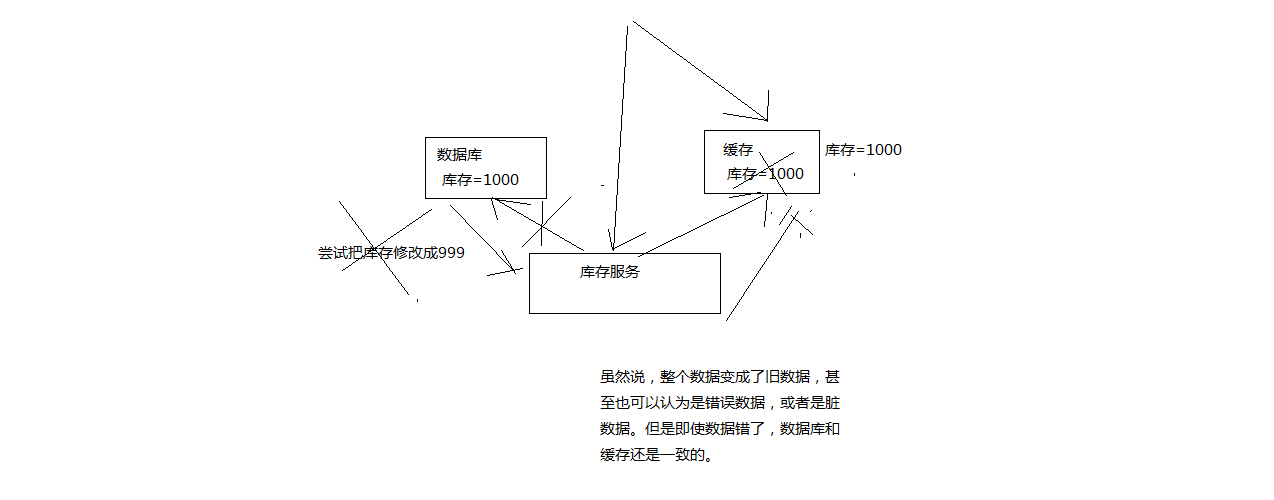
**机器级别的请求路由问题**



**最初级的数据库+缓存双写不一致问题**



**最初级的数据库+缓存双写不一致问题的解决方案**



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

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

#### 38\_在linux虚拟机中安装部署MySQL数据库

后面写的各种代码，还是要基于mysql去做一些开发的，因为缓存的底层的数据存储肯定是数据库

mysql数据库，还是要部署一下的

eshop-cache04

先用yum安装mysql server

yum install -y mysql-server

service mysqld start

chkconfig mysqld on

yum install -y mysql-connector-java

#### 39\_库存服务的开发框架整合与搭建：spring boot+mybatis+jedis

1、pom.xml

然后需要执行mvn install命令，手动安装依赖

mvn install:install-file -Dfile=F:\apache-maven-3.0.5\mvn\_repo\redis\clients\jedis\2.5.2\jedis-2.5.2.jar -DgroupId=redis.clients -DartifactId=jedis -Dversion=2.5.2 -Dpackaging=jar

可能需要将eclipse强制关闭，任务管理器里面，直接强制结束任务，关闭eclipse，重新打开

<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>

2、**com.roncoo.eshop.inventory.Application**

我们是直接讲解项目的，项目中遇到的一些技术，比如说redis，是缓存架构topic里面非常重要的一个环节，那我们当然会花费很大的篇幅去仔细讲解

但是比如spring boot、mybatis、zookeeper、storm

有些技术，是默认大家要会的，spring boot，mybatis，你说你都不会，不行，自己去学一下

zookeeper，kafka，技术，直接带着大家部署，简单介绍一下，就开始用了，如果真的是要用这个技术，做项目，还是得上网自己去查阅一些资料，学习这个技术

storm，会带着大家把20%的核心基础知识学习一下，作为java程序员，可以开发就行了

一个项目课程，涉及了十几种技术，我不可能全都给你按照从入门到精通的方式，全部讲解一遍

@EnableAutoConfiguration

@SpringBootApplication

@ComponentScan

@MapperScan("com.roncoo.eshop.inventory.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);

}

}

3、com.roncoo.eshop.inventory.model.User

4、com.roncoo.eshop.inventory.mapper.UserMapper

public interface UserMapper {

public User findUserInfo();

}

5、com.roncoo.eshop.inventory.service.UserService

6、UserController

@Controller

public class UserController {

@Autowired

private UserService userService;

@RequestMapping("/getUserInfo")

@ResponseBody

public User getUserInfo() {

User user = userService.getUserInfo();

return user;

}

}

7、resources/Application.properties

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

8、resources/mybatis/UserMapper.xml

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

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.roncoo.eshop.inventory.mapper.UserMapper">

<select id="findUserInfo" resultType="com.roncoo.eshop.inventory.model.User">

select name,age from user;

</select>

</mapper>

9、测试spring boot+mybatis的整合

在数据库中创建一个eshop database，创建一个eshop账号和密码，创建一个user表，里面插入一条数据，张三和25岁

create database if not exists eshop;

grant all privileges on eshop.\* to 'eshop'@'%' identified by 'eshop';

create table user(name varchar(255), age int)

insert into user values('张三', 25)

启动Application程序，访问getUserInfo接口，能否从mysql中查询数据，并返回到页面上

如果可以，说明spring boot+mybatis整合成功

10、整合Jedis Cluster

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

Application

@Bean

public JedisCluster **JedisClusterFactory**() {

Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();

jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7003));

jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7004));

jedisClusterNodes.add(new HostAndPort("192.168.31.227", 7006));

JedisCluster jedisCluster = new JedisCluster(jedisClusterNodes);

return jedisCluster;

}

@Repository("redisDAO")

public class RedisDAOImpl implements RedisDAO {

@Resource

private JedisCluster jedisCluster;

@Override

public void set(String key, String value) {

jedisCluster.set(key, value);

}

@Override

public String get(String key) {

return jedisCluster.get(key);

}

}

UserServiceImpl

@Override

public User getCachedUserInfo() {

redisDAO.set("cached\_user", "{\"name\": \"zhangsan\", \"age\": 25}") ;

String json = redisDAO.get("cached\_user");

JSONObject jsonObject = JSONObject.parseObject(json);

User user = new User();

user.setName(jsonObject.getString("name"));

user.setAge(jsonObject.getInteger("age"));

return user;

}

UserController

@RequestMapping("/getCachedUserInfo")

@ResponseBody

public User getCachedUserInfo() {

User user = userService.getCachedUserInfo();

return user;

}

#### 40\_在库存服务中实现缓存与数据库双写一致性保障方案（一）

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

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

(queueNum - 1) & hash

1、线程池+内存队列初始化

@Bean

public ServletListenerRegistrationBean servletListenerRegistrationBean(){

ServletListenerRegistrationBean servletListenerRegistrationBean = new ServletListenerRegistrationBean();

servletListenerRegistrationBean.setListener(new InitListener());

return servletListenerRegistrationBean;

}

java web应用，做系统的初始化，一般在哪里做呢？

ServletContextListener里面做，listener，会跟着整个web应用的启动，就初始化，类似于线程池初始化的构建

spring boot应用，Application，搞一个listener的注册

2、两种请求对象封装

3、请求异步执行Service封装

4、两种请求Controller接口封装

5、读请求去重优化

6、空数据读请求过滤优化

队列

对一个商品的库存的数据库更新操作已经在内存队列中了

然后对这个商品的库存的读取操作，要求读取数据库的库存数据，然后更新到缓存中，多个读

这多个读，其实只要有一个读请求操作压到队列里就可以了

其他的读操作，全部都wait那个读请求的操作，刷新缓存，就可以读到缓存中的最新数据了

如果读请求发现redis缓存中没有数据，就会发送读请求给库存服务，但是此时缓存中为空，可能是因为写请求先删除了缓存，也可能是数据库里压根儿没这条数据

如果是数据库中压根儿没这条数据的场景，那么就不应该将读请求操作给压入队列中，而是直接返回空就可以了

都是为了减少内存队列中的请求积压，内存队列中积压的请求越多，就可能导致每个读请求hang住的时间越长，也可能导致多个读请求被hang住

这边跟大家提前说一下，打个招呼

我的风格，跟其他一些人不太一样，我都是课上纯实时手写代码的，没有说提前自己先练个10遍（也没那时间），也不是说预先写好，然后放ppt，课程里就copy粘贴

我就是思路梳理出来，然后边讲课边写代码

就是跟真实的开发的时候是一样的，可能会犯各种各样的错误，可能会写出来bug

有可能有些错误，或者是疏忽的地方，自己课程上就发现了，然后就纠正了; 也可能我没发现自己一些遗漏的地方，QQ+我，然后跟我反馈

视频随时可以补充进去，纠错的一些东西

talk is cheap, show me the code

写代码，还有一个问题，我们是课程上现场写代码，第一遍写代码，写出来的代码，面向对象的设计，设计模式的运用，然后代码结构的设置，都不是太完美，甚至可能是有点粗糙

公司里做项目，其实那个代码会改来改去，重构，代码会变得越来越规整

课程里没办法，大家如果觉得我的代码写的不太好，比如说觉得我哪里本来应该用个设计模式，结果没有用

很重要的废话

这套课程，主要还是讲解架构的，面向架构

架构，很虚，说说说

架构还是得落地，落地还是要写代码的，不管大家是什么层级的人，开发，高工，架构师，技术总监，项目经理

所有的代码跟着一点一点写一下

#### 41\_在库存服务中实现缓存与数据库双写一致性保障方案（二）

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

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

(queueNum - 1) & hash

1、线程池+内存队列初始化

@Bean

public ServletListenerRegistrationBean servletListenerRegistrationBean(){

ServletListenerRegistrationBean servletListenerRegistrationBean = new ServletListenerRegistrationBean();

servletListenerRegistrationBean.setListener(new InitListener());

return servletListenerRegistrationBean;

}

java web应用，做系统的初始化，一般在哪里做呢？

ServletContextListener里面做，listener，会跟着整个web应用的启动，就初始化，类似于线程池初始化的构建

spring boot应用，Application，搞一个listener的注册

2、两种请求对象封装

3、请求异步执行Service封装

4、两种请求Controller接口封装

5、读请求去重优化

6、空数据读请求过滤优化

队列

对一个商品的库存的数据库更新操作已经在内存队列中了

然后对这个商品的库存的读取操作，要求读取数据库的库存数据，然后更新到缓存中，多个读

这多个读，其实只要有一个读请求操作压到队列里就可以了

其他的读操作，全部都wait那个读请求的操作，刷新缓存，就可以读到缓存中的最新数据了

如果读请求发现redis缓存中没有数据，就会发送读请求给库存服务，但是此时缓存中为空，可能是因为写请求先删除了缓存，也可能是数据库里压根儿没这条数据

如果是数据库中压根儿没这条数据的场景，那么就不应该将读请求操作给压入队列中，而是直接返回空就可以了

都是为了减少内存队列中的请求积压，内存队列中积压的请求越多，就可能导致每个读请求hang住的时间越长，也可能导致多个读请求被hang住

这边跟大家提前说一下，打个招呼

我的风格，跟其他一些人不太一样，我都是课上纯实时手写代码的，没有说提前自己先练个10遍（也没那时间），也不是说预先写好，然后放ppt，课程里就copy粘贴

我就是思路梳理出来，然后边讲课边写代码

就是跟真实的开发的时候是一样的，可能会犯各种各样的错误，可能会写出来bug

有可能有些错误，或者是疏忽的地方，自己课程上就发现了，然后就纠正了; 也可能我没发现自己一些遗漏的地方，QQ+我，然后跟我反馈

视频随时可以补充进去，纠错的一些东西

talk is cheap, show me the code

写代码，还有一个问题，我们是课程上现场写代码，第一遍写代码，写出来的代码，面向对象的设计，设计模式的运用，然后代码结构的设置，都不是太完美，甚至可能是有点粗糙

公司里做项目，其实那个代码会改来改去，重构，代码会变得越来越规整

课程里没办法，大家如果觉得我的代码写的不太好，比如说觉得我哪里本来应该用个设计模式，结果没有用

很重要的废话

这套课程，主要还是讲解架构的，面向架构

架构，很虚，说说说

架构还是得落地，落地还是要写代码的，不管大家是什么层级的人，开发，高工，架构师，技术总监，项目经理

所有的代码跟着一点一点写一下

#### 42\_在库存服务中实现缓存与数据库双写一致性保障方案（三）

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

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

(queueNum - 1) & hash

1、线程池+内存队列初始化

@Bean

public ServletListenerRegistrationBean servletListenerRegistrationBean(){

ServletListenerRegistrationBean servletListenerRegistrationBean = new ServletListenerRegistrationBean();

servletListenerRegistrationBean.setListener(new InitListener());

return servletListenerRegistrationBean;

}

java web应用，做系统的初始化，一般在哪里做呢？

ServletContextListener里面做，listener，会跟着整个web应用的启动，就初始化，类似于线程池初始化的构建

spring boot应用，Application，搞一个listener的注册

2、两种请求对象封装

3、请求异步执行Service封装

4、请求处理的工作线程封装

5、两种请求Controller接口封装

6、读请求去重优化

7、空数据读请求过滤优化

队列

对一个商品的库存的数据库更新操作已经在内存队列中了

然后对这个商品的库存的读取操作，要求读取数据库的库存数据，然后更新到缓存中，多个读

这多个读，其实只要有一个读请求操作压到队列里就可以了

其他的读操作，全部都wait那个读请求的操作，刷新缓存，就可以读到缓存中的最新数据了

如果读请求发现redis缓存中没有数据，就会发送读请求给库存服务，但是此时缓存中为空，可能是因为写请求先删除了缓存，也可能是数据库里压根儿没这条数据

如果是数据库中压根儿没这条数据的场景，那么就不应该将读请求操作给压入队列中，而是直接返回空就可以了

都是为了减少内存队列中的请求积压，内存队列中积压的请求越多，就可能导致每个读请求hang住的时间越长，也可能导致多个读请求被hang住

这边跟大家提前说一下，打个招呼

我的风格，跟其他一些人不太一样，我都是课上纯实时手写代码的，没有说提前自己先练个10遍（也没那时间），也不是说预先写好，然后放ppt，课程里就copy粘贴

我就是思路梳理出来，然后边讲课边写代码

就是跟真实的开发的时候是一样的，可能会犯各种各样的错误，可能会写出来bug

有可能有些错误，或者是疏忽的地方，自己课程上就发现了，然后就纠正了; 也可能我没发现自己一些遗漏的地方，QQ+我，然后跟我反馈

视频随时可以补充进去，纠错的一些东西

talk is cheap, show me the code

写代码，还有一个问题，我们是课程上现场写代码，第一遍写代码，写出来的代码，面向对象的设计，设计模式的运用，然后代码结构的设置，都不是太完美，甚至可能是有点粗糙

公司里做项目，其实那个代码会改来改去，重构，代码会变得越来越规整

架构还是得落地，落地还是要写代码的，不管大家是什么层级的人，开发，高工，架构师，技术总监，项目经理

所有的代码跟着一点一点写一下

#### 43\_在库存服务中实现缓存与数据库双写一致性保障方案（四）

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

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

(queueNum - 1) & hash

1、线程池+内存队列初始化

@Bean

public ServletListenerRegistrationBean servletListenerRegistrationBean(){

ServletListenerRegistrationBean servletListenerRegistrationBean = new ServletListenerRegistrationBean();

servletListenerRegistrationBean.setListener(new InitListener());

return servletListenerRegistrationBean;

}

java web应用，做系统的初始化，一般在哪里做呢？

ServletContextListener里面做，listener，会跟着整个web应用的启动，就初始化，类似于线程池初始化的构建

spring boot应用，Application，搞一个listener的注册

2、两种请求对象封装

3、请求异步执行Service封装

4、请求处理的工作线程封装

5、两种请求Controller接口封装

6、读请求去重优化

如果一个读请求过来，发现前面已经有一个写请求和一个读请求了，那么这个读请求就不需要压入队列中了

因为那个写请求肯定会更新数据库，然后那个读请求肯定会从数据库中读取最新数据，然后刷新到缓存中，自己只要hang一会儿就可以从缓存中读到数据了

7、空数据读请求过滤优化

可能某个数据，在数据库里面压根儿就没有，那么那个读请求是不需要放入内存队列的，而且读请求在controller那一层，直接就可以返回了，不需要等待

如果数据库里都没有，就说明，内存队列里面如果没有数据库更新的请求的话，一个读请求过来了，就可以认为是数据库里就压根儿没有数据吧

如果缓存里没数据，就两个情况，第一个是数据库里就没数据，缓存肯定也没数据; 第二个是数据库更新操作过来了，先删除了缓存，此时缓存是空的，但是数据库里是有的

但是的话呢，我们做了之前的读请求去重优化，用了一个flag map，只要前面有数据库更新操作，flag就肯定是存在的，你只不过可以根据true或false，判断你前面执行的是写请求还是读请求

但是如果flag压根儿就没有呢，就说明这个数据，无论是写请求，还是读请求，都没有过

那这个时候过来的读请求，发现flag是null，就可以认为数据库里肯定也是空的，那就不会去读取了

或者说，我们也可以认为每个商品有一个最最初始的库存，但是因为最初始的库存肯定会同步到缓存中去的，有一种特殊的情况，就是说，商品库存本来在redis中是有缓存的

但是因为redis内存满了，就给干掉了，但是此时数据库中是有值得

那么在这种情况下，可能就是之前没有任何的写请求和读请求的flag的值，此时还是需要从数据库中重新加载一次数据到缓存中的

8、深入的去思考优化代码的漏洞

我的一些思考，如果大家发现了其他的漏洞，随时+我Q跟我交流一下

一个读请求过来，将数据库中的数刷新到了缓存中，flag是false，然后过了一会儿，redis内存满了，自动删除了这个额缓存

下一次读请求再过来，发现flag是false，就不会去执行刷新缓存的操作了

而是hang在哪里，反复循环，等一会儿，发现在缓存中始终查询不到数据，然后就去数据库里查询，就直接返回了

这种代码，就有可能会导致，缓存永远变成null的情况

最简单的一种，就是在controller这一块，如果在数据库中查询到了，就刷新到缓存里面去，以后的读请求就又可以从缓存里面读了

队列

对一个商品的库存的数据库更新操作已经在内存队列中了

然后对这个商品的库存的读取操作，要求读取数据库的库存数据，然后更新到缓存中，多个读

这多个读，其实只要有一个读请求操作压到队列里就可以了

其他的读操作，全部都wait那个读请求的操作，刷新缓存，就可以读到缓存中的最新数据了

如果读请求发现redis缓存中没有数据，就会发送读请求给库存服务，但是此时缓存中为空，可能是因为写请求先删除了缓存，也可能是数据库里压根儿没这条数据

如果是数据库中压根儿没这条数据的场景，那么就不应该将读请求操作给压入队列中，而是直接返回空就可以了

都是为了减少内存队列中的请求积压，内存队列中积压的请求越多，就可能导致每个读请求hang住的时间越长，也可能导致多个读请求被hang住

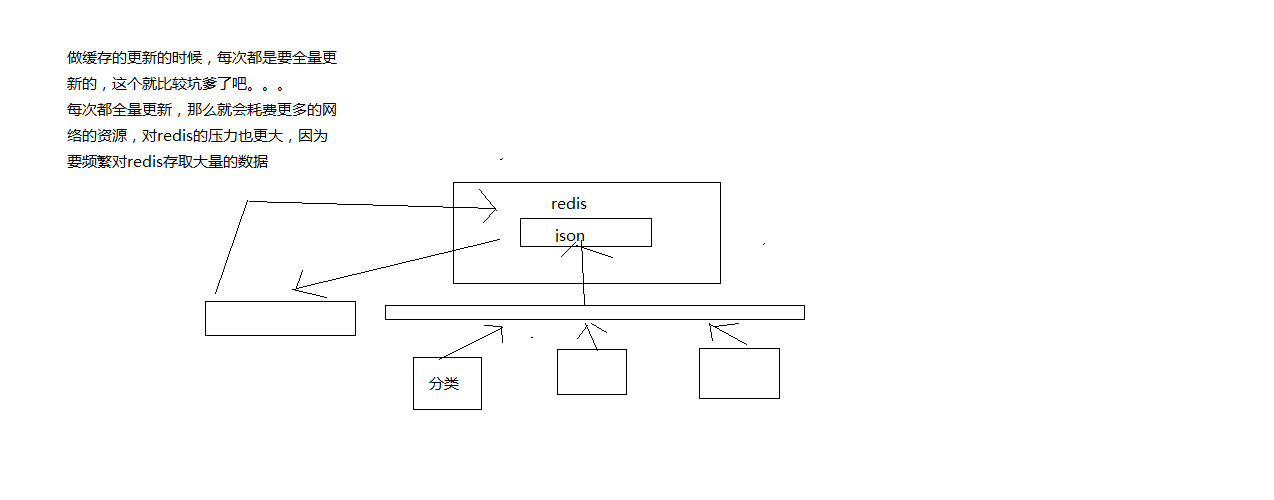
talk is cheap, show me the code

#### 44\_库存服务代码调试以及打印日志观察服务的运行流程是否正确

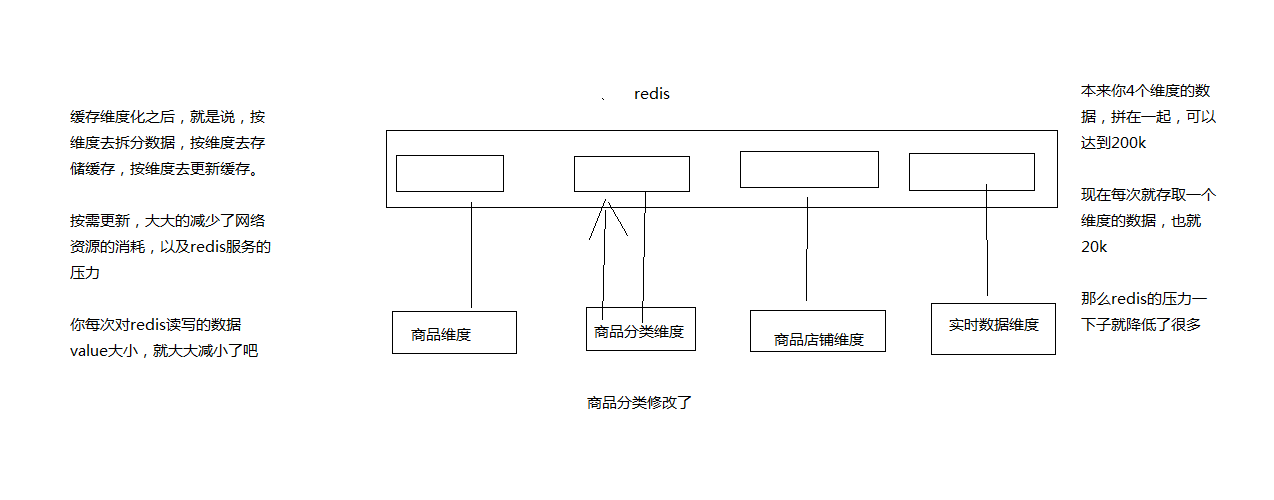
无

#### 45\_商品详情页结构分析、缓存全量更新问题以及缓存维度化解决方案

缓存全量更新的问题



缓存维度化



我们讲解过，咱们的整个缓存的技术方案，分成两块

第一块，是做实时性比较高的那块数据，比如说库存，销量之类的这种数据，我们采取的实时的缓存+数据库双写的技术方案，双写一致性保障的方案

第二块，是做实时性要求不高的数据，比如说商品的基本信息，等等，我们采取的是三级缓存架构的技术方案，就是说由一个专门的数据生产的服务，去获取整个商品详情页需要的各种数据，经过处理后，将数据放入各级缓存中，每一级缓存都有自己的作用

我们先来看看一下，所谓的这种实时性要求不高的数据，在商品详情页中，都有哪些

1、大型电商网站中的商品详情页的数据结构分析

商品的基本信息

标题：【限时直降】Apple/苹果 iPhone 7 128G 全网通4G智能手机正品

短描述：限时优惠 原封国行 正品保障

颜色：

存储容量

图片列表

规格参数

其他信息：店铺信息，分类信息，等等，非商品维度的信息

商品介绍：放缓存，看一点，ajax异步从缓存加载一点，不放我们这里讲解

实时信息：实时广告推荐、实时价格、实时活动推送，等等，ajax加载

我们不是带着大家用几十讲的时间去做一套完整的商品详情页的系统，电商网站的话，都几百个人做好几年的

将商品的各种基本信息，分类放到缓存中，每次请求过来，动态从缓存中取数据，然后动态渲染到模板中

数据放缓存，性能高，动态渲染模板，灵活性好

2、大型缓存全量更新问题

（1）网络耗费的资源大

（2）每次对redis都存取大数据，对redis的压力也比较大

（3）大家记不记得，之前我给大家提过，redis的性能和吞吐量能够支撑到多大，基本跟数据本身的大小有很大的关系

如果数据越大，那么可能导致redis的吞吐量就会急剧下降

3、缓存维度化解决方案

维度：商品维度，商品分类维度，商品店铺维度

不同的维度，可以看做是不同的角度去观察一个东西，那么每个商品详情页中，都包含了不同的维度数据

我就跟大家举个例子，如果不维度化，就导致多个维度的数据混合在一个缓存value中

但是不同维度的数据，可能更新的频率都大不一样

比如说，现在只是将1000个商品的分类批量调整了一下，但是如果商品分类的数据和商品本身的数据混杂在一起

那么可能导致需要将包括商品在内的大缓存value取出来，进行更新，再写回去，就会很坑爹，耗费大量的资源，redis压力也很大

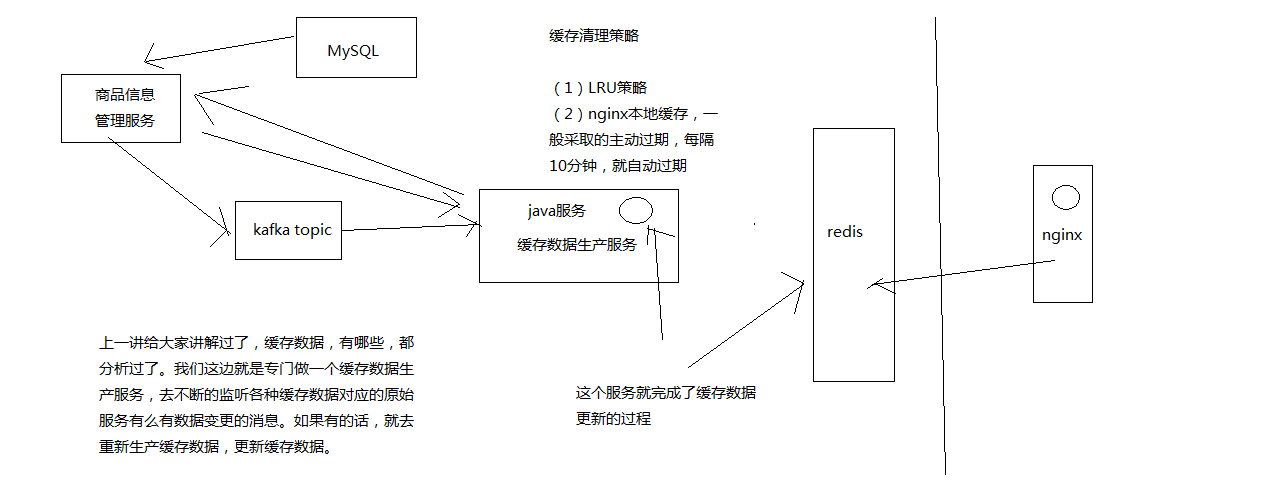
但是如果我们队对缓存进行围堵维度化

唯独化：将每个维度的数据都存一份，比如说商品维度的数据存一份，商品分类的数据存一份，商品店铺的数据存一份

那么在不同的维度数据更新的时候，只要去更新对应的维度就可以了

包括我们之前讲解的那种实时性较高的数据，也可以理解为一个维度，那么维度拆分后

#### 46\_缓存数据生产服务的工作流程分析以及工程环境搭建



接下来要做这个多级缓存架构，从底往上做，先做缓存数据的生产这一块

我们画图来简单介绍一下整个缓存数据生产服务的一个工作流程

1、商品详情页缓存数据生产服务的工作流程分析

（1）监听多个kafka topic，每个kafka topic对应一个服务（简化一下，监听一个kafka topic）

（2）如果一个服务发生了数据变更，那么就发送一个消息到kafka topic中

（3）缓存数据生产服务监听到了消息以后，就发送请求到对应的服务中调用接口以及拉取数据，此时是从mysql中查询的

（4）缓存数据生产服务拉取到了数据之后，会将数据在本地缓存中写入一份，就是ehcache中

（5）同时会将数据在redis中写入一份

2、spring boot+mybatis+redis框架整合搭建

（1）依赖

<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>

（2）Application

@EnableAutoConfiguration

@SpringBootApplication

@ComponentScan

@MapperScan("com.roncoo.eshop.inventory.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);

}

}

（3）resources/Application.properties

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

（4）resources/mybatis

（5）整合Jedis Cluster

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

Application

@Bean

public JedisCluster JedisClusterFactory() {

Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();

jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7003));

jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7004));

jedisClusterNodes.add(new HostAndPort("192.168.31.227", 7006));

JedisCluster jedisCluster = new JedisCluster(jedisClusterNodes);

return jedisCluster;

}

#### 47\_完成spring boot整合ehcache的搭建以支持服务本地堆缓存

因为之前跟大家提过，三级缓存，多级缓存，服务本地堆缓存 + redis分布式缓存 + nginx本地缓存组成的

每一层缓存在高并发的场景下，都有其特殊的用途，需要综合利用多级的缓存，才能支撑住高并发场景下各种各样的特殊情况

服务本地堆缓存，作用，预防redis层的彻底崩溃，作为缓存的最后一道防线，避免数据库直接裸奔

服务本地堆缓存，我们用什么来做缓存，难道我们自己手写一个类或者程序去管理内存吗？？？java最流行的缓存的框架，ehcache

所以我们也是用ehcache来做本地的堆缓存

spring boot + ehcache整合起来，演示一下是怎么使用的

spring boot整合ehcache

（1）依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

</dependency>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache</artifactId>

<version>2.8.3</version>

</dependency>

（2）缓存配置管理类

@Configuration

@EnableCaching

public class CacheConfiguration {

@Bean

public EhCacheManagerFactoryBean ehCacheManagerFactoryBean(){

EhCacheManagerFactoryBean cacheManagerFactoryBean = new EhCacheManagerFactoryBean();

cacheManagerFactoryBean.setConfigLocation(new ClassPathResource("ehcache.xml"));

cacheManagerFactoryBean.setShared(true);

return cacheManagerFactoryBean;

}

@Bean

public EhCacheCacheManager ehCacheCacheManager(EhCacheManagerFactoryBean bean){

return new EhCacheCacheManager(bean.getObject());

}

}

（3）ehcache.xml

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

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

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

updateCheck="false">

<diskStore path="java.io.tmpdir/Tmp\_EhCache" />

<defaultCache

eternal="false"

maxElementsInMemory="1000"

overflowToDisk="false"

diskPersistent="false"

timeToIdleSeconds="0"

timeToLiveSeconds="0"

memoryStoreEvictionPolicy="LRU" />

<cache

name="local"

eternal="false"

maxElementsInMemory="1000"

overflowToDisk="false"

diskPersistent="false"

timeToIdleSeconds="0"

timeToLiveSeconds="0"

memoryStoreEvictionPolicy="LRU" />

</ehcache>

（4）CacheService

@Service("cacheService")

public class CacheServiceImpl implements CacheService {

public static final String CACHE\_NAME = "local";

@Cacheable(value = CACHE\_NAME, key = "'key\_'+#id")

public ProductInfo findById(Long id){

return null;

}

@CachePut(value = CACHE\_NAME, key = "'key\_'+#productInfo.getId()")

public ProductInfo saveProductInfo(ProductInfo productInfo) {

return productInfo;

}

}

（5）写一个Controller测试一下ehcache的整合

@Controller

public class CacheTestController {

@Resource

private CacheService cacheService;

@RequestMapping("/testPutCache")

@ResponseBody

public void testPutCache(ProductInfo productInfo) {

System.out.println(productInfo.getId() + ":" + productInfo.getName());

cacheService.saveProductInfo(productInfo);

}

@RequestMapping("/testGetCache")

@ResponseBody

public ProductInfo testGetCache(Long id) {

ProductInfo productInfo = cacheService.findById(id);

System.out.println(productInfo.getId() + ":" + productInfo.getName());

return productInfo;

}

}

ehcache已经整合进了我们的系统，spring boot

封装好了对ehcache本地缓存进行添加和获取的方法和service

#### 48\_redis的LRU缓存清除算法讲解以及相关配置使用

之前给大家讲解过，多级缓存架构，缓存数据生产服务，监听各个数据源服务的数据变更的消息，得到消息之后，然后调用接口拉去数据

将拉去到的数据，写入本地ehcache缓存一份，spring boot整合，演示过

数据写入redis分布式缓存中一份，你不断的将数据写入redis，写入redis，然后redis的内存是有限的，每个redis实例最大一般也就是设置给10G

那如果你不断的写入数据，当数据写入的量超过了redis能承受的范围之后，改该怎么玩儿呢？？？

redis是会在数据达到一定程度之后，超过了一个最大的限度之后，就会将数据进行一定的清理，从内存中清理掉一些数据

只有清理掉一些数据之后，才能将新的数据写入内存中

1、LRU算法概述

redis默认情况下就是使用LRU策略的，因为内存是有限的，但是如果你不断地往redis里面写入数据，那肯定是没法存放下所有的数据在内存的

所以redis默认情况下，当内存中写入的数据很满之后，就会使用LRU算法清理掉部分内存中的数据，腾出一些空间来，然后让新的数据写入redis缓存中

LRU：Least Recently Used，最近最少使用算法

将最近一段时间内，最少使用的一些数据，给干掉。比如说有一个key，在最近1个小时内，只被访问了一次; 还有一个key在最近1个小时内，被访问了1万次

这个时候比如你要将部分数据给清理掉，你会选择清理哪些数据啊？肯定是那个在最近小时内被访问了1万次的数据

2、缓存清理设置

redis.conf

maxmemory，设置redis用来存放数据的最大的内存大小，一旦超出这个内存大小之后，就会立即使用LRU算法清理掉部分数据

如果用LRU，那么就是将最近最少使用的数据从缓存中清除出去

对于64 bit的机器，如果maxmemory设置为0，那么就默认不限制内存的使用，直到耗尽机器中所有的内存为止; 但是对于32 bit的机器，有一个隐式的闲置就是3GB

maxmemory-policy，可以设置内存达到最大闲置后，采取什么策略来处理

（1）noeviction: 如果内存使用达到了maxmemory，client还要继续写入数据，那么就直接报错给客户端

（2）allkeys-lru: 就是我们常说的LRU算法，移除掉最近最少使用的那些keys对应的数据

（3）volatile-lru: 也是采取LRU算法，但是仅仅针对那些设置了指定存活时间（TTL）的key才会清理掉

（4）allkeys-random: 随机选择一些key来删除掉

（5）volatile-random: 随机选择一些设置了TTL的key来删除掉

（6）volatile-ttl: 移除掉部分keys，选择那些TTL时间比较短的keys

在redis里面，写入key-value对的时候，是可以设置TTL，存活时间，比如你设置了60s。那么一个key-value对，在60s之后就会自动被删除

redis的使用，各种数据结构，list，set，等等

allkeys-lru

这边拓展一下思路，对技术的研究，一旦将一些技术研究的比较透彻之后，就喜欢横向对比底层的一些原理

storm，科普一下

玩儿大数据的人搞得，领域，实时计算领域，storm

storm有很多的流分组的一些策略，按shuffle分组，global全局分组，direct直接分组，fields按字段值hash后分组

分组策略也很多，但是，真正公司里99%的场景下，使用的也就是shuffle和fields，两种策略

redis，给了这么多种乱七八糟的缓存清理的算法，其实真正常用的可能也就那么一两种，allkeys-lru是最常用的

3、缓存清理的流程

（1）客户端执行数据写入操作

（2）redis server接收到写入操作之后，检查maxmemory的限制，如果超过了限制，那么就根据对应的policy清理掉部分数据

（3）写入操作完成执行

4、redis的LRU近似算法

科普一个相对来说稍微高级一丢丢的知识点

redis采取的是LRU近似算法，也就是对keys进行采样，然后在采样结果中进行数据清理

redis 3.0开始，在LRU近似算法中引入了pool机制，表现可以跟真正的LRU算法相当，但是还是有所差距的，不过这样可以减少内存的消耗

redis LRU算法，是采样之后再做LRU清理的，跟真正的、传统、全量的LRU算法是不太一样的

maxmemory-samples，比如5，可以设置采样的大小，如果设置为10，那么效果会更好，不过也会耗费更多的CPU资源

#### 49\_zookeeper+kafka集群的安装部署以及如何简单使用的介绍

多级缓存的架构

主要是用来解决什么样的数据的缓存的更新的啊？？？

时效性不高的数据，比如一些商品的基本信息，如果发生了变更，假设在5分钟之后再更新到页面中，供用户观察到，也是ok的

时效性要求不高的数据，那么我们采取的是异步更新缓存的策略

时效性要求很高的数据，库存，采取的是数据库+缓存双写的技术方案，也解决了双写的一致性的问题

缓存数据生产服务，监听一个消息队列，然后数据源服务（商品信息管理服务）发生了数据变更之后，就将数据变更的消息推送到消息队列中

缓存数据生产服务可以去消费到这个数据变更的消息，然后根据消息的指示提取一些参数，然后调用对应的数据源服务的接口，拉去数据，这个时候一般是从mysql库中拉去的

消息队列是什么东西？采取打的就是kafka

我工作的时候，很多项目是跟大数据相关的，当然也有很多是纯java系统的架构，最近用kafka用得比较多

kafka比较简单易用，讲课来说，很方便

解释一下，我们当然是不可能对课程中涉及的各种技术都深入浅出的讲解的了，kafka，花上20个小时给你讲解一下，不可能的

所以说呢，在这里，一些技术的组合，用什么都ok

笑傲江湖中的风清扬，手中无剑胜有剑，还有任何东西都可以当做兵器，哪怕是一根草也可以

搞技术，kafka和activemq肯定有区别，但是说，在有些场景下，其实可能没有那么大的区分度，kafka和activemq其实是一样的

生产者+消费者的场景，kafka+activemq都ok

涉及的这种架构，对时效性要求高和时效性要求低的数据，分别采取什么技术方案？数据库+缓存双写一致性？异步+多级缓存架构？大缓存的维度化拆分？

你要关注的，是一些架构上的东西和思想，而不是具体的什么mq的使用

activemq的课程，书籍，资料

kafka集群，zookeeper集群，先搭建zookeeper集群，再搭建kafka集群

kafka另外一个原因：kafka，本来就要搭建zookeeper，zookeeper这个东西，后面我们还要用呢，缓存的分布式并发更新的问题，分布式锁解决

zookeeper + kafka的集群，都是三节点

java高级工程师的思想，在干活儿，在思考，jvm，宏观的思考，通盘去考虑整个架构，还有未来的技术规划，业务的发展方向，架构的演进方向和路线

把课程里讲解的各种技术方案组合成、修改成你需要的适合你的业务的缓存架构

1、zookeeper集群搭建

将课程提供的zookeeper-3.4.5.tar.gz使用WinSCP拷贝到/usr/local目录下。

对zookeeper-3.4.5.tar.gz进行解压缩：tar -zxvf zookeeper-3.4.5.tar.gz。

对zookeeper目录进行重命名：mv zookeeper-3.4.5 zk

配置zookeeper相关的环境变量

vi ~/.bashrc

export ZOOKEEPER\_HOME=/usr/local/zk

export PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin:$ZOOKEEPER\_HOME/bin

source ~/.bashrc

cd zk/conf

cp zoo\_sample.cfg zoo.cfg

vi zoo.cfg

修改：dataDir=/usr/local/zk/data

新增：

server.0=eshop-cache01:2888:3888

server.1=eshop-cache02:2888:3888

server.2=eshop-cache03:2888:3888

cd zk

mkdir data

cd data

vi myid

0

在另外两个节点上按照上述步骤配置ZooKeeper，使用scp将zk和.bashrc拷贝到eshop-cache02和eshop-cache03上即可。唯一的区别是标识号分别设置为1和2。

**scp -r zk root@192.168.60.5:/usr/local/**

分别在三台机器上执行：zkServer.sh start。

检查ZooKeeper状态：zkServer.sh status，应该是一个leader，两个follower

jps：检查三个节点是否都有QuromPeerMain进程

2、kafka集群搭建

scala，我就不想多说了，就是一门编程语言，现在比较火，很多比如大数据领域里面的spark（计算引擎）就是用scala编写的

将课程提供的scala-2.11.4.tgz使用WinSCP拷贝到/usr/local目录下。

对scala-2.11.4.tgz进行解压缩：tar -zxvf scala-2.11.4.tgz。

对scala目录进行重命名：mv scala-2.11.4 scala

配置scala相关的环境变量

vi ~/.bashrc

export SCALA\_HOME=/usr/local/scala

export PATH=$SCALA\_HOME/bin

source ~/.bashrc

查看scala是否安装成功：scala -version

按照上述步骤在其他机器上都安装好scala。使用scp将scala和.bashrc拷贝到另外两台机器上即可。

**scp -r scala root@192.168.60.6:/usr/local/**

将课程提供的kafka\_2.9.2-0.8.1.tgz使用WinSCP拷贝到/usr/local目录下。

对kafka\_2.9.2-0.8.1.tgz进行解压缩：tar -zxvf kafka\_2.9.2-0.8.1.tgz。

对kafka目录进行改名：mv kafka\_2.9.2-0.8.1 kafka

配置kafka

vi /usr/local/kafka/config/server.properties

broker.id：依次增长的整数，0、1、2，集群中Broker的唯一id

zookeeper.connect=192.168.60.5:2180,192.168.60.6:2180,192.168.60.7:2180

安装slf4j

将课程提供的slf4j-1.7.6.zip上传到/usr/local目录下

unzip slf4j-1.7.6.zip

把slf4j中的slf4j-nop-1.7.6.jar复制到kafka的libs目录下面

解决kafka Unrecognized VM option 'UseCompressedOops'问题

vi /usr/local/kafka/bin/kafka-run-class.sh

if [ -z "$KAFKA\_JVM\_PERFORMANCE\_OPTS" ]; then

KAFKA\_JVM\_PERFORMANCE\_OPTS="-server -XX:+UseCompressedOops -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC -Djava.awt.headless=true"

fi

去掉-XX:+UseCompressedOops即可

按照上述步骤在另外两台机器分别安装kafka。用scp把kafka拷贝到其他机器即可。

唯一区别的，就是server.properties中的broker.id，要设置为1和2

**scp -r** kafka **root@192.168.60.7:/usr/local/**

在三台机器上的kafka目录下，分别执行以下命令：nohup bin/kafka-server-start.sh config/server.properties &

使用jps检查启动是否成功

使用基本命令检查kafka是否搭建成功

bin/kafka-topics.sh --zookeeper 192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181 --topic test --replication-factor 1 --partitions 1 --create

bin/kafka-console-producer.sh --broker-list 192.168.31.181:9092,192.168.31.19:9092,192.168.31.227:9092 --topic test

bin/kafka-console-consumer.sh --zookeeper 192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181 --topic test --from-beginning

#### 50\_基于kafka+ehcache+redis完成缓存数据生产服务的开发与测试

1、将kafka整合到spring boot中

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka\_2.9.2</artifactId>

<version>0.8.1</version>

</dependency>

@Bean

public ServletListenerRegistrationBean servletListenerRegistrationBean() {

ServletListenerRegistrationBean servletListenerRegistrationBean =

new ServletListenerRegistrationBean();

servletListenerRegistrationBean.setListener(new InitListener());

return servletListenerRegistrationBean;

}

public class KafkaConcusmer implements Runnable {

private final ConsumerConnector consumer;

private final String topic;

public ConsumerGroupExample(String topic) {

consumer = Consumer.createJavaConsumerConnector(createConsumerConfig());

this.topic = topic;

}

private static ConsumerConfig createConsumerConfig(String a\_zookeeper, String a\_groupId) {

Properties props = new Properties();

props.put("zookeeper.connect", "192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181");

props.put("group.id", "eshop-cache-group");

props.put("zookeeper.session.timeout.ms", "400");

props.put("zookeeper.sync.time.ms", "200");

props.put("auto.commit.interval.ms", "1000");

return new ConsumerConfig(props);

}

public void run() {

Map<String, Integer> topicCountMap = new HashMap<String, Integer>();

topicCountMap.put(topic, 1);

Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap);

List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

for (final KafkaStream stream : streams) {

new Thread(new KafkaMessageProcessor(stream)).start();

}

}

}

public class KafkaMessageProcessor implements Runnable {

private KafkaStream kafkaStream;

public ConsumerTest(KafkaStream kafkaStream) {

this.kafkaStream = kafkaStream;

}

public void run() {

ConsumerIterator<byte[], byte[]> it = kafkaStream.iterator();

while (it.hasNext()) {

String message = new String(it.next().message());

}

}

}

ServletContext sc = sce.getServletContext();

ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(sc);

2、编写业务逻辑

（1）两种服务会发送来数据变更消息：商品信息服务，商品店铺信息服务，每个消息都包含服务名以及商品id

（2）接收到消息之后，根据商品id到对应的服务拉取数据，这一步，我们采取简化的模拟方式，就是在代码里面写死，会获取到什么数据，不去实际再写其他的服务去调用了

（3）商品信息：id，名称，价格，图片列表，商品规格，售后信息，颜色，尺寸

（4）商品店铺信息：其他维度，用这个维度模拟出来缓存数据维度化拆分，id，店铺名称，店铺等级，店铺好评率

（5）分别拉取到了数据之后，将数据组织成json串，然后分别存储到ehcache中，和redis缓存中

3、测试业务逻辑

（1）创建一个kafka topic

（2）在命令行启动一个kafka producer

（3）启动系统，消费者开始监听kafka topic

C:\Windows\System32\drivers\etc\hosts

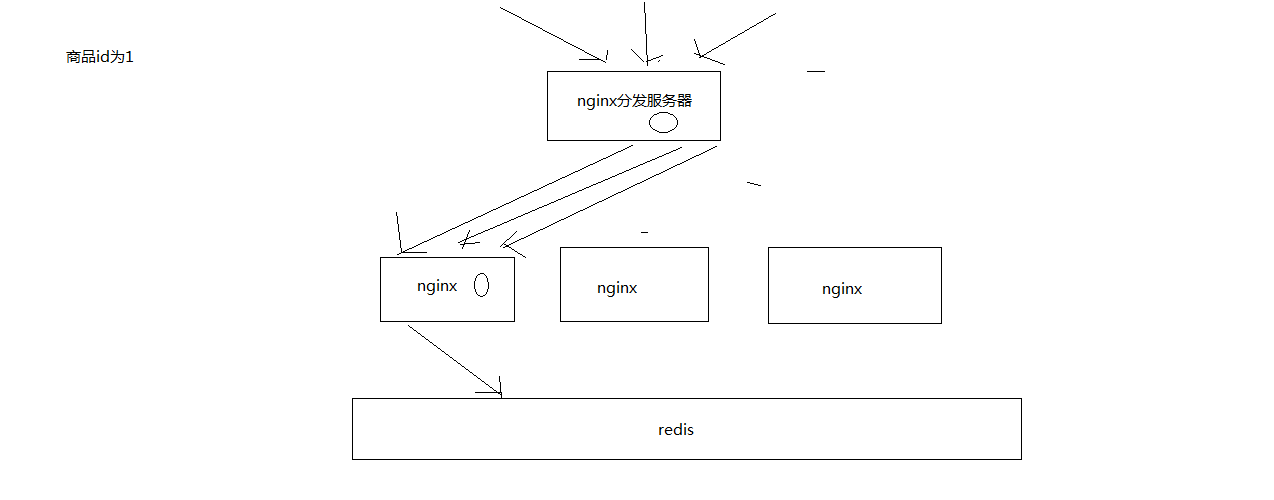
（4）在producer中，分别发送两条消息，一个是商品信息服务的消息，一个是商品店铺信息服务的消息

（5）能否接收到两条消息，并模拟拉取到两条数据，同时将数据写入ehcache中，并写入redis缓存中

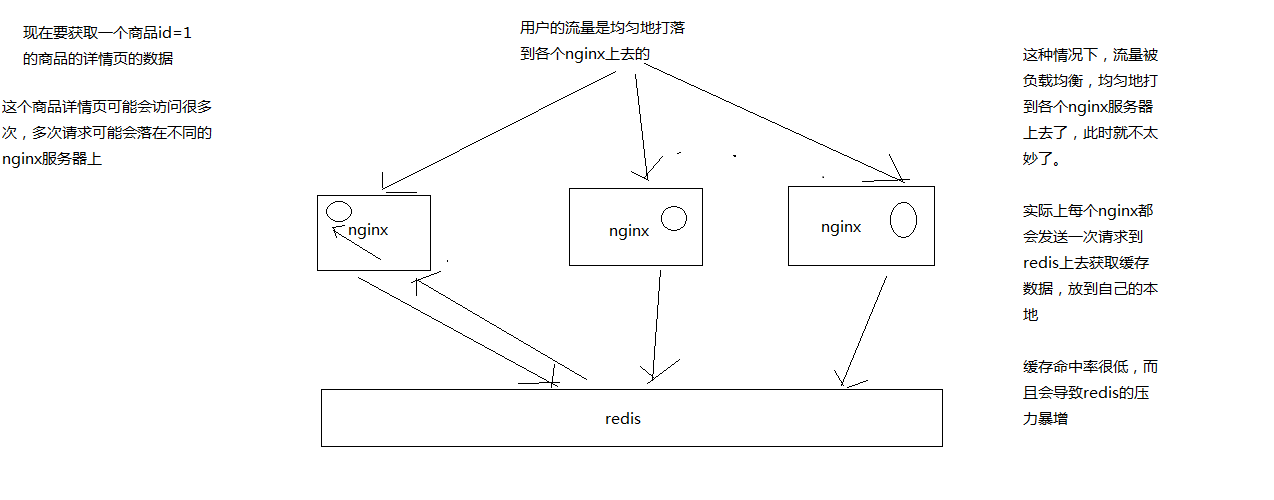
（6）ehcache通过打印日志方式来观察，redis通过手工连接上去来查询

#### 51\_基于“分发层+应用层”双层nginx架构提升缓存命中率方案分析

分发层+应用层双层nginx架构



缓存命中率低的原因



1、缓存命中率低

缓存数据生产服务那一层已经搞定了，相当于三层缓存架构中的本地堆缓存+redis分布式缓存都搞定了

就要来做三级缓存中的nginx那一层的缓存了

如果一般来说，你默认会部署多个nginx，在里面都会放一些缓存，就默认情况下，此时缓存命中率是比较低的

2、如何提升缓存命中率

分发层+应用层，双层nginx

分发层nginx，负责流量分发的逻辑和策略，这个里面它可以根据你自己定义的一些规则，比如根据productId去进行hash，然后对后端的nginx数量取模

将某一个商品的访问的请求，就固定路由到一个nginx后端服务器上去，保证说只会从redis中获取一次缓存数据，后面全都是走nginx本地缓存了

后端的nginx服务器，就称之为应用服务器; 最前端的nginx服务器，被称之为分发服务器

看似很简单，其实很有用，在实际的生产环境中，可以大幅度提升你的nginx本地缓存这一层的命中率，大幅度减少redis后端的压力，提升性能

#### 52\_基于OpenResty部署应用层nginx以及nginx+lua开发hello world

我们这里玩儿nginx，全都会在nginx里去写lua脚本，因为我们需要自定义一些特殊的业务逻辑

比如说，流量分发，自己用lua去写分发的逻辑，在分发层nginx里去写的

再比如说，要用lua去写多级缓存架构存取的控制逻辑，在应用层nginx里去写的

后面还要做热点数据的自动降级机制，也是用lua脚本去写降级机制的，在分发层nginx里去写的

因为我们要用nginx+lua去开发，所以会选择用最流行的开源方案，就是用OpenResty

nginx+lua打包在一起，而且提供了包括redis客户端，mysql客户端，http客户端在内的大量的组件

我们这一讲是去部署应用层nginx，会采用OpenResty的方式去部署nginx，而且会带着大家写一个nginx+lua开发的一个hello world

1、部署第一个nginx，作为应用层nginx（192.168.31.187那个机器上）

（1）部署openresty

mkdir -p /usr/servers

cd /usr/servers/

yum install -y readline-devel pcre-devel openssl-devel gcc

wget http://openresty.org/download/ngx\_openresty-1.7.7.2.tar.gz

tar -xzvf ngx\_openresty-1.7.7.2.tar.gz

cd /usr/servers/ngx\_openresty-1.7.7.2/

cd bundle/LuaJIT-2.1-20150120/

make clean && make && make install

ln -sf luajit-2.1.0-alpha /usr/local/bin/luajit

cd bundle

wget https://github.com/FRiCKLE/ngx\_cache\_purge/archive/2.3.tar.gz

tar -xvf 2.3.tar.gz

cd bundle

wget https://github.com/yaoweibin/nginx\_upstream\_check\_module/archive/v0.3.0.tar.gz

tar -xvf v0.3.0.tar.gz

cd /usr/servers/ngx\_openresty-1.7.7.2

./configure --prefix=/usr/servers --with-http\_realip\_module --with-pcre --with-luajit --add-module=./bundle/ngx\_cache\_purge-2.3/ --add-module=./bundle/nginx\_upstream\_check\_module-0.3.0/ -j2

make && make install

cd /usr/servers/

ll

/usr/servers/luajit

/usr/servers/lualib

/usr/servers/nginx

/usr/servers/nginx/sbin/nginx -V

启动nginx: /usr/servers/nginx/sbin/nginx

（2）nginx+lua开发的hello world

vi /usr/servers/nginx/conf/nginx.conf

在http部分添加：

lua\_package\_path "/usr/servers/lualib/?.lua;;";

lua\_package\_cpath "/usr/servers/lualib/?.so;;";

/usr/servers/nginx/conf下，创建一个lua.conf

server {

listen 80;

server\_name \_;

}

在nginx.conf的http部分添加：

include lua.conf;

验证配置是否正确：

/usr/servers/nginx/sbin/nginx -t

在lua.conf的server部分添加：

location /lua {

default\_type 'text/html';

content\_by\_lua 'ngx.say("hello world")';

}

/usr/servers/nginx/sbin/nginx -t

重新nginx加载配置

/usr/servers/nginx/sbin/nginx -s reload

访问http: http://192.168.31.187/lua

vi /usr/servers/nginx/conf/lua/test.lua

ngx.say("hello world");

修改lua.conf

location /lua {

default\_type 'text/html';

content\_by\_lua\_file conf/lua/test.lua;

}

查看异常日志

tail -f /usr/servers/nginx/logs/error.log

（3）工程化的nginx+lua项目结构

项目工程结构

hello

hello.conf

lua

hello.lua

lualib

\*.lua

\*.so

放在/usr/hello目录下

/usr/servers/nginx/conf/nginx.conf

worker\_processes 2;

error\_log logs/error.log;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type text/html;

lua\_package\_path "/usr/hello/lualib/?.lua;;";

lua\_package\_cpath "/usr/hello/lualib/?.so;;";

include /usr/hello/hello.conf;

}

/usr/hello/hello.conf

server {

listen 80;

server\_name \_;

location /lua {

default\_type 'text/html';

lua\_code\_cache off;

content\_by\_lua\_file /usr/example/lua/test.lua;

}

}

2、如法炮制，在另外一个机器上，也用OpenResty部署一个nginx

大家可能会发现说，有的人可能对nginx也不是太懂，对lua也没什么了解

对于我的课程来说，主要还是关注，关注的是我们核心的topic，缓存，缓存，缓存，缓存的各种解决方案，大型缓存的架构

那么对于课程里涉及到的各种技术来说，比如nginx，lua脚本

你说让我给你讲成nginx从入门到精通，也不太现实; 讲一个lua脚本开发从入门到精通，也不太现实

我只能说，跟着整个项目的思路去走，把项目里涉及的相关技术的知识给你讲解一下，然后保证说，带着你手把手的去做，让你至少可以学会项目里讲解的这些知识，可以做出来

如果你后面真的是要自己去用nginx+lua去做项目，其实个人建议你还是得去查询和学习一些更多的资料，nginx的一些知识，lua的一些语法

龙果，最受欢迎的一套课程，就是dubbo实战课程，里面也是dubbo整合了各种技术，active mq，zookeeper，redis 3.0分布式集群，mysql读写分离

但是有个问题，每个课程，我相信一个好的课程，它总是可以让你学到很多知识的

但是任何一个好的课程，它都不是万能的，dubbo，zookeeper，注册中心，zookeeper分布式锁，分布式协调，分布式选举，等等技术，你学到吗？

dubbo它也不可能说是给你把zookeeper，redis，mysql全部讲解到从入门到精通这样子

topic，主题，基于dubbo复杂的分布式系统的通用架构，分布式系统，dubbo rpc的调用，服务的开发; zookeeper注册中心; redis分布式集群; mysql读写分离; tomcat集群; hudson持续集成

它告诉你的是一个通用的分布式系统的架构

我这里的也会带着你做nginx的部署,openresty，nginx+lua的开发，redis集群/高可用/高并发/读写分离/持久化/数据备份恢复，zookeeper分布式锁，kafka去做消息通信，hystrix去做限流

但是任何一个技术都不可能给你从入门到精通讲解完

大家可以去关注一下我的es的课程，你如果录一套课程，基本选择方向就两个，要不就是讲解技术本身，大量案例实战贯穿，把技术本身讲解的很细致

我之前有两个es顶尖高手系列的课程，技术课程，把es这个技术讲解的非常非常的细致

像我们现在这个课程，大规模缓存，支撑高并发，高性能，海量数据，类似之前dubbo实战课程，它讲解的还是一种架构课程，那么就关注点在整个架构的整体，整合，架构方案，架构设计，架构思想

里面涉及的技术是不可能给你去深入讲解的

#### 53\_部署分发层nginx以及基于lua完成基于商品id的定向流量分发策略

大家可以自己按照上一讲讲解的内容，基于OpenResty在另外两台机器上都部署一下nginx+lua的开发环境

我已经在eshop-cache02和eshop-cache03上都部署好了

我这边的话呢，是打算用eshop-cache01和eshop-cache02作为应用层nginx服务器，用eshop-cache03作为分发层nginx

在eshop-cache03，也就是分发层nginx中，编写lua脚本，完成基于商品id的流量分发策略

当然了，我们这里主要会简化策略，简化业务逻辑，实际上在你的公司中，你可以随意根据自己的业务逻辑和场景，去制定自己的流量分发策略

1、获取请求参数，比如productId

2、对productId进行hash

3、hash值对应用服务器数量取模，获取到一个应用服务器

4、利用http发送请求到应用层nginx

5、获取响应后返回

这个就是基于商品id的定向流量分发的策略，lua脚本来编写和实现

我们作为一个流量分发的nginx，会发送http请求到后端的应用nginx上面去，所以要先引入lua http lib包

cd /usr/hello/lualib/resty/

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http\_headers.lua

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http.lua

代码：

local uri\_args = ngx.req.get\_uri\_args()

local productId = uri\_args["productId"]

local host = {"192.168.31.19", "192.168.31.187"}

local hash = ngx.crc32\_long(productId)

hash = (hash % 2) + 1

backend = "http://"..host[hash]

local method = uri\_args["method"]

local requestBody = "/"..method.."?productId="..productId

local http = require("resty.http")

local httpc = http.new()

local resp, err = httpc:request\_uri(backend, {

method = "GET",

path = requestBody

})

if not resp then

ngx.say("request error :", err)

return

end

ngx.say(resp.body)

httpc:close()

/usr/servers/nginx/sbin/nginx -s reload

基于商品id的定向流量分发策略的lua脚本就开发完了，而且也测试过了

我们就可以看到，如果你请求的是固定的某一个商品，那么就一定会将流量打到固定的一个应用nginx上面去

#### 54\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（一）

分发层nginx，lua应用，会将商品id，商品店铺id，都转发到后端的应用nginx

/usr/servers/nginx/sbin/nginx -s reload

1、应用nginx的lua脚本接收到请求

2、获取请求参数中的商品id，以及商品店铺id

3、根据商品id和商品店铺id，在nginx本地缓存中尝试获取数据

4、如果在nginx本地缓存中没有获取到数据，那么就到redis分布式缓存中获取数据，如果获取到了数据，还要设置到nginx本地缓存中

但是这里有个问题，建议不要用nginx+lua直接去获取redis数据

因为openresty没有太好的redis cluster的支持包，所以建议是发送http请求到缓存数据生产服务，由该服务提供一个http接口

缓存数生产服务可以基于redis cluster api从redis中直接获取数据，并返回给nginx

cd /usr/hello/lualib/resty/

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http\_headers.lua

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http.lua

5、如果缓存数据生产服务没有在redis分布式缓存中没有获取到数据，那么就在自己本地ehcache中获取数据，返回数据给nginx，也要设置到nginx本地缓存中

6、如果ehcache本地缓存都没有数据，那么就需要去原始的服务中拉去数据，该服务会从mysql中查询，拉去到数据之后，返回给nginx，并重新设置到ehcache和redis中

这里先不考虑，后面要专门讲解一套分布式缓存重建并发冲突的问题和解决方案

7、nginx最终利用获取到的数据，动态渲染网页模板

cd /usr/hello/lualib/resty/

wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template.lua

mkdir /usr/hello/lualib/resty/html

cd /usr/hello/lualib/resty/html

wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template/html.lua

在hello.conf的server中配置模板位置

set $template\_location "/templates";

set $template\_root "/usr/hello/templates";

mkdir /usr/hello/templates

vi product.html

product id: {\* productId \*}<br/>

product name: {\* productName \*}<br/>

product picture list: {\* productPictureList \*}<br/>

product specification: {\* productSpecification \*}<br/>

product service: {\* productService \*}<br/>

product color: {\* productColor \*}<br/>

product size: {\* productSize \*}<br/>

shop id: {\* shopId \*}<br/>

shop name: {\* shopName \*}<br/>

shop level: {\* shopLevel \*}<br/>

shop good cooment rate: {\* shopGoodCommentRate \*}<br/>

8、将渲染后的网页模板作为http响应，返回给分发层nginx

hello.conf中：

lua\_shared\_dict my\_cache 128m;

lua脚本中：

local uri\_args = ngx.req.get\_uri\_args()

local productId = uri\_args["productId"]

local shopId = uri\_args["shopId"]

local cache\_ngx = ngx.shared.my\_cache

local productCacheKey = "product\_info\_"..productId

local shopCacheKey = "shop\_info\_"..shopId

local productCache = cache\_ngx:get(productCacheKey)

local shopCache = cache\_ngx:get(shopCacheKey)

if productCache == "" or productCache == nil then

local http = require("resty.http")

local httpc = http.new()

local resp, err = httpc:request\_uri("http://192.168.31.179:8080",{

method = "GET",

path = "/getProductInfo?productId="..productId

})

productCache = resp.body

cache\_ngx:set(productCacheKey, productCache, 10 \* 60)

end

if shopCache == "" or shopCache == nil then

local http = require("resty.http")

local httpc = http.new()

local resp, err = httpc:request\_uri("http://192.168.31.179:8080",{

method = "GET",

path = "/getShopInfo?shopId="..shopId

})

shopCache = resp.body

cache\_ngx:set(shopCacheKey, shopCache, 10 \* 60)

end

local cjson = require("cjson")

local productCacheJSON = cjson.decode(productCache)

local shopCacheJSON = cjson.decode(shopCache)

local context = {

productId = productCacheJSON.id,

productName = productCacheJSON.name,

productPrice = productCacheJSON.price,

productPictureList = productCacheJSON.pictureList,

productSpecification = productCacheJSON.specification,

productService = productCacheJSON.service,

productColor = productCacheJSON.color,

productSize = productCacheJSON.size,

shopId = shopCacheJSON.id,

shopName = shopCacheJSON.name,

shopLevel = shopCacheJSON.level,

shopGoodCommentRate = shopCacheJSON.goodCommentRate

}

local template = require("resty.template")

template.render("product.html", context)

#### 55\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（二）

/usr/servers/nginx/sbin/nginx -s reload

lua\_shared\_dict my\_cache 128m;

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>商品详情页</title>

</head>

<body>

商品id: {\* productId \*}<br/>

商品名称: {\* productName \*}<br/>

商品图片列表: {\* productPictureList \*}<br/>

商品规格: {\* productSpecification \*}<br/>

商品售后服务: {\* productService \*}<br/>

商品颜色: {\* productColor \*}<br/>

商品大小: {\* productSize \*}<br/>

店铺id: {\* shopId \*}<br/>

店铺名称: {\* shopName \*}<br/>

店铺评级: {\* shopLevel \*}<br/>

店铺好评率: {\* shopGoodCommentRate \*}<br/>

</body>

</html>

#### 56\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（三）

/usr/servers/nginx/sbin/nginx -s reload

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>商品详情页</title>

</head>

<body>

商品id: {\* productId \*}<br/>

商品名称: {\* productName \*}<br/>

商品图片列表: {\* productPictureList \*}<br/>

商品规格: {\* productSpecification \*}<br/>

商品售后服务: {\* productService \*}<br/>

商品颜色: {\* productColor \*}<br/>

商品大小: {\* productSize \*}<br/>

店铺id: {\* shopId \*}<br/>

店铺名称: {\* shopName \*}<br/>

店铺等级: {\* shopLevel \*}<br/>

店铺好评率: {\* shopGoodCommentRate \*}<br/>

</body>

</html>

第一次访问的时候，其实在nginx本地缓存中是取不到的，所以会发送http请求到后端的缓存服务里去获取，会从redis中获取

拿到数据以后，会放到nginx本地缓存里面去，过期时间是10分钟

然后将所有数据渲染到模板中，返回模板

以后再来访问的时候，就会直接从nginx本地缓存区获取数据了

缓存数据生产 -> 有数据变更 -> 主动更新两级缓存（ehcache+redis）-> 缓存维度化拆分

分发层nginx + 应用层nginx -> 自定义流量分发策略提高缓存命中率

nginx shared dict缓存 -> 缓存服务 -> redis -> ehcache -> 渲染html模板 -> 返回页面

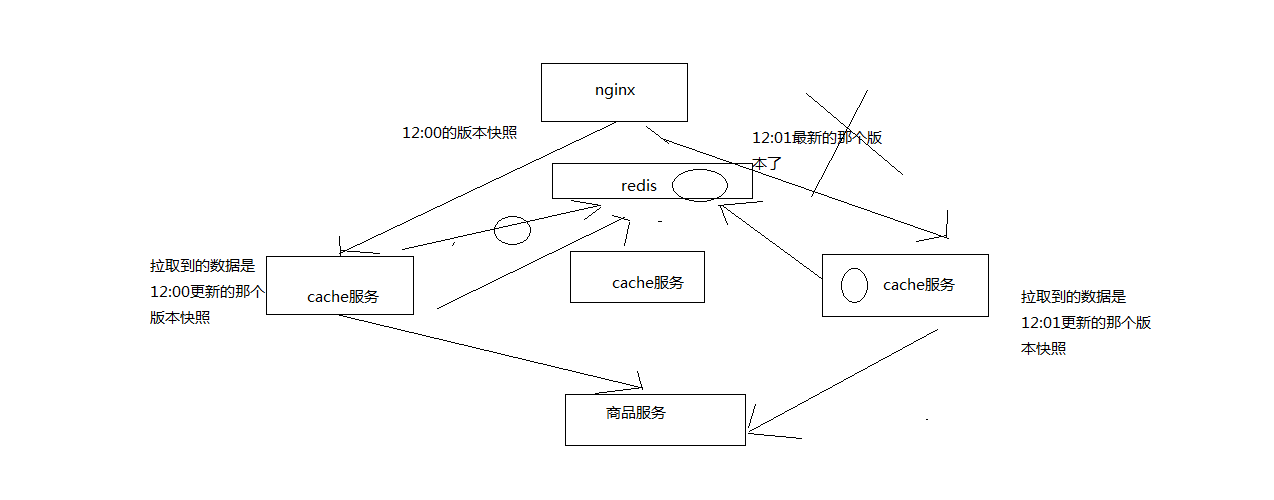
还差最后一个很关键的要点，就是如果你的数据在nginx -> redis -> ehcache三级缓存都不在了，可能就是被LRU清理掉了

这个时候缓存服务会重新拉去数据，去更新到ehcache和redis中，这里我们还没讲解

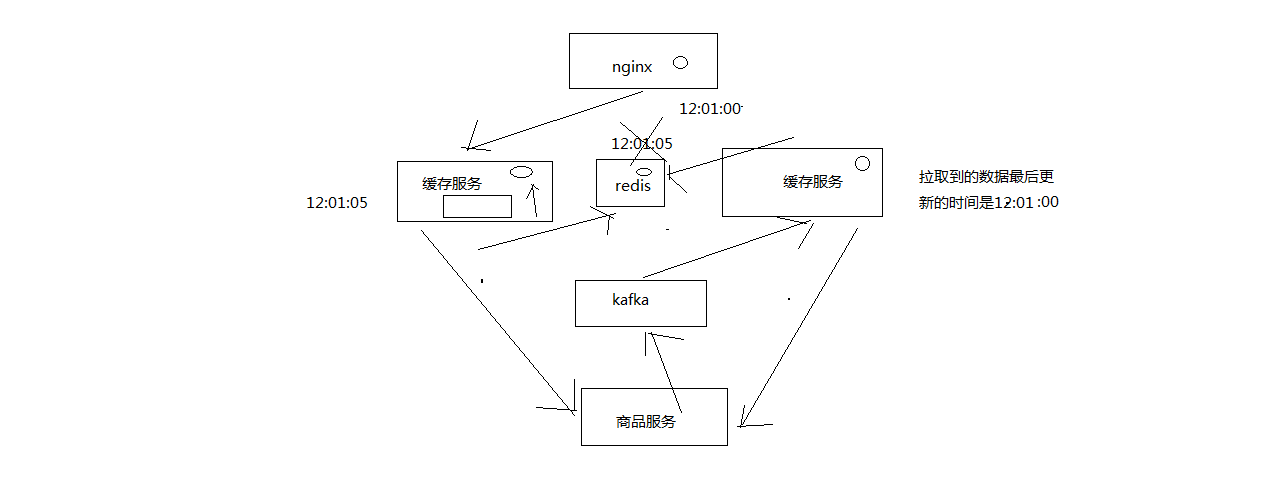
分布式的缓存重建的并发问题

#### 57\_分布式缓存重建并发冲突问题以及zookeeper分布式锁解决方案

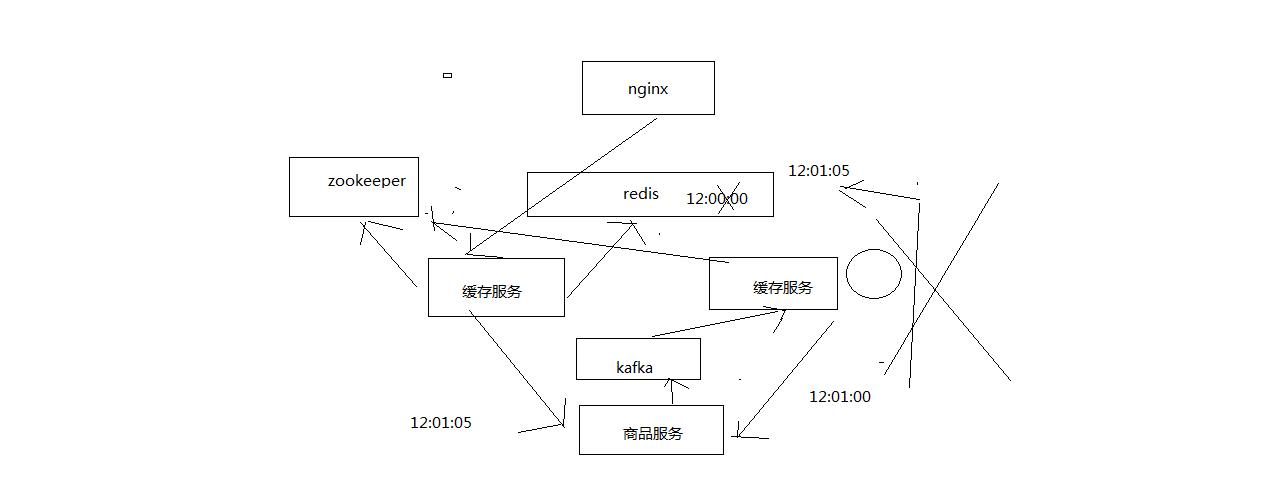
多个缓存服务实例分布式重建的并发冲突问题



缓存更新和缓存重建在不同机器上的并发冲突问题



基于zookeeper分布式锁的冲突解决方案



整个三级缓存的架构已经走通了

我们还遇到一个问题，就是说，如果缓存服务在本地的ehcache中都读取不到数据，那就恩坑爹了

这个时候就意味着，需要重新到源头的服务中去拉去数据，拉取到数据之后，赶紧先给nginx的请求返回，同时将数据写入ehcache和redis中

分布式重建缓存的并发冲突问题

重建缓存：比如我们这里，数据在所有的缓存中都不存在了（LRU算法弄掉了），就需要重新查询数据写入缓存，重建缓存

分布式的重建缓存，在不同的机器上，不同的服务实例中，去做上面的事情，就会出现多个机器分布式重建去读取相同的数据，然后写入缓存中

分布式重建缓存的并发冲突问题。。。。。。

1、流量均匀分布到所有缓存服务实例上

应用层nginx，是将请求流量均匀地打到各个缓存服务实例中的，可能咱们的eshop-cache那个服务，可能会部署多实例在不同的机器上

2、应用层nginx的hash，固定商品id，走固定的缓存服务实例

分发层的nginx的lua脚本，是怎么写的，怎么玩儿的，搞一堆应用层nginx的地址列表，对每个商品id做一个hash，然后对应用nginx数量取模

将每个商品的请求固定分发到同一个应用层nginx上面去

在应用层nginx里，发现自己本地lua shared dict缓存中没有数据的时候，就采取一样的方式，对product id取模，然后将请求固定分发到同一个缓存服务实例中去

这样的话，就不会出现说多个缓存服务实例分布式的去更新那个缓存了

留个作业，大家去做吧，这个东西，之前已经讲解果了，lua脚本几乎都是一模一样的，我们就不去做了，节省点时间

3、源信息服务发送的变更消息，需要按照商品id去分区，固定的商品变更走固定的kafka分区，也就是固定的一个缓存服务实例获取到

缓存服务，是监听kafka topic的，一个缓存服务实例，作为一个kafka consumer，就消费topic中的一个partition

所以你有多个缓存服务实例的话，每个缓存服务实例就消费一个kafka partition

所以这里，一般来说，你的源头信息服务，在发送消息到kafka topic的时候，都需要按照product id去分区

也就时说，同一个product id变更的消息一定是到同一个kafka partition中去的，也就是说同一个product id的变更消息，一定是同一个缓存服务实例消费到的

我们也不去做了，其实很简单，kafka producer api，里面send message的时候，多加一个参数就可以了，product id传递进去，就可以了

4、问题是，自己写的简易的hash分发，与kafka的分区，可能并不一致！！！

我们自己写的简易的hash分发策略，是按照crc32去取hash值，然后再取模的

关键你又不知道你的kafka producer的hash策略是什么，很可能说跟我们的策略是不一样的

拿就可能导致说，数据变更的消息所到的缓存服务实例，跟我们的应用层nginx分发到的那个缓存服务实例也许就不在一台机器上了

这样的话，在高并发，极端的情况下，可能就会出现冲突

5、分布式的缓存重建并发冲突问题发生了。。。

6、基于zookeeper分布式锁的解决方案

分布式锁，如果你有多个机器在访问同一个共享资源，那么这个时候，如果你需要加个锁，让多个分布式的机器在访问共享资源的时候串行起来

那么这个时候，那个锁，多个不同机器上的服务共享的锁，就是分布式锁

分布式锁当然有很多种不同的实现方案，redis分布式锁，zookeeper分布式锁

zk，做分布式协调这一块，还是很流行的，大数据应用里面，hadoop，storm，都是基于zk去做分布式协调

zk分布式锁的解决并发冲突的方案

（1）变更缓存重建以及空缓存请求重建，更新redis之前，都需要先获取对应商品id的分布式锁

（2）拿到分布式锁之后，需要根据时间版本去比较一下，如果自己的版本新于redis中的版本，那么就更新，否则就不更新

（3）如果拿不到分布式锁，那么就等待，不断轮询等待，直到自己获取到分布式的锁

#### 58\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（一）

zk分布式锁的代码封装

zookeeper java client api去封装连接zk，以及获取分布式锁，还有释放分布式锁的代码

先简单介绍一下zk分布式锁的原理

我们通过去创建zk的一个临时node，来模拟给摸一个商品id加锁

zk会给你保证说，只会创建一个临时node，其他请求过来如果再要创建临时node，就会报错，NodeExistsException

那么所以说，我们的所谓上锁，其实就是去创建某个product id对应的一个临时node

如果临时node创建成功了，那么说明我们成功加锁了，此时就可以去执行对redis立面数据的操作

如果临时node创建失败了，说明有人已经在拿到锁了，在操作reids中的数据，那么就不断的等待，直到自己可以获取到锁为止

基于zk client api，去封装上面的这个代码逻辑

释放一个分布式锁，去删除掉那个临时node就可以了，就代表释放了一个锁，那么此时其他的机器就可以成功创建临时node，获取到锁

即使是用zk去实现一个分布式锁，也有很多种做法，有复杂的，也有简单的

应该说，我演示的这种分布式锁的做法，是非常简单的一种，但是很实用，大部分情况下，用这种简单的分布式锁都能搞定

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.5</version>

</dependency>

/\*\*

\* ZooKeeperSession

\* @author Administrator

\*

\*/

public class ZooKeeperSession {

private static CountDownLatch connectedSemaphore = new CountDownLatch(1);

private ZooKeeper zookeeper;

public ZooKeeperSession() {

// 去连接zookeeper server，创建会话的时候，是异步去进行的

// 所以要给一个监听器，说告诉我们什么时候才是真正完成了跟zk server的连接

try {

this.zookeeper = new ZooKeeper(

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

50000,

new ZooKeeperWatcher());

// 给一个状态CONNECTING，连接中

System.out.println(zookeeper.getState());

try {

// CountDownLatch

// java多线程并发同步的一个工具类

// 会传递进去一些数字，比如说1,2 ，3 都可以

// 然后await()，如果数字不是0，那么久卡住，等待

// 其他的线程可以调用coutnDown()，减1

// 如果数字减到0，那么之前所有在await的线程，都会逃出阻塞的状态

// 继续向下运行

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 void acquireDistributedLock(Long productId) {

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

try {

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

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

System.out.println("success to acquire lock for product[id=" + productId + "]");

} catch (Exception e) {

// 如果那个商品对应的锁的node，已经存在了，就是已经被别人加锁了，那么就这里就会报错

// NodeExistsException

int count = 0;

while(true) {

try {

Thread.sleep(20);

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

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

} catch (Exception e2) {

e2.printStackTrace();

count++;

continue;

}

System.out.println("success to acquire lock for product[id=" + productId + "] after " + count + " times try......");

break;

}

}

}

/\*\*

\* 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);

} 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();

}

}

}

/\*\*

\* 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();

}

}

#### 59\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（二）

业务代码

1、主动更新

监听kafka消息队列，获取到一个商品变更的消息之后，去哪个源服务中调用接口拉取数据，更新到ehcache和redis中

先获取分布式锁，然后才能更新redis，同时更新时要比较时间版本

2、被动重建

直接读取源头数据，直接返回给nginx，同时推送一条消息到一个队列，后台线程异步消费

后台现成负责先获取分布式锁，然后才能更新redis，同时要比较时间版本

#### 60\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（三）

无

#### 61\_Java程序员、缓存架构以及Storm大数据实时计算之间的关系

接下来，我们是要讲解这个商品详情页缓存架构，缓存预热问题和解决方案，缓存热点数据可能导致整个系统崩溃的问题，以及解决方案

缓存，热，预热，热数据

解决方案，和架构设计中，会引入大数据的实时计算的技术，storm

为什么要引入这个storm，难道必须是storm吗？我们后面去讲解那个解决方案的时候再说

java工程师，storm的关系是什么呢，缓存架构和storm的关系

缓存架构和storm的关系，因为有些热点数据相关的一些实时处理的一些方案，比如快速预热，热点数据的实时感知和快速降级，全部要用到storm

因为我们可能需要实时的去计算出热点缓存数据，实时计算，亿级流量，高并发，大量的请求过来

这个时候，你要做一些实时的计算，那么必须涉及到分布式的一些技术，分布式的技术，才能处理高并发，大量的请求

目前在时候计算的领域，最成熟的大数据的技术，就是storm

storm分布式的大数据实时计算的技术/系统

java工程师，我跟storm之间的关系是什么？

1、介绍，我自己本身这么多年，一直在大公司，BAT公司，一线的大互联网公司，我认识的很多的java工程师

java开发和架构，后来开始大数据的架构

大公司里的很多java工程师，都是会用一些大数据的一些技术的，比如storm，或者hbase，或者zookeeper，或者hive，spark

因为在大公司里，容易遇到一些复杂的挑战和场景，比如高并发，海量数据，场景

你做一些java先关的项目，和系统，可能也会遇到这种问题，很多时候，直接用大数据的一些技术，实时计算，你是自己去写个系统，还是用现成的storm

更好的选择时用storm，成熟

我也只是说部分java的人，但是也有很多搞java的工程师就是纯java技术栈

2、java系统跟大数据技术的关系

（1）大数据不仅仅只是大数据工程师要关注的东西

（2）大数据也是Java程序员在构建各类系统的时候一种全新的思维，以及架构理念，比如Storm，Hive，Spark，ZooKeeper，HBase，Elasticsearch，等等

（3）举例说明

Storm：实时缓存热点数据统计->缓存预热->缓存热点数据自动降级

Hive：Hadoop生态栈里面，做数据仓库的一个系统，高并发访问下，海量请求日志的批量统计分析，日报周报月报，接口调用情况，业务使用情况，等等

我所知，在一些大公司里面，是有些人是将海量的请求日志打到hive里面，做离线的分析，然后反过来去优化自己的系统

Spark：离线批量数据处理，比如从DB中一次性批量处理几亿数据，清洗和处理后写入Redis中供后续的系统使用，大型互联网公司的用户相关数据

ZooKeeper：分布式系统的协调，分布式锁，分布式选举->高可用HA架构，轻量级元数据存储

用java开发了分布式的系统架构，你的整套系统拆分成了多个部分，每个部分都会负责一些功能，互相之间需要交互和协调

服务A说，我在处理某件事情的时候，服务B你就别处理了

服务A说，我一旦发生了某些状况，希望服务B你立即感知到，然后做出相应的对策

HBase：海量数据的在线存储和简单查询，替代MySQL分库分表，提供更好的伸缩性

java底层，对应的是海量数据，然后要做一些简单的存储和查询，同时数据增多的时候要快速扩容

mysql分库分表就不太合适了，mysql分库分表扩容，还是比较麻烦的

Elasticsearch：海量数据的复杂检索以及搜索引擎的构建，支撑有大量数据的各种企业信息化系统的搜索引擎，电商/新闻等网站的搜索引擎，等等

mysql的like "%xxxx%"，更加合适一些，性能更加好

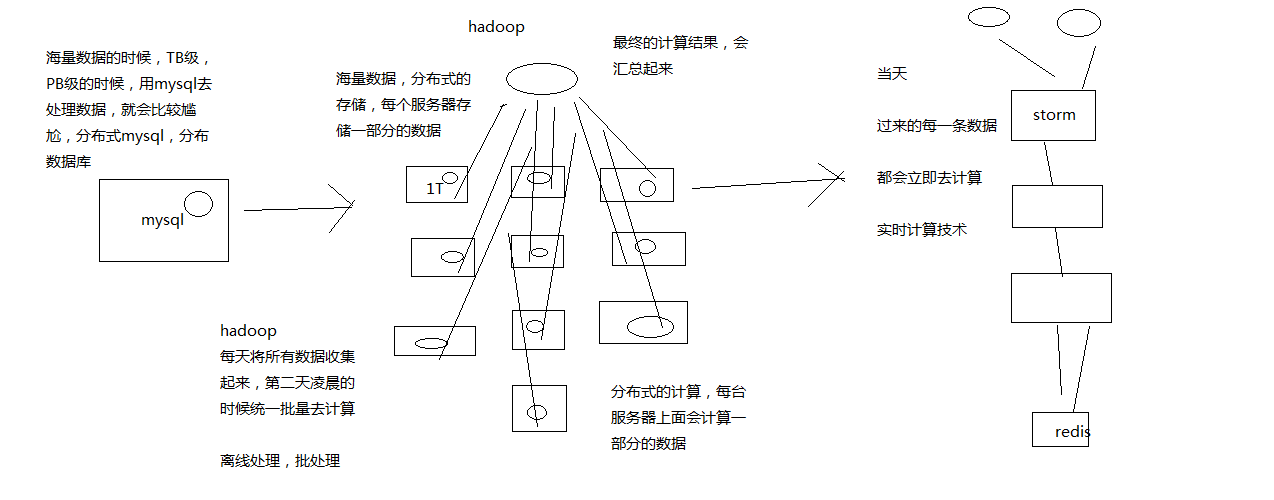
大家不要说觉得来听课程，就必须每堂课都是代码，代码，代码，就不喜欢听我这些废话

我告诉大家，这些还真不是废话，代码很重要，手写代码，copy。我可能做为一个过来人，很多项目都做过，很多技术都用过，也做过。

比较我的角度，去给大家讲一讲，行业，一些技术领域的问题

#### 62\_讲给Java工程师的史上最通俗易懂Storm教程：大白话介绍

mysql、hadoop与storm的关系



这块给大家解释一下，就是说，有些技术我们可能就是简单带着大家去用一下就好了

nginx，java，一般都会一些

kafka，zookeeper，lua，我觉得，那些东西的话，主要是讲解基于他们的一些架构，和解决方案的设计还有开发

redis：跟我们的这个topic是很有关系的，大型缓存架构，高并发高性能高可用的缓存架构的底层支持，redis，细致的去讲解，那块redis技术和知识是本套课程的一个重点

数据库+缓存双写，多级缓存架构，大家重点去理解里面的方案设计和架构思想

热数据的处理，缓存雪崩 --> storm，hystrix

对于这两个技术，都是关键性的会去影响你的热数据，缓存雪崩时的系统可用性和稳定性

对这两个技术，storm，hystrix，都很重要

会类似redis，花费较多的篇幅去给大家讲解一下，让大家可以把这两个技术同时也学习的非常好

正好跟着我们的大的项目实战在走，学完以后，直接可以学以致用，用到我们的系统架构中去

kafka，消息队列，用起来很简单，而且搞java得一般来说，对消息队列都有一些了解吧，而且到了真实的生产环境中，kafka你是可以换成其他的技术，Active MQ，Rabbit MQ，Rocket MQ

zookeeper，分布式锁，分布式锁，搞java一般也会知道一些，zk去做，redis去做锁也是可以的

lua，大家后面真的是要用到lua，觉得课程里讲解的东西不够，可以自己去网上查一些lua的语法可以了，语法是最最简单的

storm，说句实话，在做热数据这块，如果要做复杂的热数据的统计和分析，亿流量，高并发的场景下，我还真觉得，最合适的技术就是storm，没有其他

缓存架构，热数据先关的架构设计，热数据相关的架构中最重要的唯一的可选技术，storm，好好的去讲一下的

hystrix，分布式系统的高可用性的限流，熔断，降级，等等，一些措施，缓存雪崩的方案，限流的技术

讲给Java工程师的史上最通俗易懂Storm教程

讲给Java工程师：我知道你没什么大数据的背景和经验，基础，那么我就把你当做一个大数据小白，主要是java背景和基础

史上最通俗易懂：市面上其他的storm视频课程，或者是一些书籍，我告诉，storm还是挺难的，事务，云里雾里，云里雾里

搞storm大数据的，连这个并行度和流分组的本质它都说不清楚，因为市面上的资料也说不清楚

会把你当做小白，用最最通俗易懂的语言，给你去讲解这块的知识，画图

一、Storm到底是什么？

1、mysql，hadoop与storm

mysql：事务性系统，面临海量数据的尴尬

hadoop：离线批处理

storm：实时计算

2、我们能不能自己搞一套storm？

来一条数据，我理解就算一条，来一条，算一条

坑，海量高并发大数据，高并发的请求数据，分布式的系统，流式处理的分布式系统

如果自己搞一套实时流系统出来，也是可以的，但是。。。。

（1）花费大量的时间在底层技术细节上：如何部署各种中间队列，节点间的通信，容错，资源调配，计算节点的迁移和部署，等等

（2）花费大量的时间在系统的高可用上问题上：如何保证各种节点能够高可用稳定运行

（3）花费大量的时间在系统扩容上：吞吐量需要扩容的时候，你需要花费大量的时间去增加节点，修改配置，测试，等等

5万/s，10万/s，扩容

国内，国产的实时大数据计算系统，唯一做出来的，做得好的，做得影响力特别大，特别牛逼的，就是JStorm，阿里

阿里，技术实力，世界一流，顶尖，国内顶尖，一流

JStorm，clojure编程预压，Java重新写了一遍，Galaxy流式计算的系统，百度，腾讯，也都自己做了，也能做得很好

3、storm的特点是什么？

（1）支撑各种实时类的项目场景：实时处理消息以及更新数据库，基于最基础的实时计算语义和API（实时数据处理领域）；对实时的数据流持续的进行查询或计算，同时将最新的计算结果持续的推送给客户端展示，同样基于最基础的实时计算语义和API（实时数据分析领域）；对耗时的查询进行并行化，基于DRPC，即分布式RPC调用，单表30天数据，并行化，每个进程查询一天数据，最后组装结果

storm做各种实时类的项目都ok

（2）高度的可伸缩性：如果要扩容，直接加机器，调整storm计算作业的并行度就可以了，storm会自动部署更多的进程和线程到其他的机器上去，无缝快速扩容

扩容起来，超方便

（3）数据不丢失的保证：storm的消息可靠机制开启后，可以保证一条数据都不丢

数据不丢失，也不重复计算

（4）超强的健壮性：从历史经验来看，storm比hadoop、spark等大数据类系统，健壮的多的多，因为元数据全部放zookeeper，不在内存中，随便挂都不要紧

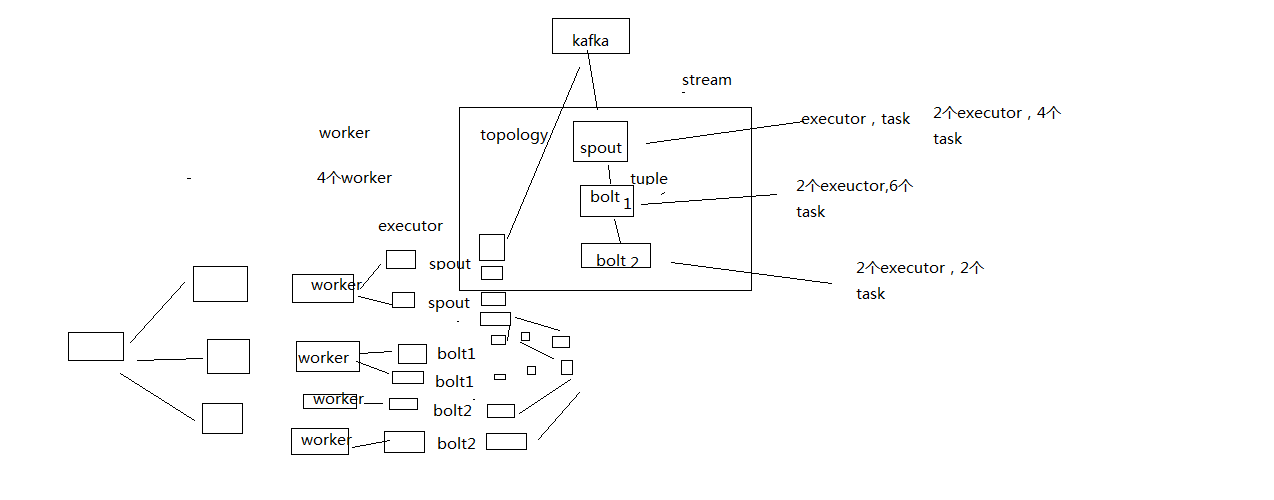
特别的健壮，稳定性和可用性很高

（5）使用的便捷性：核心语义非常的简单，开发起来效率很高

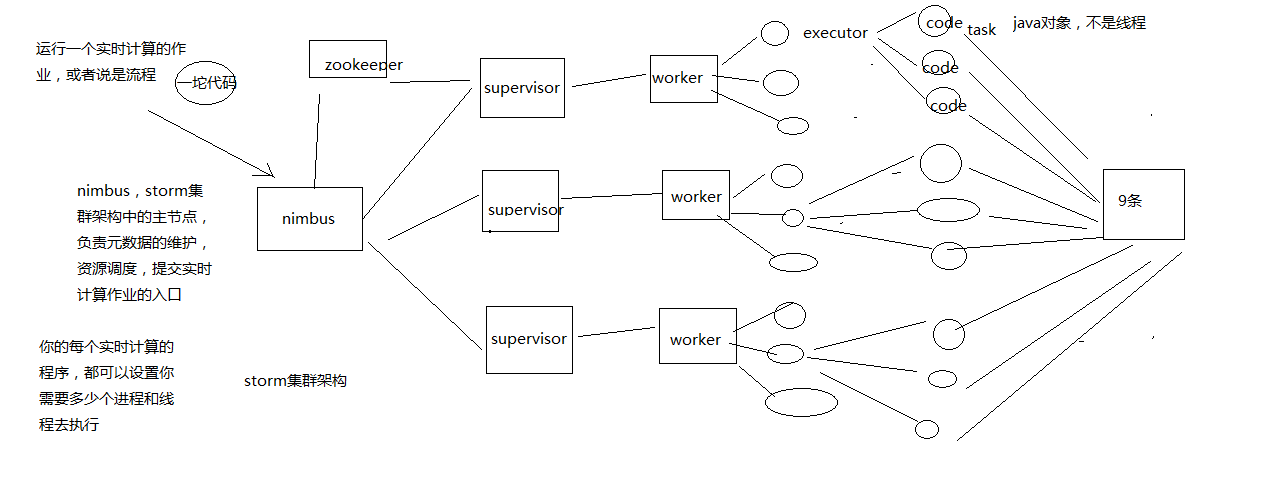
用起来很简单，开发API还是很简单的

#### 63\_讲给Java工程师的史上最通俗易懂Storm教程：大白话讲集群架构与核心概念

storm核心概念



storm集群架构



大白话讲解

二、Storm的集群架构以及核心概念

1、Storm的集群架构

Nimbus，Supervisor，ZooKeeper，Worker，Executor，Task

2、Storm的核心概念

Topology，Spout，Bolt，Tuple，Stream

拓扑：务虚的一个概念

Spout：数据源的一个代码组件，就是我们可以实现一个spout接口，写一个java类，在这个spout代码中，我们可以自己尝试去数据源获取数据，比如说从kafka中消费数据

bolt：一个业务处理的代码组件，spout会将数据传送给bolt，各种bolt还可以串联成一个计算链条，java类实现了一个bolt接口

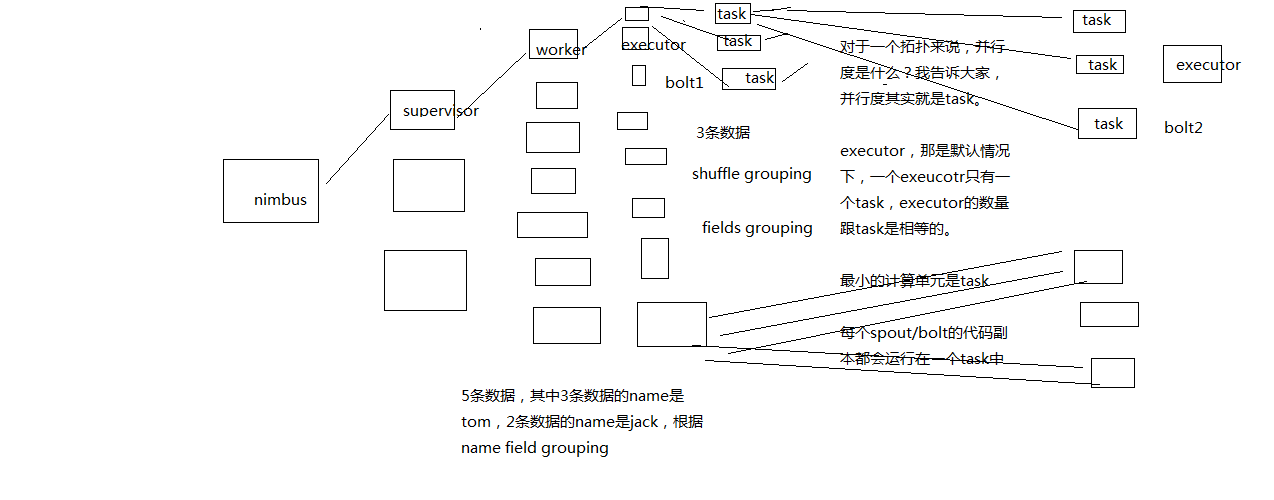
一堆spout+bolt，就会组成一个topology，就是一个拓扑，实时计算作业，spout+bolt，一个拓扑涵盖数据源获取/生产+数据处理的所有的代码逻辑，topology

tuple：就是一条数据，每条数据都会被封装在tuple中，在多个spout和bolt之间传递

stream：就是一个流，务虚的一个概念，抽象的概念，源源不断过来的tuple，就组成了一条数据流

#### 64\_讲给Java工程师的史上最通俗易懂Storm教程：大白话讲并行度和流分组

并行度和流分组



三、Storm的并行度以及流分组

因为我们在这里，是讲给java工程师的storm教程

所以我期望的场景是，你们所在的公司，基本上来说，已经有大数据团队，有人在维护storm集群

我觉得，对于java工程师来说，先不说精通storm

至少说，对storm的核心的基本原理，门儿清，你都很清楚，集群架构、核心概念、并行度和流分组

接下来，掌握最常见的storm开发范式，spout消费kafka，后面跟一堆bolt，bolt之间设定好流分组的策略

在bolt中填充各种代码逻辑

了解如何将storm拓扑打包后提交到storm集群上去运行

掌握如何能够通过storm ui去查看你的实时计算拓扑的运行现状

你在一个公司里，如果说，需要在你的java系统架构中，用到一些类似storm的大数据技术，如果已经有人维护了storm的集群

那么此时你就可以直接用，直接掌握如何开发和部署即可

但是，当然了，如果说，恰巧没人负责维护storm集群，也没什么大数据的团队，那么你可能需要说再去深入学习一下storm

当然了，如果你的场景不是特别复杂，整个数据量也不是特别大，其实自己主要研究一下，怎么部署storm集群

你自己部署一个storm集群，也ok

好多年前，我第一次接触storm的时候，真的，我觉得都没几个人能彻底讲清楚，用一句话讲清楚什么是并行度，什么是流分组

很多时候，你以外你明白了，其实你不明白

比如我经常面试一些做过storm的人过来，我就问一个问题，就知道它的水深水浅，流分组的时候，数据在storm集群中的流向，你画一下

比如你自己随便设想一个拓扑结果出来，几个spout，几个bolt，各种流分组情况下，数据是怎么流向的，要求具体画出集群架构中的流向

worker，executor，task，supervisor，流的

几乎没几个人能画对，为什么呢，很多人就没搞明白这个并行度和流分组到底是什么

并行度：Worker->Executor->Task，没错，是Task

流分组：Task与Task之间的数据流向关系

Shuffle Grouping：随机发射，负载均衡

Fields Grouping：根据某一个，或者某些个，fields，进行分组，那一个或者多个fields如果值完全相同的话，那么这些tuple，就会发送给下游bolt的其中固定的一个task

你发射的每条数据是一个tuple，每个tuple中有多个field作为字段

比如tuple，3个字段，name，age，salary

{"name": "tom", "age": 25, "salary": 10000} -> tuple -> 3个field，name，age，salary

All Grouping

Global Grouping

None Grouping

Direct Grouping

Local or Shuffle Grouping

#### 65\_讲给Java工程师的史上最通俗易懂Storm教程：纯手敲WordCount程序

storm核心的基本原理，都了解了

写一下代码，去体验一下，storm的程序是怎么开发的，通过了解了代码之后，再回头，你去看一下之前讲解的一些基本原理，就很清楚了

大数据，入门程序，wordcount，单词计数

你可以认为，storm源源不断的接收到一些句子，然后你需要实时的统计出句子中每个单词的出现次数

（1）搭建工程环境

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<artifactId>storm-wordcount</artifactId>

<packaging>jar</packaging>

<name>storm-wordcount</name>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>1.1.0</version>

</dependency>

<dependency>

<groupId>commons-collections</groupId>

<artifactId>commons-collections</artifactId>

<version>3.2.1</version>

</dependency>

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

</dependency>

</dependencies>

<build>

<sourceDirectory>src/main/java</sourceDirectory>

<testSourceDirectory>test/main/java</testSourceDirectory>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<configuration>

<createDependencyReducedPom>true</createDependencyReducedPom>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.sf</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.dsa</exclude>

<exclude>META-INF/\*.RSA</exclude>

<exclude>META-INF/\*.rsa</exclude>

<exclude>META-INF/\*.EC</exclude>

<exclude>META-INF/\*.ec</exclude>

<exclude>META-INF/MSFTSIG.SF</exclude>

<exclude>META-INF/MSFTSIG.RSA</exclude>

</excludes>

</filter>

</filters>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer" />

<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>exec-maven-plugin</artifactId>

<version>1.2.1</version>

<executions>

<execution>

<goals>

<goal>exec</goal>

</goals>

</execution>

</executions>

<configuration>

<executable>java</executable>

<includeProjectDependencies>true</includeProjectDependencies>

<includePluginDependencies>false</includePluginDependencies>

<classpathScope>compile</classpathScope>

<mainClass></mainClass>

</configuration>

</plugin>

</plugins>

</build>

</project>

（2）编写代码

public class WordCountTopology {

public static class RandomSentenceSpout extends BaseRichSpout {

SpoutOutputCollector \_collector;

Random \_rand;

@Override

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {

\_collector = collector;

\_rand = new Random();

}

@Override

public void nextTuple() {

Utils.sleep(100);

String[] sentences = new String[]{sentence("the cow jumped over the moon"), sentence("an apple a day keeps the doctor away"),

sentence("four score and seven years ago"), sentence("snow white and the seven dwarfs"), sentence("i am at two with nature")};

final String sentence = sentences[\_rand.nextInt(sentences.length)];

\_collector.emit(new Values(sentence));

}

protected String sentence(String input) {

return input;

}

@Override

public void ack(Object id) {

}

@Override

public void fail(Object id) {

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word"));

}

}

public static class SplitSentence implements IRichBolt {

public SplitSentence() {

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word"));

}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

}

public static class WordCount extends BaseBasicBolt {

Map<String, Integer> counts = new HashMap<String, Integer>();

@Override

public void execute(Tuple tuple, BasicOutputCollector collector) {

String word = tuple.getString(0);

Integer count = counts.get(word);

if (count == null)

count = 0;

count++;

counts.put(word, count);

collector.emit(new Values(word, count));

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word", "count"));

}

}

public static void main(String[] args) throws Exception {

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);

builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");

builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

Config conf = new Config();

conf.setDebug(true);

if (args != null && args.length > 0) {

conf.setNumWorkers(3);

StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());

}

else {

conf.setMaxTaskParallelism(3);

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("word-count", conf, builder.createTopology());

Thread.sleep(10000);

cluster.shutdown();

}

}

}

（4）测试代码

#### 66\_讲给Java工程师的史上最通俗易懂Storm教程：纯手工集群部署

讲了手写了storm wordcount程序

蕴含了很多的知识点

（1）Spout

（2）Bolt

（3）OutputCollector，Declarer

（4）Topology

（5）设置worker，executor，task，流分组

storm的核心基本原理，基本的开发，学会了

storm集群部署，怎么将storm的拓扑扔到storm集群上去跑

六、部署一个storm集群

（1）安装Java 7和Pythong 2.6.6

（2）下载storm安装包，解压缩，重命名，配置环境变量

（3）修改storm配置文件

mkdir /var/storm

conf/storm.yaml

storm.zookeeper.servers:

- "111.222.333.444"

- "555.666.777.888"

storm.local.dir: "/mnt/storm"

nimbus.seeds: ["111.222.333.44"]

slots.ports，指定每个机器上可以启动多少个worker，一个端口号代表一个worker

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

(4)启动storm集群和ui界面

一个节点，storm nimbus >/dev/null 2>&1 &

三个节点，storm supervisor >/dev/null 2>&1 &

一个节点，storm ui >/dev/null 2>&1 &

（5）访问一下ui界面，8080端口

#### 67\_讲给Java工程师的史上最通俗易懂Storm教程：基于集群运行计算拓扑

七、提交作业到storm集群来运行

将eclipse中的工程，进行打包

（1）提交作业到storm集群

storm jar path/to/allmycode.jar org.me.MyTopology arg1 arg2 arg3

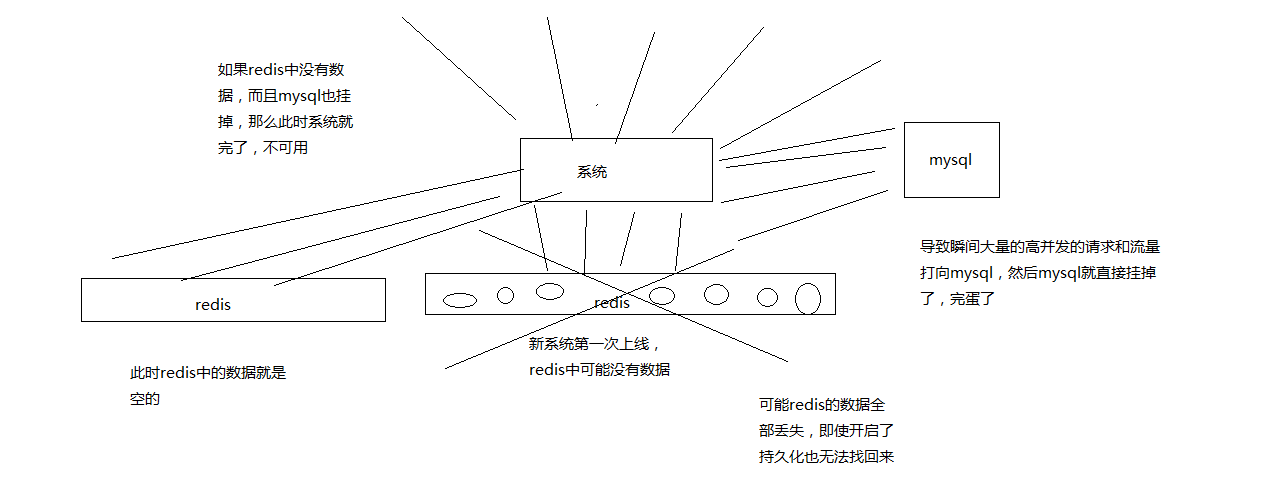
（2）在storm ui上观察storm作业的运行

（3）kill掉某个storm作业

storm kill topology-name

#### 68\_缓存冷启动问题：新系统上线、redis彻底崩溃导致数据无法恢复

缓存冷启动问题



缓存冷启动的问题

新系统第一次上线，此时在缓存里可能是没有数据的

系统在线上稳定运行着，但是突然间重要的redis缓存全盘崩溃了，而且不幸的是，数据全都无法找回来

系统第一次上线启动，系统在redis故障的情况下重新启动，在高并发的场景下

#### 69\_缓存预热解决方案：基于storm实时热点统计的分布式并行缓存预热

0、缓存预热

缓存冷启动，redis启动后，一点数据都没有，直接就对外提供服务了，mysql就裸奔

（1）提前给redis中灌入部分数据，再提供服务

（2）肯定不可能将所有数据都写入redis，因为数据量太大了，第一耗费的时间太长了，第二根本redis容纳不下所有的数据

（3）需要根据当天的具体访问情况，实时统计出访问频率较高的热数据

（4）然后将访问频率较高的热数据写入redis中，肯定是热数据也比较多，我们也得多个服务并行读取数据去写，并行的分布式的缓存预热

（5）然后将灌入了热数据的redis对外提供服务，这样就不至于冷启动，直接让数据库裸奔了

1、nginx+lua将访问流量上报到kafka中

要统计出来当前最新的实时的热数据是哪些，我们就得将商品详情页访问的请求对应的流浪，日志，实时上报到kafka中

2、storm从kafka中消费数据，实时统计出每个商品的访问次数，访问次数基于LRU内存数据结构的存储方案

优先用内存中的一个LRUMap去存放，性能高，而且没有外部依赖

我之前做过的一些项目，不光是这个项目，还有很多其他的，一些广告计费类的系统，storm

否则的话，依赖redis，我们就是要防止redis挂掉数据丢失的情况，就不合适了; 用mysql，扛不住高并发读写; 用hbase，hadoop生态系统，维护麻烦，太重了

其实我们只要统计出最近一段时间访问最频繁的商品，然后对它们进行访问计数，同时维护出一个前N个访问最多的商品list即可

热数据，最近一段时间，可以拿到最近一段，比如最近1个小时，最近5分钟，1万个商品请求，统计出最近这段时间内每个商品的访问次数，排序，做出一个排名前N的list

计算好每个task大致要存放的商品访问次数的数量，计算出大小

然后构建一个LRUMap，apache commons collections有开源的实现，设定好map的最大大小，就会自动根据LRU算法去剔除多余的数据，保证内存使用限制

即使有部分数据被干掉了，然后下次来重新开始计数，也没关系，因为如果它被LRU算法干掉，那么它就不是热数据，说明最近一段时间都很少访问了

3、每个storm task启动的时候，基于zk分布式锁，将自己的id写入zk同一个节点中

4、每个storm task负责完成自己这里的热数据的统计，每隔一段时间，就遍历一下这个map，然后维护一个前3个商品的list，更新这个list

5、写一个后台线程，每隔一段时间，比如1分钟，都将排名前3的热数据list，同步到zk中去，存储到这个storm task对应的一个znode中去

6、我们需要一个服务，比如说，代码可以跟缓存数据生产服务放一起，但是也可以放单独的服务

服务可能部署了很多个实例

每次服务启动的时候，就会去拿到一个storm task的列表，然后根据taskid，一个一个的去尝试获取taskid对应的znode的zk分布式锁

如果能获取到分布式锁的话，那么就将那个storm task对应的热数据的list取出来

然后将数据从mysql中查询出来，写入缓存中，进行缓存的预热，多个服务实例，分布式的并行的去做，基于zk分布式锁做了协调了，分布式并行缓存的预热

#### 70\_基于nginx+lua完成商品详情页访问流量实时上报kafka的开发

在nginx这一层，接收到访问请求的时候，就把请求的流量上报发送给kafka

这样的话，storm才能去消费kafka中的实时的访问日志，然后去进行缓存热数据的统计

用得技术方案非常简单，从lua脚本直接创建一个kafka producer，发送数据到kafka

wget https://github.com/doujiang24/lua-resty-kafka/archive/master.zip

yum install -y unzip

unzip lua-resty-kafka-master.zip

cp -rf /usr/local/lua-resty-kafka-master/lib/resty /usr/hello/lualib

nginx -s reload

local cjson = require("cjson")

local producer = require("resty.kafka.producer")

local broker\_list = {

{ host = "192.168.31.187", port = 9092 },

{ host = "192.168.31.19", port = 9092 },

{ host = "192.168.31.227", port = 9092 }

}

local log\_json = {}

log\_json["headers"] = ngx.req.get\_headers()

log\_json["uri\_args"] = ngx.req.get\_uri\_args()

log\_json["body"] = ngx.req.read\_body()

log\_json["http\_version"] = ngx.req.http\_version()

log\_json["method"] =ngx.req.get\_method()

log\_json["raw\_reader"] = ngx.req.raw\_header()

log\_json["body\_data"] = ngx.req.get\_body\_data()

local message = cjson.encode(log\_json);

local productId = ngx.req.get\_uri\_args()["productId"]

local async\_producer = producer:new(broker\_list, { producer\_type = "async" })

local ok, err = async\_producer:send("access-log", productId, message)

if not ok then

ngx.log(ngx.ERR, "kafka send err:", err)

return

end

两台机器上都这样做，才能统一上报流量到kafka

bin/kafka-topics.sh --zookeeper 192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181 --topic access-log --replication-factor 1 --partitions 1 --create

bin/kafka-console-consumer.sh --zookeeper 192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181 --topic access-log --from-beginning

（1）kafka在187上的节点死掉了，可能是虚拟机的问题，杀掉进程，重新启动一下

nohup bin/kafka-server-start.sh config/server.properties &

（2）需要在nginx.conf中，http部分，加入resolver 8.8.8.8;

（3）需要在kafka中加入advertised.host.name = 192.168.31.187，重启三个kafka进程

（4）需要启动eshop-cache缓存服务，因为nginx中的本地缓存可能不在了

#### 71\_基于storm+kafka完成商品访问次数实时统计拓扑的开发

maven构建出的一些问题，直接从maven中央仓库可能下载不到jar包，自己去百度一下jar，下载下来

根据错误提示，拷贝到maven本地仓库对应的目录中去，然后手工安装一下

1、kafka consumer spout

单独的线程消费，写入队列

nextTuple，每次都是判断队列有没有数据，有的话再去获取并发射出去，不能阻塞

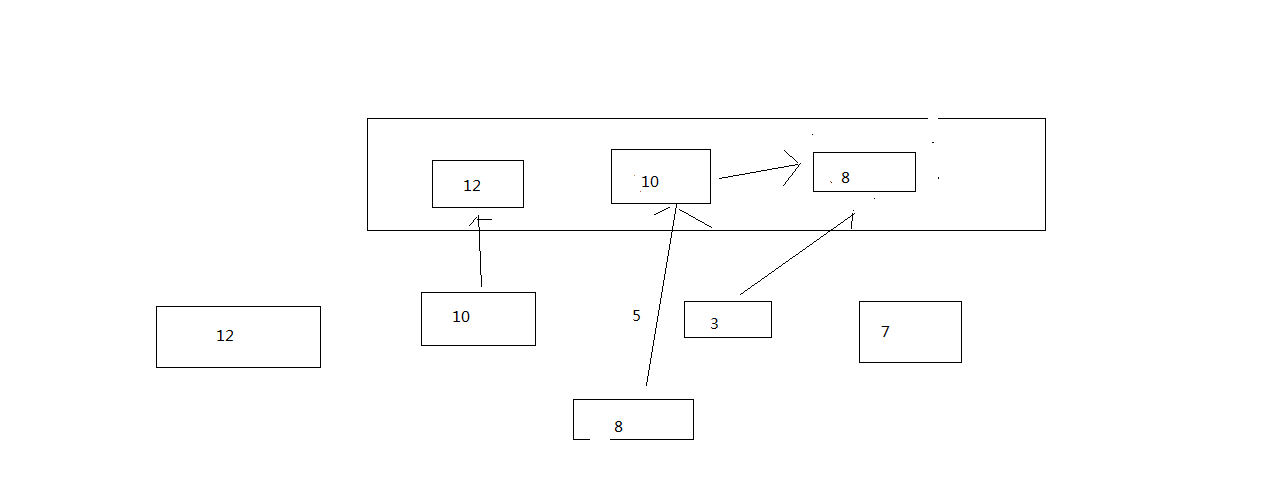
2、日志解析bolt

3、商品访问次数统计bolt

基于LRUMap完成统计

#### 72\_基于storm完成LRUMap中topn热门商品列表的算法讲解与编写

topn list生成算法讲解



1、storm task启动的时候，基于分布式锁将自己的taskid累加到一个znode中

2、开启一个单独的后台线程，每隔1分钟算出top3热门商品list

3、每个storm task将自己统计出的热数据list写入自己对应的znode中

#### 73\_基于storm+zookeeper完成热门商品列表的分段存储

1、task初始化

2、热门商品list保存

#### 74\_基于双重zookeeper分布式锁完成分布式并行缓存预热的代码开发

1、服务启动的时候，进行缓存预热

2、从zk中读取taskid列表

3、依次遍历每个taskid，尝试获取分布式锁，如果获取不到，快速报错，不要等待，因为说明已经有其他服务实例在预热了

4、直接尝试获取下一个taskid的分布式锁

5、即使获取到了分布式锁，也要检查一下这个taskid的预热状态，如果已经被预热过了，就不再预热了

6、执行预热操作，遍历productid列表，查询数据，然后写ehcache和redis

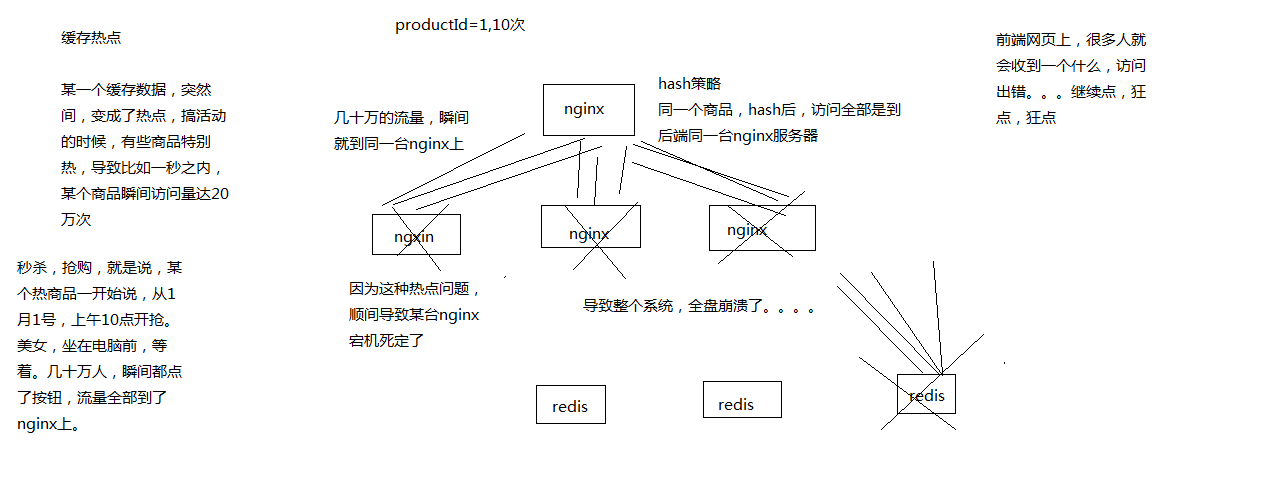
7、预热完成后，设置taskid对应的预热状态

#### 75\_将缓存预热解决方案的代码运行后观察效果以及调试和修复所有的bug

缓存预热，我们已经全部搞完了，所以说，接下来呢，storm拓扑，缓存服务，都给跑起来，看看能不能符合我们的期望

#### 76\_热点缓存问题：促销抢购时的超级热门商品可能导致系统全盘崩溃的场景

热点缓存导致系统崩溃的问题



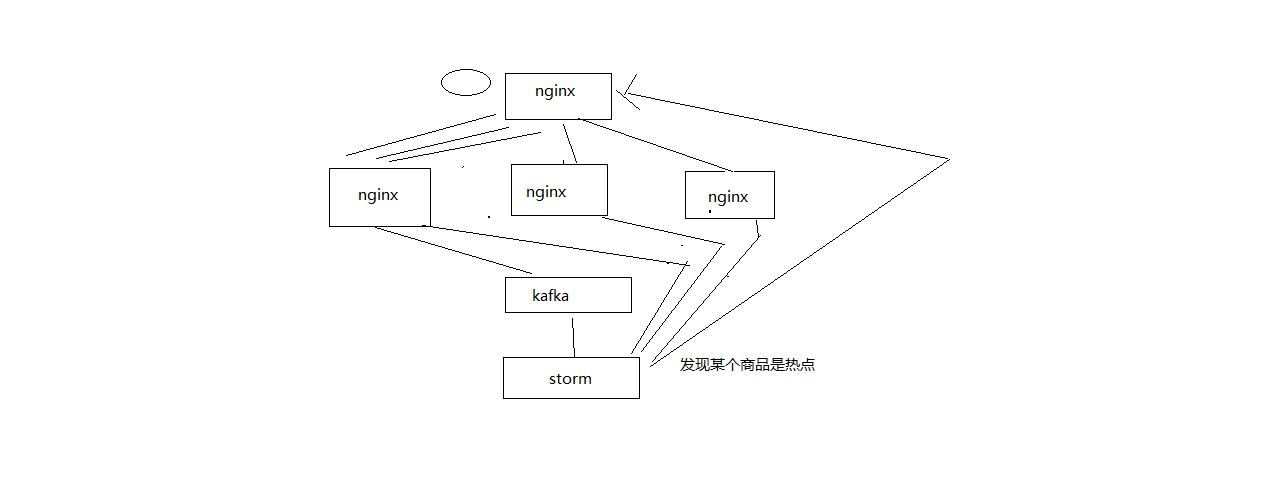
热数据 -> 热数据的统计 -> redis中缓存的预热 -> 避免新系统刚上线，或者是redis崩溃数据丢失后重启，redis中没有数据，redis冷启动 -> 大量流量直接到数据库

redis启动前，必须确保其中是有部分热数据的缓存的

瞬间的缓存热点

#### 77\_基于nginx+lua+storm的热点缓存的流量分发策略自动降级解决方案

热点缓存的解决方案



1、在storm中，实时的计算出瞬间出现的热点

有很多种算法，给大家介绍一种我们的比较简单的算法

某个storm task，上面算出了1万个商品的访问次数，LRUMap

频率高一些，每隔5秒，去遍历一次LRUMap，将其中的访问次数进行排序，统计出往后排的95%的商品访问次数的平均值

1000

999

888

777

666

50

60

80

100

120

比如说，95%的商品，访问次数的平均值是100

然后，从最前面开始，往后遍历，去找有没有瞬间出现的热点数据

1000，95%的平均值（100）的10倍，这个时候要设定一个阈值，比如说超出95%平均值得n倍，5倍

我们就认为是瞬间出现的热点数据，判断其可能在短时间内继续扩大的访问量，甚至达到平均值几十倍，或者几百倍

当遍历，发现说第一个商品的访问次数，小于平均值的5倍，就安全了，就break掉这个循环

热点数据，热数据，不是一个概念

有100个商品，前10个商品比较热，都访问量在500左右，其他的普通商品，访问量都在200左右，就说前10个商品是热数据

统计出来

预热的时候，将这些热数据放在缓存中去预热就可以了

热点，前面某个商品的访问量，瞬间超出了普通商品的10倍，或者100倍，1000倍，热点

2、storm这里，会直接发送http请求到nginx上，nginx上用lua脚本去处理这个请求

storm会将热点本身对应的productId，发送到流量分发的nginx上面去，放在本地缓存中

storm会将热点对应的完整的缓存数据，发送到所有的应用nginx服务器上去，直接放在本地缓存中

3、流量分发nginx的分发策略降级

流量分发nginx，加一个逻辑，就是每次访问一个商品详情页的时候，如果发现它是个热点，那么立即做流量分发策略的降级

hash策略，同一个productId的访问都同一台应用nginx服务器上

降级成对这个热点商品，流量分发采取随机负载均衡发送到所有的后端应用nginx服务器上去

瞬间将热点缓存数据的访问，从hash分发，全部到一台nginx，变成了，负载均衡发送到多台nginx上去

避免说大量的流量全部集中到一台机器，50万的访问量到一台nginx，5台应用nginx，每台就可以承载10万的访问量

4、storm还需要保存下来上次识别出来的热点list

下次去识别的时候，这次的热点list跟上次的热点list做一下diff，看看可能有的商品已经不是热点了

热点的取消的逻辑，发送http请求到流量分发的nginx上去，取消掉对应的热点数据，从nginx本地缓存中，删除

#### 78\_在storm拓扑中加入热点缓存实时自动识别和感知的代码逻辑

无

#### 79\_在storm拓扑中加入nginx反向推送缓存热点与缓存数据的代码逻辑

<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;

}

}

#### 80\_在流量分发+后端应用双层nginx中加入接收热点缓存数据的接口

流量分发

local uri\_args = ngx.req.get\_uri\_args()

local product\_id = uri\_args["productId"]

local cache\_ngx = ngx.shared.my\_cache

local hot\_product\_cache\_key = "hot\_product\_"..product\_id

cache\_ngx:set(hot\_product\_cache\_key, "true", 60 \* 60)

后端应用

local uri\_args = ngx.req.get\_uri\_args()

local product\_id = uri\_args["productId"]

local product\_info = uri\_args["productInfo"]

local product\_cache\_key = "product\_info\_"..product\_id

local cache\_ngx = ngx.shared.my\_cache

cache\_ngx:set(product\_cache\_key,product\_info,60 \* 60)

#### 81\_在nginx+lua中实现热点缓存自动降级为负载均衡流量分发策略的逻辑

math.randomseed(tostring(os.time()):reverse():sub(1, 7))

math.random(1, 2)

local uri\_args = ngx.req.get\_uri\_args()

local productId = uri\_args["productId"]

local shopId = uri\_args["shopId"]

local hosts = {"192.168.31.187", "192.168.31.19"}

local backend = ""

local hot\_product\_key = "hot\_product\_"..productId

local cache\_ngx = ngx.shared.my\_cache

local hot\_product\_flag = cache\_ngx:get(hot\_product\_key)

if hot\_product\_flag == "true" then

math.randomseed(tostring(os.time()):reverse():sub(1, 7))

local index = math.random(1, 2)

backend = "http://"..hosts[index]

else

local hash = ngx.crc32\_long(productId)

local index = (hash % 2) + 1

backend = "http://"..hosts[index]

end

local requestPath = uri\_args["requestPath"]

requestPath = "/"..requestPath.."?productId="..productId.."&shopId="..shopId

local http = require("resty.http")

local httpc = http.new()

local resp, err = httpc:request\_uri(backend,{

method = "GET",

path = requestPath

})

if not resp then

ngx.say("request error: ", err)

return

end

ngx.say(resp.body)

httpc:close()

#### 82\_在storm拓扑中加入热点缓存消失的实时自动识别和感知的代码逻辑

无

#### 83\_将热点缓存自动降级解决方案的代码运行后观察效果以及调试和修复bug

1、storm中打印日志

2、重新部署storm拓扑

3、nginx中修改html模板

4、手动构造出一个热点缓存出来，看热点缓存能否进行负载均衡

5、手动让热点缓存消失，看热点缓存能否自动小时，重新进行hash分发

http://192.168.31.187/hot?productId=15&productInfo={"id":15,"name":"iphone7手机","price":5599.0,"pictureList":"a.jpg,b.jpg","specification":"iphone7的规格","service":"iphone7的售后服务","color":"红色,白色,黑色","size":"5.5","shopId":1,"modifiedTime":"2017-01-01 12:01:00"}

#### 84\_hystrix与高可用系统架构：资源隔离+限流+熔断+降级+运维监控

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

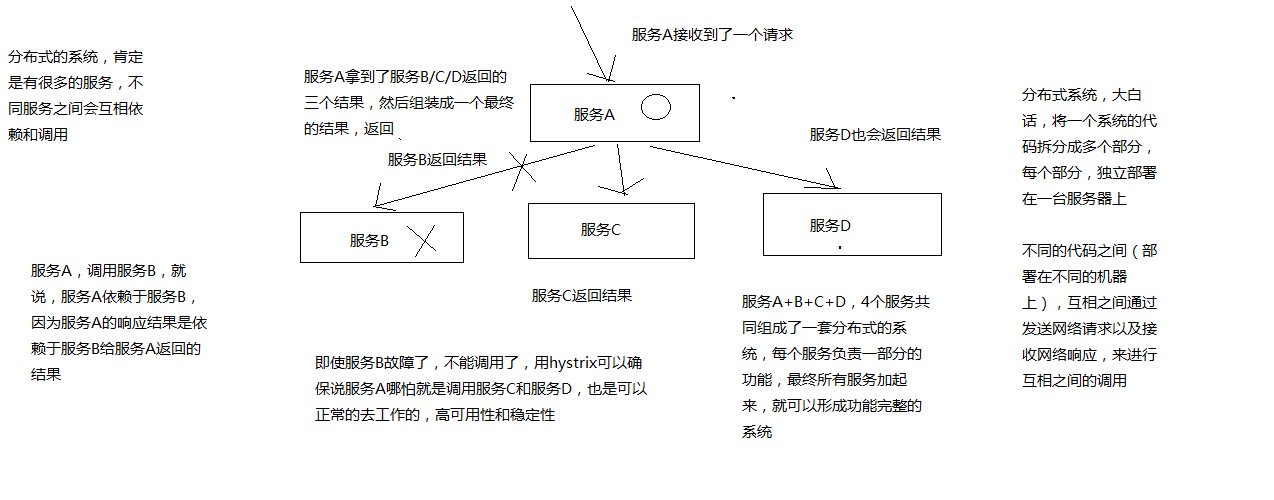
我之前做的一个项目，我们多个项目都用了公司里公用的缓存的存储，缓存彻底挂了，雪崩了，导致各种业务系统全部崩溃，崩溃了好几个小时

导致公司损失了大量的资金的损失

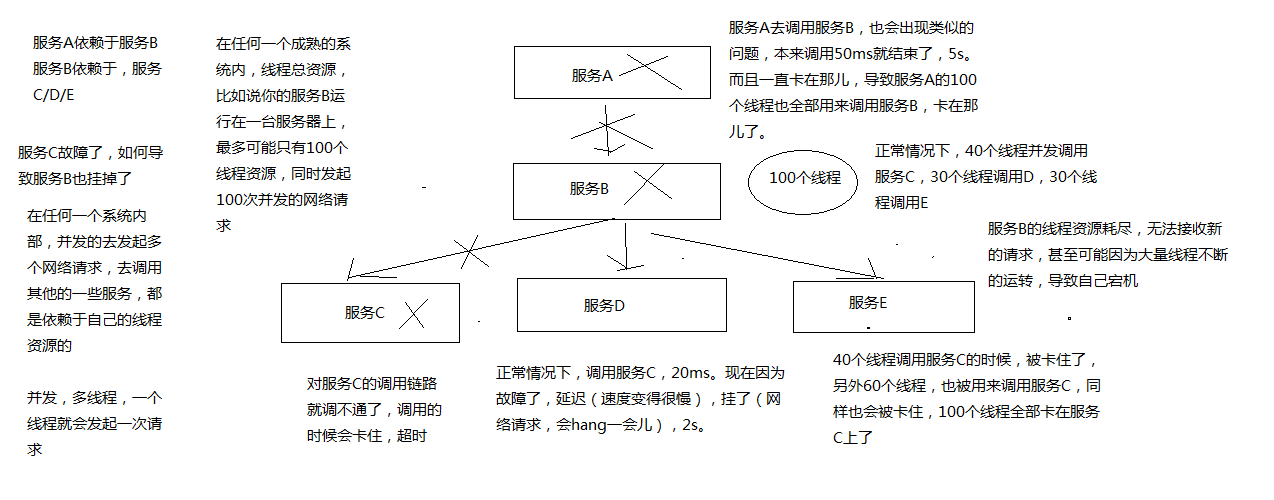
其中导致公司损失最大的负责人，受到了很大的处分

#### 85\_hystrix要解决的分布式系统可用性问题以及其设计原则

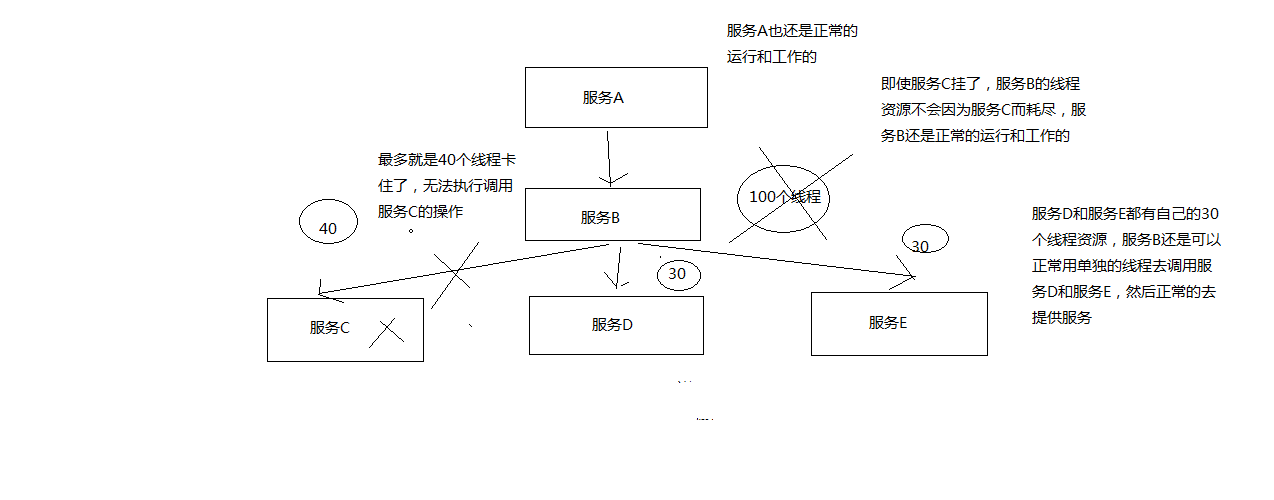
什么是分布式系统以及其中的故障和hystrix



依赖服务的故障导致服务被拖垮以及故障的蔓延



资源隔离如何保护依赖服务的故障不要拖垮整个系统



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

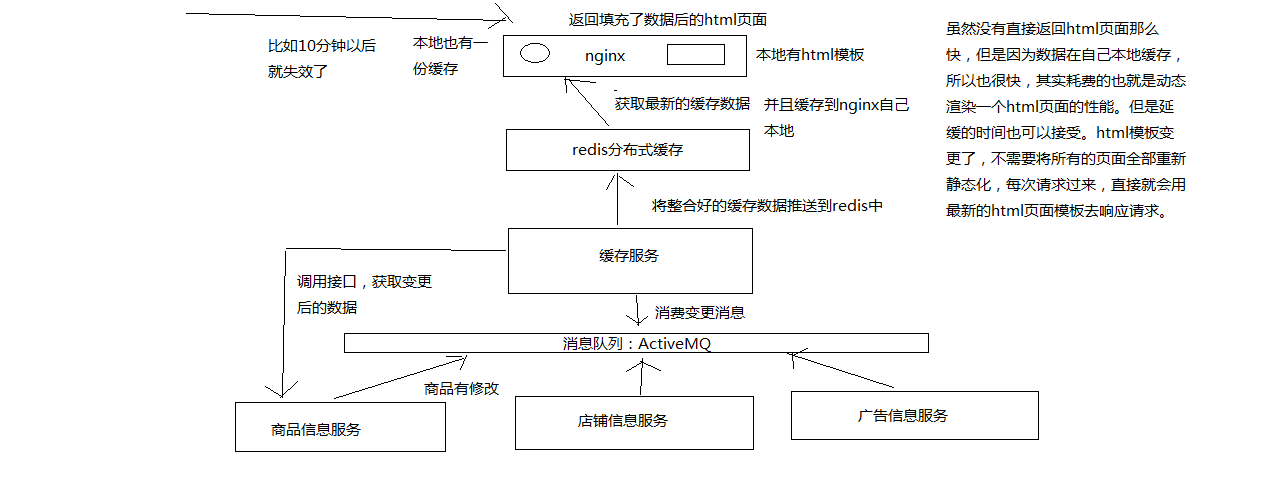
13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

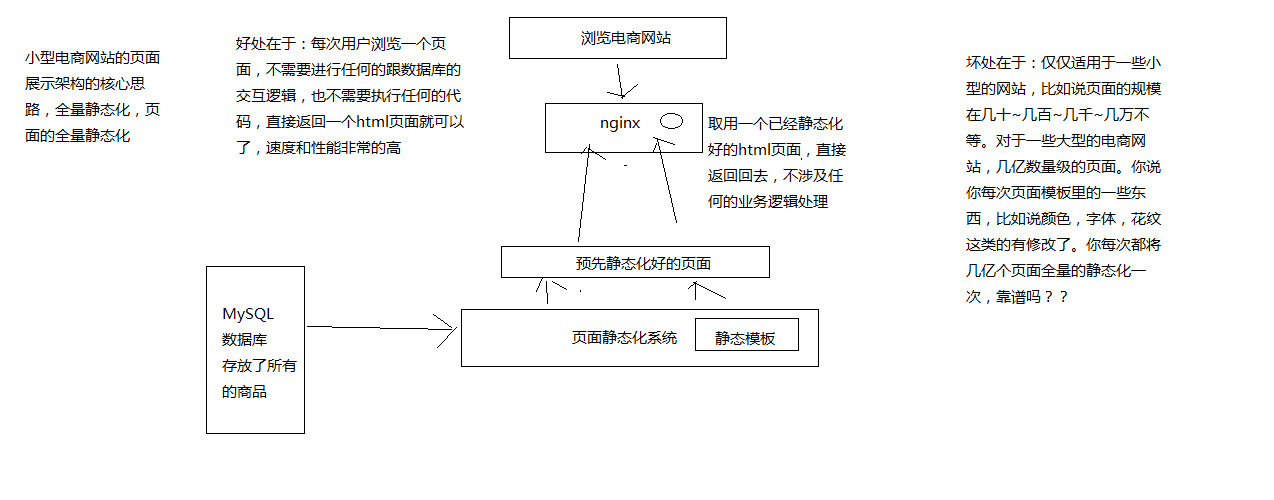
15、高并发场景下的缓存失效解决方案

#### 86\_电商网站的商品详情页缓存服务业务背景以及框架结构说明

大型电商网站的详情页系统的架构



小型电商网站的静态化方案



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

#### 87\_基于spring boot快速构建缓存服务以及商品服务

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

#### 88\_快速完成缓存服务接收数据变更消息以及调用商品服务接口的代码编写

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

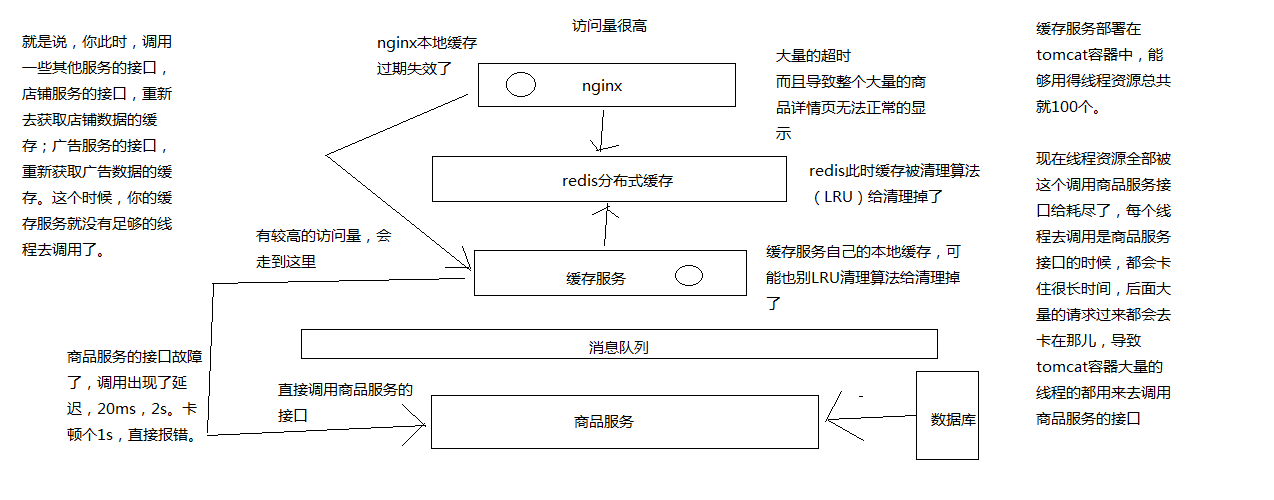
13、基于hystrix的大规模缓存雪崩解决方案

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

#### 89\_商品服务接口故障导致的高并发访问耗尽缓存服务资源的场景分析

商品服务接口导致缓存服务资源耗尽的问题



1、商品服务接口调用故障，导致缓存服务资源耗尽

2、hystrix针对一个一个的具体的业务场景，去开发高可用的架构

**《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

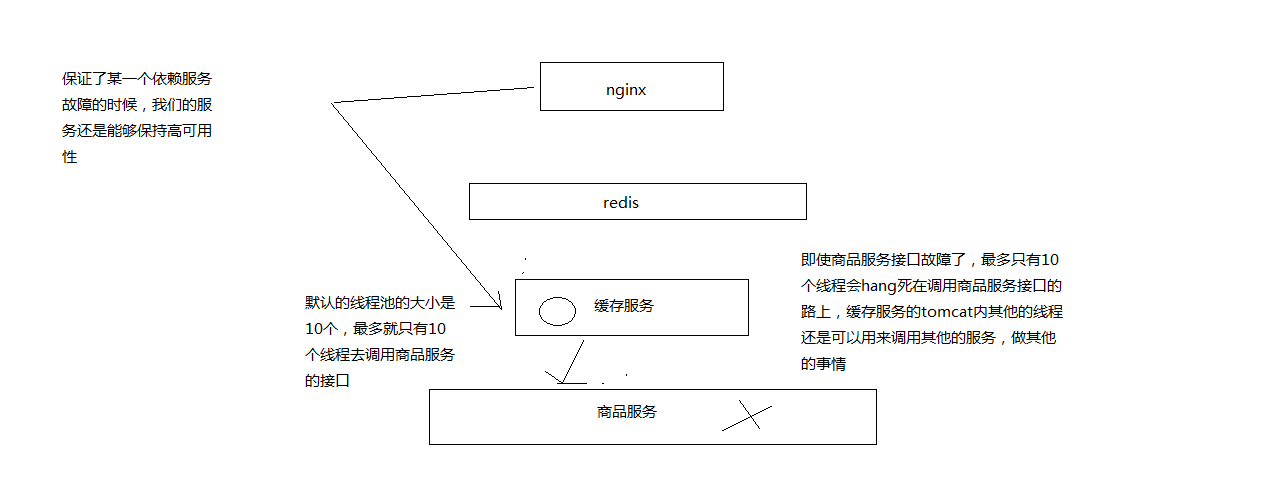
13、基于hystrix的大规模缓存雪崩解决方案

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

#### 90\_基于hystrix的线程池隔离技术进行商品服务接口的资源隔离

资源隔离生效的讲解



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

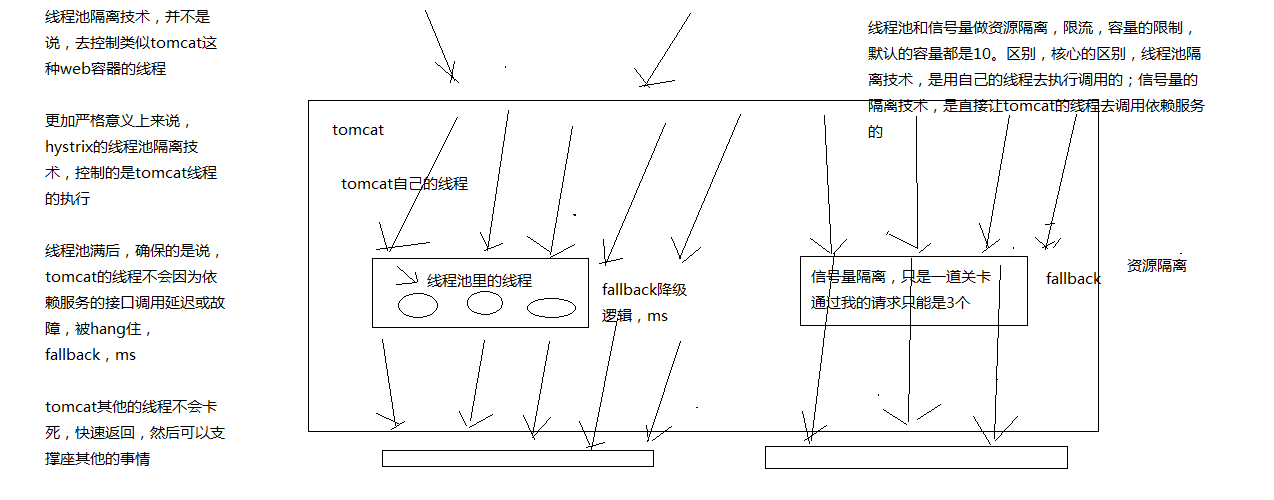
13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

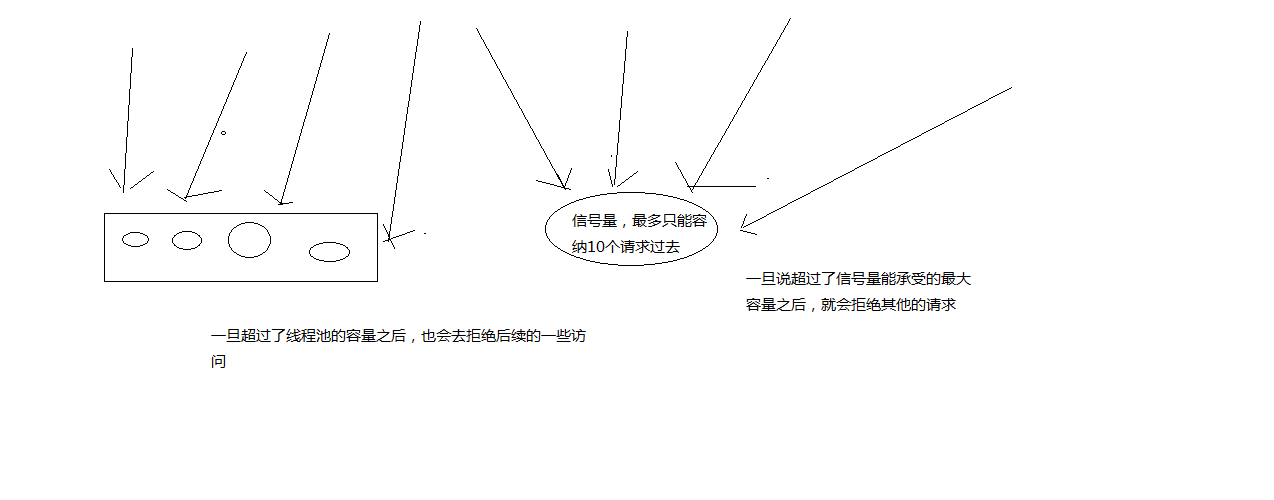
15、高并发场景下的缓存失效解决方案

#### 91\_基于hystrix的信号量技术对地理位置获取逻辑进行资源隔离与限流

线程池隔离和信号量隔离的原理以及区别



信号量的资源隔离与限流的说明



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

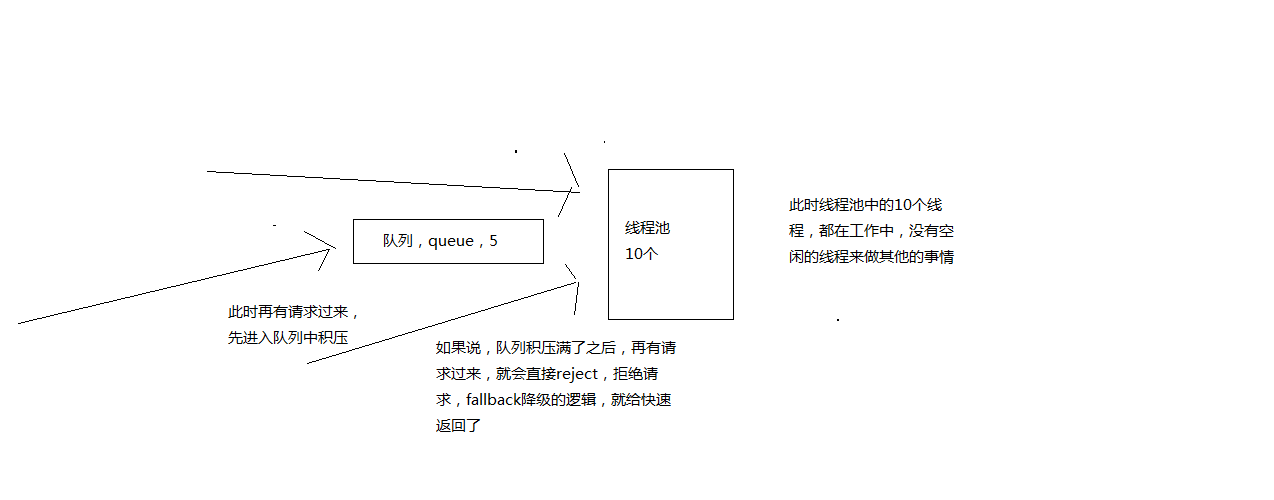
13、基于hystrix的大规模缓存雪崩解决方案

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

#### 92\_hystrix的线程池+服务+接口划分以及资源池的容量大小控制

线程池+queue的工作原理



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

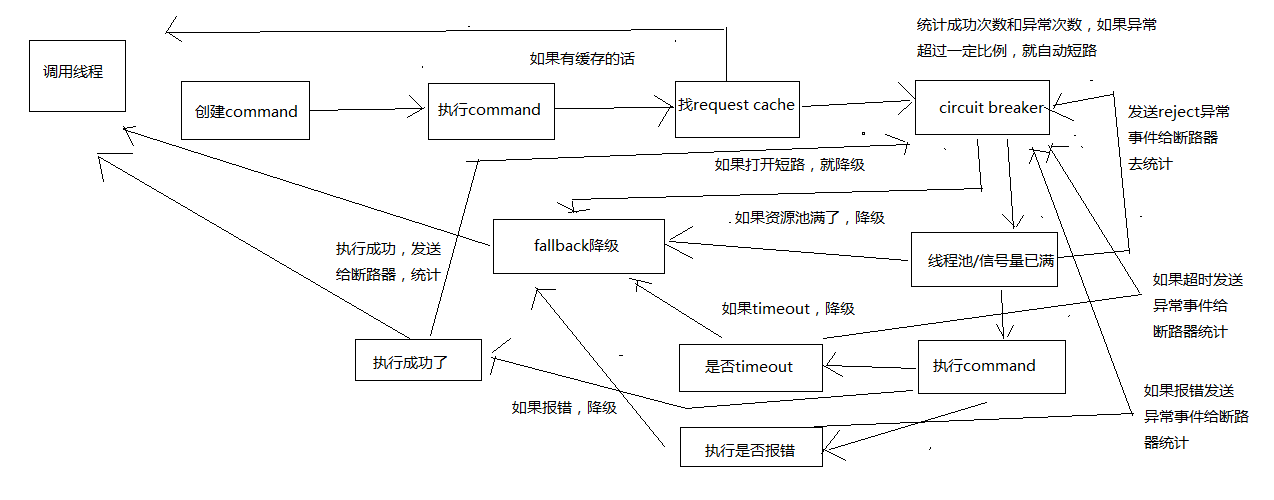
13、基于hystrix的大规模缓存雪崩解决方案

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

#### 93\_深入分析hystrix执行时的8大流程步骤以及内部原理

hystrix执行时的8大流程以及内部原理



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

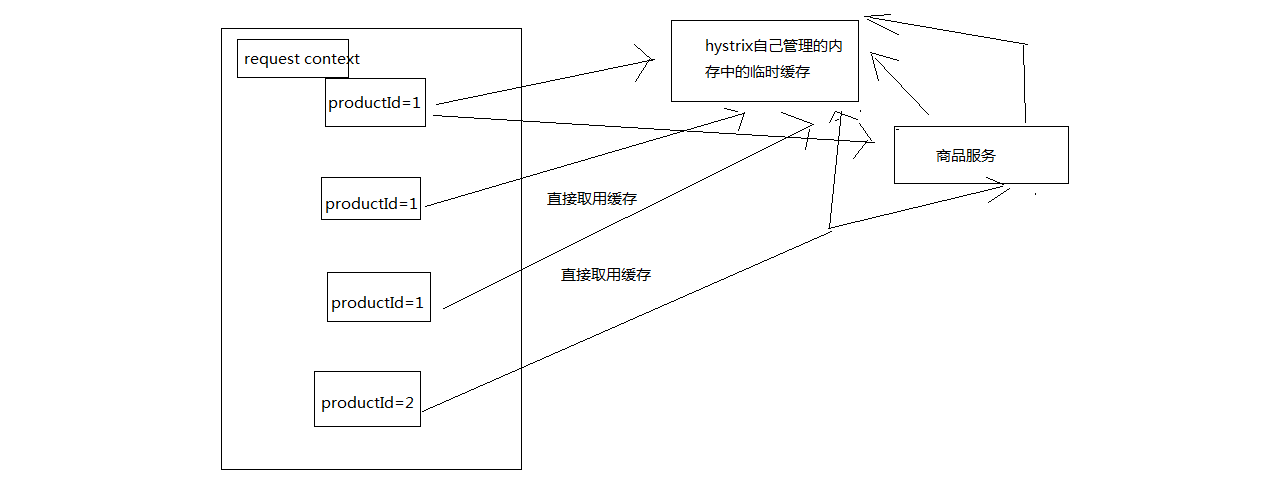
13、基于hystrix的大规模缓存雪崩解决方案

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

#### 94\_基于request cache请求缓存技术优化批量商品数据查询接口

request cache的原理



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

#### 95\_开发品牌名称获取接口的基于本地缓存的fallback降级机制

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

#### 96\_深入理解hystrix的短路器执行原理以及模拟接口异常时的短路实验

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

#### 97\_深入理解线程池隔离技术的设计原则以及动手实战接口限流实验

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

#### 98\_基于timeout机制来为商品服务接口的调用超时提供安全保护

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

#### 99\_基于hystrix的高可用分布式系统架构项目实战课程的总结

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

#### 100\_基于request collapser请求合并技术进一步优化批量查询

hystrix，高级的技术，request collapser，请求合并技术，collapser折叠

优化过一个批量查询的接口了，request cache来做优化，可能有相同的商品就可以直接取用缓存了

多个商品，需要发送多次网络请求，调用多次接口，才能拿到结果

可以使用HystrixCollapser将多个HystrixCommand合并到一起，多个command放在一个command里面去执行，发送一次网络请求，就拉取到多条数据

用请求合并技术，将多个请求合并起来，可以减少高并发访问下需要使用的线程数量以及网络连接数量，这都是hystrix自动进行的

其实对于高并发的访问来说，是可以提升性能的

请求合并有很多种级别

（1）global context，tomcat所有调用线程，对一个依赖服务的任何一个command调用都可以被合并在一起，hystrix就传递一个HystrixRequestContext

（2）user request context，tomcat内某一个调用线程，将某一个tomcat线程对某个依赖服务的多个command调用合并在一起

（3）object modeling，基于对象的请求合并，如果有几百个对象，遍历后依次调用每个对象的某个方法，可能导致发起几百次网络请求，基于hystrix可以自动将对多个对象模型的调用合并到一起

请求合并技术的开销有多大

使用请求合并技术的开销就是导致延迟大幅度增加，因为需要一定的时间将多个请求合并起来

发送过来10个请求，每个请求本来大概是2ms可以返回，要把10个请求合并在一个command内，统一一起执行，先后等待一下，5ms

所以说，要考量一下，使用请求合并技术是否合适，如果一个请求本来耗费的时间就比较长，那么进行请求合并，增加一些延迟影响并不大

请求合并技术，不是针对那种访问延时特别低的请求的，比如说你的访问延时本身就比较高，20ms，10个请求合并在一起，25ms，这种情况下就还好

好处在哪里，大幅度削减你的线程池的资源耗费，线程池，10个线程，一秒钟可以执行10个请求，合并在一起，1个线程执行10个请求，10个线程就可以执行100个请求

增加你的吞吐量

减少你对后端服务访问时的网络资源的开销，10个请求，10个command，10次网络请求的开销，1次网络请求的开销了

每个请求就2ms，batch，8~10ms，延迟增加了4~5倍

每个请求本来就30ms~50ms，batch，35ms~55ms，延迟增加不太明显

将多个command请求合并到一个command中执行

请求合并时，可以设置一个batch size，以及elapsed time（控制什么时候触发合并后的command执行）

有两种合并模式，一种是request scope，另一种是global scope，默认是rquest scope，在collapser构造的时候指定scope模式

request scope的batch收集是建立在一个request context内的，而global scope的batch收集是横跨多个request context的

所以对于global context来说，必须确保能在一个command内处理多个requeset context的请求

在netflix，是只用request scope请求合并的，因为默认是用唯一一个request context包含所有的command，所以要做合并，肯定就是request scope

一般请求合并技术，对于那种访问同一个资源的command，但是参数不同，是很有效的

批量查询，HystrixObservableCommand，HystrixCommand+request cache，都是每个商品发起一次网络请求

一个批量的商品过来以后，我们还是多个command的方式去执行，request collapser+request cache，相同的商品还是就查询一次，不同的商品合并到一起通过一个网络请求得到结果

timeout问题解释：开发机上，特别慢，第一次请求的时候，几百毫秒，默认的timeout时长比较短

第二次的时候，访问的速度会快很多，就不会超时了

反应在系统上，第一次启动的时候，会有个别的超时，但是后面就好了，手动将timeout时长设置的大一些

（1）maxRequestsInBatch

控制一个Batch中最多允许多少个request被合并，然后才会触发一个batch的执行

默认值是无限大，就是不依靠这个数量来触发执行，而是依靠时间

HystrixCollapserProperties.Setter()

.withMaxRequestsInBatch(int value)

（2）timerDelayInMilliseconds

控制一个batch创建之后，多长时间以后就自动触发batch的执行，默认是10毫秒

HystrixCollapserProperties.Setter()

.withTimerDelayInMilliseconds(int value)

super(Setter.withCollapserKey(HystrixCollapserKey.Factory.asKey("GetProductInfosCollapser"))

.andCollapserPropertiesDefaults(HystrixCollapserProperties.Setter()

.withMaxRequestsInBatch(100)

.withTimerDelayInMilliseconds(20)));

#### 101\_hystirx的fail-fast与fail-silient两种最基础的容错模式

fail-fast，就是不给fallback降级逻辑，HystrixCommand.run()，直接报错，直接会把这个报错抛出来，给你的tomcat调用线程

fail-silent，给一个fallback降级逻辑，如果HystrixCommand.run()，报错了，会走fallback降级，直接返回一个空值，HystrixCommand，就给一个null

HystrixObservableCommand，Observable.empty()

很少会用fail-fast模式，比较常用的可能还是fail-silent，特别常用，既然都到了fallback里面，肯定要做点降级的事情

#### 102\_为商品服务接口调用增加stubbed fallback降级机制

stubbed fallback，残缺的降级

用请求中的部分数据拼装成结果，然后再填充一些默认值，返回

比如说你发起了一个请求，然后请求中可能本身就附带了一些信息，如果主请求失败了，走到降级逻辑

在降级逻辑里面，可以将这个请求中的数据，以及部分本地缓存有的数据拼装在一起，再给数据填充一些简单的默认值

然后尽可能将自己有的数据返回到请求方

stubbed，残缺了，比如说应该查询到一个商品信息，里面包含20个字段

请求参数搂出来一两个字段，从本地的少量缓存中比如说，可以搂出来那么两三个字段，最终的话返回的字段可能就五六个，其他的字段都是填充的默认值

数据有残缺

我们主要是演示一下这种模式的使用，你硬要我拿最真实的业务和代码去演示，不可能的

公司真实的项目，真实的业务代码，都极其的复杂

我做过的真实的项目，简化，抽象，仿真，拿出来模拟的业务场景，给大家来讲解

效果肯定是比你纯粹一点业务都没有，全都是一些最基础的demo代码，业务的feel，是怎么用的s

在你自己的项目里去用的话，你就必须结合你自己的业务场景，去思考，stubbed fallback，从请求参数里尽可能提取一些数据，请求参数多给一些

你要考虑到可以将哪些量比较少的数据保存在内存中，提取部分数据

默认的值怎么设置，看起来能稍微靠谱一些

#### 103\_基于双层嵌套command开发商品服务接口的多级降级机制

多级降级

先降一级，尝试用一个备用方案去执行，如果备用方案失败了，再用最后下一个备用方案去执行

command嵌套command

尝试从备用服务器接口去拉取结果

给大家科普一下，常见的多级降级的做法，有一个操作，要访问MySQL数据库

mysql数据库访问报错，降级，去redis中获取数据

如果说redis又挂了，然后就去从本地ehcache缓存中获取数据

hystrix command fallback语义，很容易就可以实现多级降级的策略

商品服务接口，多级降级的策略

command，fallback，又套了一个command，第二个command其实是第一级降级策略

第二个command的fallback是第二级降级策略

第一级降级策略，可以是

storm，我们之前做storm这块，第一级降级，一般是搞一个storm的备用机房，部署了一套一模一样的拓扑，如果主机房中的storm拓扑挂掉了，备用机房的storm拓扑定顶上

如果备用机房的storm拓扑也挂了

第二级降级，可能就降级成用mysql/hbase/redis/es，手工封装的一套，按分钟粒度去统计数据的系统

第三季降级，离线批处理去做，hdfs+spark，每个小时执行一次数据统计，去降级

特别复杂，重要的系统，肯定是要搞好几套备用方案的，一个方案死了，立即上第二个方案，而且要尽量做到是自动化的

商品接口拉取

主流程，访问的商品服务，是从主机房去访问的，服务，如果主机房的服务出现了故障，机房断电，机房的网络负载过高，机器硬件出了故障

第一级降级策略，去访问备用机房的服务

第二级降级策略，用stubbed fallback降级策略，比较常用的，返回一些残缺的数据回去

#### 104\_基于facade command开发商品服务接口的手动降级机制

手动降级

你写一个command，在这个command它的主流程中，根据一个标识位，判断要执行哪个流程

可以执行主流程，command，也可以执行一个备用降级的command

一般来说，都是去执行一个主流程的command，如果说你现在知道有问题了，希望能够手动降级的话，动态给服务发送个请求

在请求中修改标识位，自动就让command以后都直接过来执行备用command

3个command，套在最外面的command，是用semaphore信号量做限流和资源隔离的，因为这个command不用去care timeout的问题，嵌套调用的command会自己去管理timeout超时的

商品服务接口的手动降级的方案

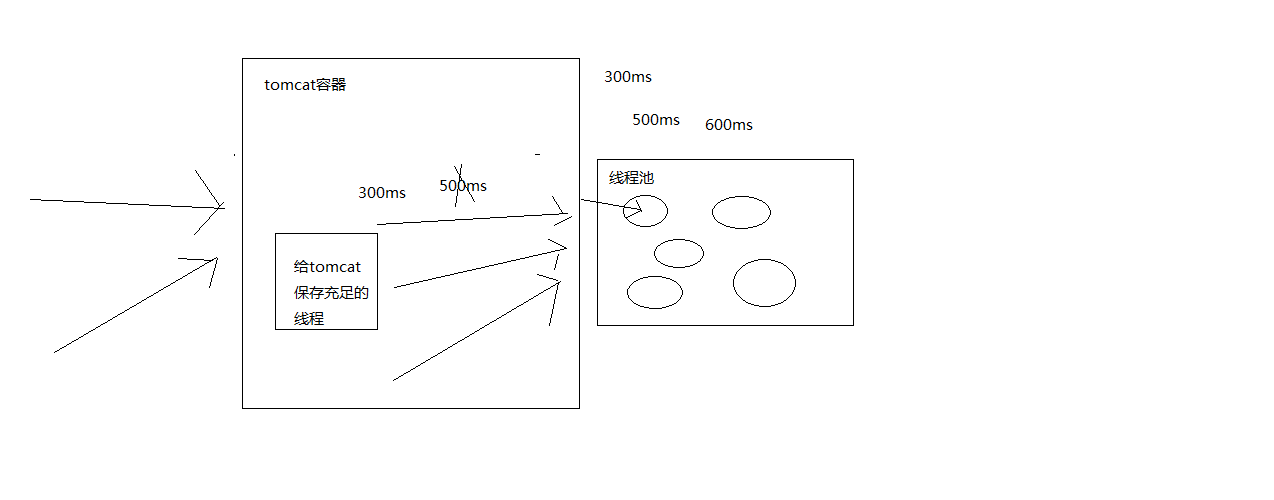
主流程还是去走GetProductInfoCommand，手动降级的方案，比如说是从某一个数据源，自己去简单的获取一些数据，尝试封装一下返回

手动降级的策略，就比较low了，调用别人的接口去获取数据的，业务逻辑的封装

主流程有问题，那么可能你就需要立即自己写一些逻辑发布上去，从mysql数据库的表中获取一些数据去返回，手动调整一下降级标识，做一下手动降级

#### 105\_生产环境中的线程池大小以及timeout超时时长优化经验总结

timeout和线程池的超时关系



生产环境里面，一个是线程池的大小怎么设置，timeout时长怎么

不合理的话，问题还是很大的

在生产环境中部署一个短路器，一开始需要将一些关键配置设置的大一些，比如timeout超时时长，线程池大小，或信号量容量

然后逐渐优化这些配置，直到在一个生产系统中运作良好

（1）一开始先不要设置timeout超时时长，默认就是1000ms，也就是1s

（2）一开始也不要设置线程池大小，默认就是10

（3）直接部署hystrix到生产环境，如果运行的很良好，那么就让它这样运行好了

（4）让hystrix应用，24小时运行在生产环境中

（5）依赖标准的监控和报警机制来捕获到系统的异常运行情况

（6）在24小时之后，看一下调用延迟的占比，以及流量，来计算出让短路器生效的最小的配置数字

（7）直接对hystrix配置进行热修改，然后继续在hystrix dashboard上监控

（8）看看修改配置后的系统表现有没有改善

下面是根据系统表现优化和调整线程池大小，队列大小，信号量容量，以及timeout超时时间的经验

假设对一个依赖服务的高峰调用QPS是每秒30次

一开始如果默认的线程池大小是10

我们想的是，理想情况下，每秒的高峰访问次数 \* 99%的访问延时 + buffer = 30 \* 0.2 + 4 = 10线程，10个线程每秒处理30次访问应该足够了，每个线程处理3次访问

此时，我们合理的timeout设置应该为300ms，也就是99.5%的访问延时，计算方法是，因为判断每次访问延时最多在250ms（TP99如果是200ms的话），再加一次重试时间50ms，就是300ms，感觉也应该足够了

因为如果timeout设置的太多了，比如400ms，比如如果实际上，在高峰期，还有网络情况较差的时候，可能每次调用要耗费350ms，也就是达到了最长的访问时长

那么每个线程处理2个请求，就会执行700ms，然后处理第三个请求的时候，就超过1秒钟了，此时会导致线程池全部被占满，都在处理请求

这个时候下一秒的30个请求再进来了，那么就会导致线程池已满，拒绝请求的情况，就会调用fallback降级机制

因此对于短路器来说，timeout超时一般应该设置成TP99.5，比如设置成300ms，那么可以确保说，10个线程，每个线程处理3个访问，每个访问最多就允许执行300ms，过时就timeout了

这样才能保证说每个线程都在1s内执行完，才不会导致线程池被占满，然后后续的请求过来大量的reject

对于线程池大小来说，一般应该控制在10个左右，20个以内，最少5个，不要太多，也不要太少

大家可能会想，每秒的高峰访问次数是30次，如果是300次，甚至是3000次，30000次呢？？？

30000 \* 0.2 = 6000 + buffer = 6100，一个服务器内一个线程池给6000个线程把

如果你一个依赖服务占据的线程数量太多的话，会导致其他的依赖服务对应的线程池里没有资源可以用了

6000 / 20 = 300台虚拟机也是ok的

虚拟机，4个cpu core，4G内存，虚拟机，300台

物理机，十几个cpu core，几十个G的内存，5~8个虚拟机，300个虚拟机 = 50台物理机

你要真的说是，你的公司服务的用户量，或者数据量，或者请求量，真要是到了每秒几万的QPS，

3万QPS，60 \* 3 = 180万访问量，1800，1亿8千，1亿，10个小时，10亿的访问量，app，系统

几十台服务器去支撑，我觉得很正常

QPS每秒在几千都算多的了

#### 106\_生产环境中的线程池自动扩容与缩容的动态资源分配经验

可能会出现一种情况，比如说我们的某个依赖，在高峰期，需要耗费100个线程，但是在那个时间段，刚好其他的依赖的线程池其实就维持一两个就可以了

但是，如果我们都是设置死的，每个服务就给10个线程，那就很坑，可能就导致有的服务在高峰期需要更多的资源，但是没资源了，导致很多的reject

但是其他的服务，每秒钟就易一两个请求，结果也占用了10个线程，占着茅坑不拉屎

做成弹性的线程资源调度的模式

刚开始的时候，每个依赖服务都是给1个线程，3个线程，但是我们允许说，如果你的某个线程池突然需要大量的线程，最多可以到100个线程

如果你使用了100个线程，高峰期过去了，自动将空闲的线程给释放掉

（1）coreSize

设置线程池的大小，默认是10

HystrixThreadPoolProperties.Setter()

.withCoreSize(int value)

（2）maximumSize

设置线程池的最大大小，只有在设置allowMaximumSizeToDivergeFromCoreSize的时候才能生效

默认是10

HystrixThreadPoolProperties.Setter()

.withMaximumSize(int value)

（5）keepAliveTimeMinutes

设置保持存活的时间，单位是分钟，默认是1

如果设置allowMaximumSizeToDivergeFromCoreSize为true，那么coreSize就不等于maxSize，此时线程池大小是可以动态调整的，可以获取新的线程，也可以释放一些线程

如果coreSize < maxSize，那么这个参数就设置了一个线程多长时间空闲之后，就会被释放掉

HystrixThreadPoolProperties.Setter()

.withKeepAliveTimeMinutes(int value)

（6）allowMaximumSizeToDivergeFromCoreSize

允许线程池大小自动动态调整，设置为true之后，maxSize就生效了，此时如果一开始是coreSize个线程，随着并发量上来，那么就会自动获取新的线程，但是如果线程在keepAliveTimeMinutes内空闲，就会被自动释放掉

默认是fales

HystrixThreadPoolProperties.Setter()

.withAllowMaximumSizeToDivergeFromCoreSize(boolean value)

生产环境中，这块怎么玩儿的

也是根据你的服务的实际的运行的情况切看的，比如说你发现某个服务，平时3个并发QPS就够了，高峰期可能要到30个

那么你就可以给设置弹性的资源调度

因为你可能一个服务会有多个线程池，你要计算好，每个线程池的最大的大小加起来不能过大，30个依赖，30个线程池，每个线程池最大给到30,900个线程，很坑的

还有一种模式，就是说让多个依赖服务共享一个线程池，我们不推荐，多个依赖服务就做不到资源隔离，互相之间会影响的

1，coreSize

#### 107\_hystrix的metric统计相关的各种高阶配置讲解

1、为什么需要监控与报警？

HystrixCommand执行的时候，会生成一些执行耗时等方面的统计信息。这些信息对于系统的运维来说，是很有帮助的，因为我们通过这些统计信息可以看到整个系统是怎么运行的。hystrix对每个command key都会提供一份metric，而且是秒级统计粒度的。

这些统计信息，无论是单独看，还是聚合起来看，都是很有用的。如果将一个请求中的多个command的统计信息拿出来单独查看，包括耗时的统计，对debug系统是很有帮助的。聚合起来的metric对于系统层面的行为来说，是很有帮助的，很适合做报警或者报表。hystrix dashboard就很适合。

2、hystrix的事件类型

对于hystrix command来说，只会返回一个值，execute只有一个event type，fallback也只有一个event type，那么返回一个SUCCESS就代表着命令执行的结束

对于hystrix observable command来说，多个值可能被返回，所以emit event代表一个value被返回，success代表成功，failure代表异常

（1）execute event type

EMIT observable command返回一个value

SUCCESS 完成执行，并且没有报错

FAILURE 执行时抛出了一个异常，会触发fallback

TIMEOUT 开始执行了，但是在指定时间内没有完成执行，会触发fallback

BAD\_REQUEST 执行的时候抛出了一个HystrixBadRequestException

SHORT\_CIRCUITED 短路器打开了，触发fallback

THREAD\_POOL\_REJECTED 线程成的容量满了，被reject，触发fallback

SEMAPHORE\_REJECTED 信号量的容量满了，被reject，触发fallback

（2）fallback event type

FALLBACK\_EMIT observable command，fallback value被返回了

FALLBACK\_SUCCESS fallback逻辑执行没有报错

FALLBACK\_FAILURE fallback逻辑抛出了异常，会报错

FALLBACK\_REJECTION fallback的信号量容量满了，fallback不执行，报错

FALLBACK\_MISSING fallback没有实现，会报错

（3）其他的event type

EXCEPTION\_THROWN command生命自周期是否抛出了异常

RESPONSE\_FROM\_CACHE command是否在cache中查找到了结果

COLLAPSED command是否是一个合并batch中的一个

（4）thread pool event type

EXECUTED 线程池有空间，允许command去执行了

REJECTED 线程池没有空间，不允许command执行，reject掉了

（5）collapser event type

BATCH\_EXECUTED collapser合并了一个batch，并且执行了其中的command

ADDED\_TO\_BATCH command加入了一个collapser batch

RESPONSE\_FROM\_CACHE 没有加入batch，而是直接取了request cache中的数据

3、metric storage

metric被生成之后，就会按照一段时间来存储，存储了一段时间的数据才会推送到其他系统中，比如hystrix dashboard

另外一种方式，就是每次生成metric就实时推送metric流到其他地方，但是这样的话，会给系统带来很大的压力

hystrix的方式是将metric写入一个内存中的数据结构中，在一段时间之后就可以查询到

hystrix 1.5x之后，采取的是为每个command key都生成一个start event和completion event流，而且可以订阅这个流。每个thread pool key也是一样的，包括每个collapser key也是一样的。

每个command的event是发送给一个线程安全的RxJava中的rx.Subject，因为是线程安全的，所以不需要进行线程同步

因此每个command级别的，threadpool级别的，每个collapser级别的，event都会发送到对应的RxJava的rx.Subject对象中。这些rx.Subject对象接着就会被暴露出Observable接口，可以被订阅。

5、metric统计相关的配置

（1）metrics.rollingStats.timeInMilliseconds

设置统计的rolling window，单位是毫秒，hystrix只会维持这段时间内的metric供短路器统计使用

这个属性是不允许热修改的，默认值是10000，就是10秒钟

HystrixCommandProperties.Setter()

.withMetricsRollingStatisticalWindowInMilliseconds(int value)

（2）metrics.rollingStats.numBuckets

该属性设置每个滑动窗口被拆分成多少个bucket，而且滑动窗口对这个参数必须可以整除，同样不允许热修改

默认值是10，也就是说，每秒钟是一个bucket

随着时间的滚动，比如又过了一秒钟，那么最久的一秒钟的bucket就会被丢弃，然后新的一秒的bucket会被创建

HystrixCommandProperties.Setter()

.withMetricsRollingStatisticalWindowBuckets(int value)

（3）metrics.rollingPercentile.enabled

控制是否追踪请求耗时，以及通过百分比方式来统计，默认是true

HystrixCommandProperties.Setter()

.withMetricsRollingPercentileEnabled(boolean value)

（4）metrics.rollingPercentile.timeInMilliseconds

设置rolling window被持久化保存的时间，这样才能计算一些请求耗时的百分比，默认是60000，60s，不允许热修改

相当于是一个大的rolling window，专门用于计算请求执行耗时的百分比

HystrixCommandProperties.Setter()

.withMetricsRollingPercentileWindowInMilliseconds(int value)

（5）metrics.rollingPercentile.numBuckets

设置rolling percentile window被拆分成的bucket数量，上面那个参数除以这个参数必须能够整除，不允许热修改

默认值是6，也就是每10s被拆分成一个bucket

HystrixCommandProperties.Setter()

.withMetricsRollingPercentileWindowBuckets(int value)

（6）metrics.rollingPercentile.bucketSize

设置每个bucket的请求执行次数被保存的最大数量，如果再一个bucket内，执行次数超过了这个值，那么就会重新覆盖从bucket的开始再写

举例来说，如果bucket size设置为100，而且每个bucket代表一个10秒钟的窗口，但是在这个bucket内发生了500次请求执行，那么这个bucket内仅仅会保留100次执行

如果调大这个参数，就会提升需要耗费的内存，来存储相关的统计值，不允许热修改

默认值是100

HystrixCommandProperties.Setter()

.withMetricsRollingPercentileBucketSize(int value)

（7）metrics.healthSnapshot.intervalInMilliseconds

控制成功和失败的百分比计算，与影响短路器之间的等待时间，默认值是500毫秒

HystrixCommandProperties.Setter()

.withMetricsHealthSnapshotIntervalInMilliseconds(int value)

#### 108\_hystrix dashboard可视化分布式系统监控环境部署

1、安装metrics stream

<dependency>

<groupId>com.netflix.hystrix</groupId>

<artifactId>hystrix-metrics-event-stream</artifactId>

<version>1.4.10</version>

</dependency>

@Bean

public ServletRegistrationBean indexServletRegistration() {

ServletRegistrationBean registration = new ServletRegistrationBean(new HystrixMetricsStreamServlet());

registration.addUrlMappings("/hystrix.stream");

return registration;

}

2、安装gradle

类似于maven，一种java里面的打包和构建的工具，hystrix是用gradle去管理打包和构建的

配置环境变量，GRADLE\_HOME

配置PATH，%GRADLE\_HOME%/bin

gradle -v

3、下载tomcat7

解压缩

4、下载hystrix-dashboard的war包

cp hystrix-dashboard-\*.war apache-tomcat-7.\*/webapps/hystrix-dashboard.war

5、下载turbin

下载并解压缩

cp turbine-web/build/libs/turbine-web-\*.war ./apache-tomcat-7.\*/webapps/turbine.war

在/WEB-INF/classes下放置配置文件

config.properties

turbine.ConfigPropertyBasedDiscovery.default.instances=localhost

turbine.instanceUrlSuffix=:8081/hystrix.stream

turbin是用来监控一个集群的，可以将一个集群的所有机器都配置在这里

6、启动我们的服务

7、启动tomcat中的hystrix dashboard和turbin

localhost:8080/hystrix-dashboard

http://localhost:8081/hystrix.stream，监控单个机器

http://localhost:8080/turbine/turbine.stream，监控整个集群

8、发送几个请求，看看效果

9、hystrix dashboard

hystrix的dashboard可以支持实时监控metric

netflix开始用这个dashboard的时候，大幅度优化了工程运维的操作，帮助节约了恢复系统的时间。大多数生产系统的故障持续时间变得很短，而且影响幅度小了很多，主要是因为hystrix dashborad提供了可视化的监控。

截图说明，dashboard上的指标都是什么？

圆圈的颜色和大小代表了健康状况以及流量，折线代表了最近2分钟的请求流量

集群中的机器数量，请求延时的中位数以及平均值

最近10秒内的异常请求比例，请求QPS，每台机器的QPS，以及整个集群的QPS

断路器的状态

最近一分钟的请求延时百分比，TP90，TP99，TP99.5

几个有颜色的数字，代表了最近10秒钟的统计，以1秒钟为粒度

成功的请求数量，绿颜色的数字; 短路的请求数量，蓝色的数字; timeout超时的请求数量，黄色的数字; 线程池reject的请求数量，紫色的数字; 请求失败，抛出异常的请求数量，红色的数字

#### 109\_生产环境中的hystrix分布式系统的工程运维经验总结

如果发现了严重的依赖调用延时，先不用急着去修改配置，如果一个command被限流了，可能本来就应该限流

在netflix早期的时候，经常会有人在发现短路器因为访问延时发生的时候，去热修改一些皮遏制，比如线程池大小，队列大小，超时时长，等等，给更多的资源，但是这其实是不对的

如果我们之前对系统进行了良好的配置，然后现在在高峰期，系统在进行线程池reject，超时，短路，那么此时我们应该集中精力去看底层根本的原因，而不是调整配置

为什么在高峰期，一个10个线程的线程池，搞不定这些流量呢？？？代码写的太烂了，异步，更好的算法

千万不要急于给你的依赖调用过多的资源，比如线程池大小，队列大小，超时时长，信号量容量，等等，因为这可能导致我们自己对自己的系统进行DDOS攻击

疯狂的大量的访问你的机器，最后给打垮

举例来说，想象一下，我们现在有100台服务器组成的集群，每台机器有10个线程大小的线程池去访问一个服务，那么我们对那个服务就有1000个线程资源去访问了

在正常情况下，可能只会用到其中200~300个线程去访问那个后端服务

但是如果再高峰期出现了访问延时，可能导致1000个线程全部被调用去访问那个后端服务，如果我们调整到每台服务器20个线程呢？

如果因为你的代码等问题导致访问延时，即使有20个线程可能还是会导致线程池资源被占满，此时就有2000个线程去访问后端服务，可能对后端服务就是一场灾难

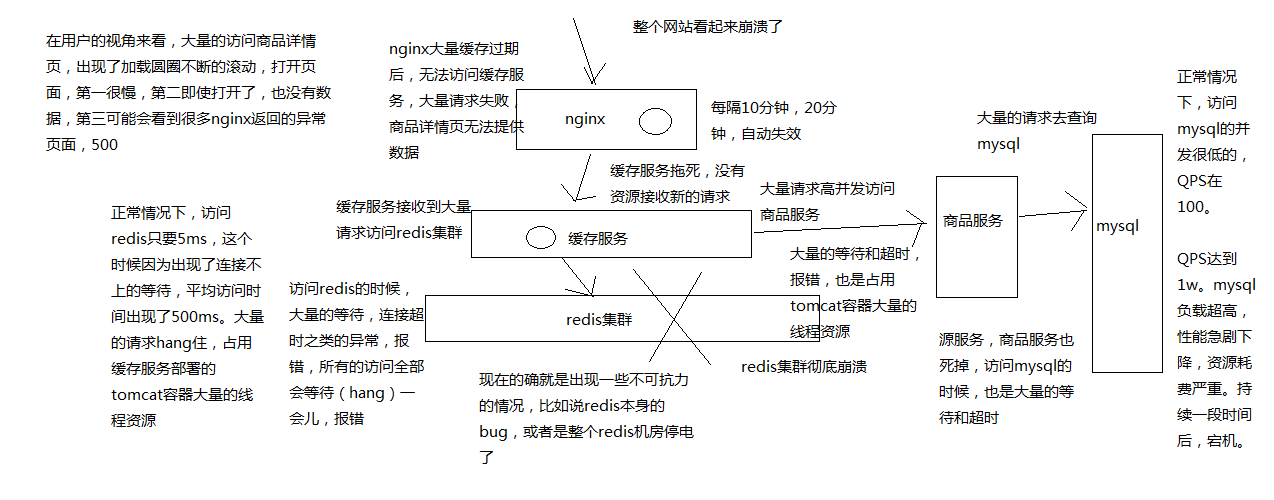
这就是断路器的作用了，如果我们把后端服务打死了，或者产生了大量的压力，有大量的timeout和reject，那么就自动短路，一段时间后，等流量洪峰过去了，再重启访问

简单来说，让系统自己去限流，短路，超时，以及reject，直到系统重新变得正常了

就是不要随便乱改资源配置，不要随便乱增加线程池大小，等待队列大小，异常情况是正常的

#### 110\_高并发场景下恐怖的缓存雪崩现象以及导致系统全盘崩溃的后果

缓存雪崩的过程和后果



缓存雪崩这种场景，缓存架构中非常重要的一个环节，应对缓存雪崩的解决方案，避免缓存雪崩的时候，造成整个系统崩溃，带来巨大的经济损失

1、redis集群彻底崩溃

2、缓存服务大量对redis的请求hang住，占用资源

3、缓存服务大量的请求打到源头服务去查询mysql，直接打死mysql

4、源头服务因为mysql被打死也崩溃，对源服务的请求也hang住，占用资源

5、缓存服务大量的资源全部耗费在访问redis和源服务无果，最后自己被拖死，无法提供服务

6、nginx无法访问缓存服务，redis和源服务，只能基于本地缓存提供服务，但是缓存过期后，没有数据提供

7、网站崩溃

行业里真实的缓存雪崩的经验和教训

某电商，之前就是出现过，整个缓存的集群彻底崩溃了，因为主要是集群本身的bug，导致自己把自己给弄死了，虽然当时也是部署了双机房的，但是还是死了

电商大量的，几乎所有的应用都是基于那个缓存集群去开发的

导致各种服务的线程资源全部被耗尽，然后用在了访问那个缓存集群时的等待、超时和报错上了

然后导致各种服务就没有资源对外提供服务咯

然后各种降级措施也没做好，直接就是整体系统的全盘崩溃

导致网站就没法对外出售商品咯，导致了很大数额的经济的损失

java架构师，资深java工程师，对自己技术有点要求，多学一些，多思考一些各种场景下的缓存架构，用来解决各种各样的问题

自己做系统架构设计的时候，多留个心眼儿，考虑一下各种高并发场景下可能出现的问题，数据不一致，热点缓存，重建并发冲突，redis高可用性，缓存雪崩， 缓存穿透，缓存失效

架构设计做好一些，稳定性也做好一些

你的系统能够承载各种各样的故障，才能在真正发生故障的时候，减少对公司的损失，保住大家的饭碗

你说你用过redis，系统里面涉及过这种缓存的架构，高并发场景下的各种问题，结合你的业务，怎么去设计整套缓存架构的，跳槽面试的时候，说点牛逼的出来

#### 111\_缓存雪崩的基于事前+事中+事后三个层次的完美解决方案

空

#### 112\_基于hystrix完成对redis访问的资源隔离以避免缓存服务被拖垮

这一讲开始，用几讲的时间，给咱们的redis的访问这一块，加上保护措施，给商品服务的访问加上限流的保护措施（重复，之前已经）

redis这一块，全都用hystrix的command进行封装，做资源隔离，确保说，redis的访问只能在固定的线程池内的资源来进行访问

哪怕是redis访问的很慢，有等待和超时，也不要紧，只有少量额线程资源用来访问，缓存服务不会被拖垮

#### 113\_为redis集群崩溃时的访问失败增加fail silent容错机制

上一节课，我们已经通过hystrix command对redis的访问进行了资源隔离

资源隔离，避免说redis访问频繁失败，或者频繁超时的时候，耗尽大量的tomcat容器的资源去hang在redis的访问上

限定只有一部分线程资源可以用来访问redis

你是不是说，如果redis集群彻底崩溃了，这个时候，可能command对redis的访问大量的报错和timeout超时，熔断（短路）

降级机制，fallback

fail silent模式，fallback里面直接返回一个空值，比如一个null，最简单了

在外面调用redis的代码（CacheService类），是感知不到redis的访问异常的，只要你把timeout、熔断、熔断恢复、降级，都做好了

可能会出现的情况是，当redis集群崩溃的时候，CacheService获取到的是大量的null空值

根据这个null空值，我们还可以去做多级缓存的降级访问，nginx本地缓存，redis分布式集群缓存，ehcache本地缓存，CacheController

#### 114\_为redis集群崩溃时的场景部署定制化的熔断策略

缓存雪崩的解决方案，事中，发生缓存雪崩的时候，解决方案

redis集群崩溃的时候，会怎么样？

（1）首先大量的等待，超时，报错

（2）如果是短时间内报错，会直接走fallback降级，直接返回null

（3）超时控制，你应该判断说redis访问超过了多长时间，就直接给timeout掉了

不推荐说用默认的值，一般不太精准，redis的访问你首先自己先统计一下访问时长的百分比，hystrix dashboard，TP90 TP95 TP99

一般来说，redis访问，假设说TP99在100ms，那么此时，你的timeout稍微多给一些，100ms

1、timeout超时控制

HystrixCommandProperties.Setter()

.withExecutionTimeoutInMilliseconds(int value)

意义在于哪里，一旦说redis出现了大面积的故障，此时肯定是访问的时候大量的超过100ms，大量的在等待和超时

就可以确保说，大量的请求不会hang住过长的时间，比如说hang住个1s，500ms，100ms直接就报timeout，走fallback降级了

2、熔断策略

（1）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)

我们应该根据我们自己的平时的访问流量去设置，而不是用默认值，比如说，我们认为平时一般的时候，流量也可以在每秒在QPS 100，10秒的滑动窗口就是1000

一般来说，你可以设置这样的一个值，根据你自己的系统的流量去设置

假如说，你设置的太少了，或者太多了，都不太合适

举个例子，你设置一个20，结果在晚上最低峰的时候，刚好是30，可能晚上的时候因为访问不频繁，大量的找不到缓存，可能超时频繁了一些，结果直接就给短路了

（2）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)

我们最好还是自己定制，自己设置，你说如果是要50%的时候才短路的话，会有什么情况呢，10%短路，也不太靠谱，90%异常，才短路

我觉得这个值可以稍微高一些，redis集群彻底崩溃，那么基本上就是所有的请求，100%都会异常，60%，70%

也有可能偶然出现网络的抖动，导致比如说就这10秒钟，访问延时高了一些，其实可能并不需要立即就短路，可能下10秒马上就恢复了

金融支付类的接口，可能这个比例就会设置的很低，因为对异常系统必须要很敏感，可能就是10%异常了，就直接短路了，不让继续访问了

比如金融支付类的接口，正常来说，是很重要的，而且必须是很稳定，我们不能容忍任何的延迟或者是报错

一旦支付类的接口，有10%的异常的话，我们基本就可以认为这个接口已经出问题了，再继续访问的话，也许访问的就是有问题的接口，可能造成资金的错乱，等给公司造成损失

熔断，不让访问了，走降级策略

就是对整个系统，是一个安全性的保障

（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)

如果redis集群崩溃了，会在5s内就直接恢复，1分钟

#### 115\_基于hystrix限流完成源服务的过载保护以避免流量洪峰打死MySQL

redis集群彻底崩溃的时候，一个是对redis本身做资源隔离、超时控制、熔断策略

大量的请求，高并发会去访问源服务，商品服务（提供商品数据），QPS 10000去访问商品服务，基于mysql去查询

QPS 10000去访问mysql，会怎么样，mysql打死，商品服务也会死掉

就是要对商品服务这种源服务的访问施加限流的措施

限流怎么限，hystrix本身就是提供了两种机制，线程池（内部做了异步化处理，可以处理超时），semaphore（信号量，让tomcat线程执行运行逻辑，没有内部的异步化处理，一旦超时，会导致tomcat线程就hang住了）

一般推荐的是，线程池用来做有网络访问的这种资源隔离，因为涉及到网络，就很容易超时；sempahore是用来做对服务纯内存的一些复杂业务逻辑的操作，进行限流，因为不涉及网络访问，就是纯粹为了避免说对内存内的复杂业务逻辑进行太高并发的访问，造成系统本身的故障

semaphore是很合适的，比如一些推荐、搜索，有部分算法，复杂的算法，是放在服务内部纯内存去运行的，一个服务暴露出来的就是某个算法的执行

这个时候，就很适合用semaphore

访问外部的商品服务，所以还是用线程池做限流了。。。

算一下，要限多少，怎么限

假设说，每次商品服务的访问性能在200ms，1个线程一秒可以执行5次访问，假设说我们一个缓存服务实例对这个商品服务的访问每秒在150次

所以这个时候，我们就需要30个线程，每个线程每秒可以访问5次，总共每秒30个线程可以访问150次

这个时候呢，我们限流，要做得事情是这样子的，我们算的这个每秒150次访问时正常情况下，如果是非正常情况下，每秒1000次，甚至1w次，此时就可以自然限流

因为我们的线程池就30个。。。，还要设置等待队列

非正常情况下，直接线程池+等待队列全满，此时就会会出现大量的reject操作，然后就会去调用降级逻辑

接下来，我们要做限流，设置的就是线程池的大小，还有等待队列的大小，30个线程可以每秒处理150个请求，但是偶尔会多一些出来，同时30个线程处理150个请求会快一些，不用花费1秒钟，等待队列给一些buffer，不要偶尔1秒钟来了200条请求，50条直接给reject掉，等待队列，150个，30个线程直接500ms处理完了，等待队列中的50个请求可以继续处理

#### 116\_为源头服务的限流场景增加stubbed fallback降级机制

我们上一讲讲到说，限流，计算了一下线程池的最大的大小，和这个等待队列，去限制了每秒钟最多能发送多少次请求到商品服务

避免大量的请求都发送到商品服务商去

限流过后，就会导致什么呢，比如redis集群崩溃了，雪崩，大量的请求涌入到商品服务调用的command中，是线程池不够

reject，被reject掉的请求就会去执行fallback降级逻辑

理清楚一些前提，首先一个请求都发送到这里来了，那么nginx本地缓存肯定就没了，redis已经崩溃了，ehcache中找不到这条数据对应的缓存

只能从源头的商品服务里面去查询，但是被限流了，这个请求只能走降级方案

都是用之前讲解的一些技术，stubbed fallback降级机制，残缺的降级

一般这种情况下，就是说，用请求参数中少量的数据，加上纯内存中缓存的少量的数据来提供残缺的数据服务

就给大家举个例子，我们之前讲解的stubbed fallback，是从内存中加载了部分品牌数据，加载了部分城市地理位置的数据啦。。。

方案，可以做，冷热分离

冷数据，也就是说你可以这么认为，将一些过时的数据，比如一个商品信息一周前的版本，放入大数据的在线存储中，比如比较合适做冷数据存放的是hbase

hadoop，离线批处理，hdfs分布式存储，yarn分布式资源调度（跟hbase没关系），mapreduce分布式计算

hbase，基于hdfs分布式存储基础之上，封装了一个系统，叫做hbase，分布式在线存储，分布式NoSQL数据库，里面可以放大量的冷数据

hbase，可以做商品服务热数据是放mysql，可以将一周前，一个月前的数据快照，做一份冷备放到hbase来备用

你本来正常情况下是直接去访问商品服务，去拉取热数据

发送请求去访问hbase，去加载冷数据，hbase本身是分布式的，所以也是可以承载高并发的访问的（分布式的特性比mysql），即使这个时候大量并发到了hbase，如果你集群运维够好的话，也开始以撑住的，加载到一条冷数据的话，那么此时就是过期的数据，商品一周前或者一个月前的一个快照版本

但是至少有数据，还可以显示一下

多级降级机制，先走hbase冷备，然后再走stubbed fallback

缓存雪崩的回顾

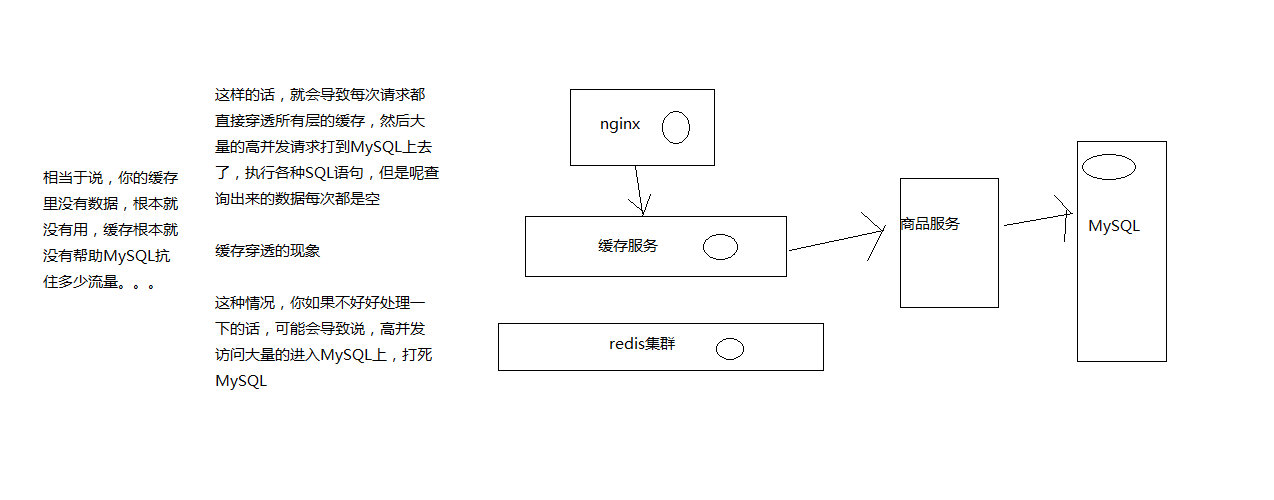
1、事前，redis高可用性，redis cluster，sentinal，复制，主从，从->主，双机房部署

2、事中，ehcache可以抗一抗，redis挂掉之后的资源隔离、超时控制、熔断，商品服务的访问限流、多级降级，缓存服务在雪崩场景下存活下来，基于ehcache和存活的商品服务提供数据

3、事后，快速恢复Redis，备份+恢复，快速的缓存预热的方案

#### 117\_高并发场景下的缓存穿透导致MySQL压力倍增问题以及其解决方案

缓存穿透的现象



缓存穿透

缓存穿透的现象是什么呢？

#### 118\_在缓存服务中开发缓存穿透的保护性机制

我们的缓存穿透的解决方案，其实非常的简单，就是说每次如果从源服务（商品服务）查询到的数据是空，就说明这个数据根本就不存在

那么如果这个数据不存在的话，我们不要不往redis和ehcache等缓存中写入数据，我们呢，给写入一个空的数据，比如说空的productInfo的json串

给nginx也是，返回一个空的productInfo的json串咯

因为我们有一个异步监听数据变更的机制在里面，也就是说，如果数据变更的话，某个数据本来是没有的，可能会导致缓存穿透，所以我们给了个空数据

但是现在这个数据有了，我们接收到这个变更的消息过后，就可以将数据再次从源服务中查询出来

然后设置到各级缓存中去了

#### 119\_高并发场景下的nginx缓存失效导致redis压力倍增问题以及解决方案

缓存失效

就是大家还记得，我们在nginx中设置本地的缓存的时候，会给一个过期的时间，比如说10分钟

10分钟以后自动过期，过期了以后，就会重新从redis中去获取数据

这个10分钟到期自动过期的事情，就叫做缓存的失效

如果缓存失效以后，那么实际上此时，就会有大量的请求回到redis中去查询

缓存失效的问题。。。。

如果说同一时间来了1000个请求，都将缓存cache在了nginx自己的本地，缓存失效的时间都设置了10分钟

那么是不是可能导致10分钟过后，这些数据，就自动全部在同一时间失效了

如果同一时间全部失效，会不会导致说同一时间大量的请求过来，在nginx里找不到缓存数据，全部高并发走到redis上去了

加重大量的网络请求，网络负载也会加重

解决方案是什么呢？

#### 120\_在nginx lua脚本中开发缓存失效的保护性机制

math.randomseed(tostring(os.time()):reverse():sub(1, 7))

local expireTime = math.random(600, 1200)

#### 121\_支撑高并发与高可用的大型电商详情页系统的缓存架构课程总结

1、亿级流量电商网站的商品详情页系统架构

面临难题：对于每天上亿流量，拥有上亿页面的大型电商网站来说，能够支撑高并发访问，同时能够秒级让最新模板生效的商品详情页系统的架构是如何设计的？

解决方案：异步多级缓存架构+nginx本地化缓存+动态模板渲染的架构

2、redis企业级集群架构

面临难题：如何让redis集群支撑几十万QPS高并发+99.99%高可用+TB级海量数据+企业级数据备份与恢复？

解决方案：redis的企业级备份恢复方案+复制架构+读写分离+哨兵架构+redis cluster集群部署

3、多级缓存架构设计

面临难题：如何将缓存架构设计的能够支撑高性能以及高并发到极致？同时还要给缓存架构最后的一个安全保护层？

解决方案：nginx抗热点数据+redis抗大规模离线请求+ehcache抗redis崩溃的三级缓存架构

4、数据库+缓存双写一致性解决方案

面临难题：高并发场景下，如何解决数据库与缓存双写的时候数据不一致的情况？

解决方案：异步队列串行化的数据库+缓存双写一致性解决方案

5、缓存维度化拆分解决方案

面临难题：如何解决大value缓存的全量更新效率低下问题？

解决方案：商品缓存数据的维度化拆分解决方案

6、缓存命中率提升解决方案

面临难题：如何将缓存命中率提升到极致？

解决方案：双层nginx部署架构+lua脚本实现一致性hash流量分发策略

7、缓存并发重建冲突解决方案

面临难题：如何解决高并发场景下，缓存重建时的分布式并发重建的冲突问题？

解决方案：基于zookeeper分布式锁的缓存并发重建冲突解决方案

8、缓存预热解决方案

面临难题：如何解决高并发场景下，缓存冷启动导致MySQL负载过高，甚至瞬间被打死的问题？

解决方案：基于storm实时统计热数据的分布式快速缓存预热解决方案

9、热点缓存自动降级方案

面临难题：如何解决热点缓存导致单机器负载瞬间超高？

解决方案：基于storm的实时热点发现+毫秒级的实时热点缓存负载均衡降级

10、高可用分布式系统架构设计

面临难题：如何解决分布式系统中的服务高可用问题？避免多层服务依赖因为少量故障导致系统崩溃？

解决方案：基于hystrix的高可用缓存服务，资源隔离+限流+降级+熔断+超时控制

11、复杂的高可用分布式系统架构设计

面临难题：如何针对复杂的分布式系统将其中的服务设计为高可用架构？

解决方案：基于hystrix的容错+多级降级+手动降级+生产环境参数优化经验+可视化运维与监控

12、缓存雪崩解决方案

面临难题：如何解决恐怖的缓存雪崩问题？避免给公司带来巨大的经济损失？

解决方案：全网独家的事前+事中+事后三层次完美缓存雪崩解决方案

13、缓存穿透解决方案

面临难题：如何解决高并发场景下的缓存穿透问题？避免给MySQL带来过大的压力？

解决方案：缓存穿透解决方案

14、缓存失效解决方案

面临难题：如何解决高并发场景下的缓存失效问题？避免给redis集群带来过大的压力？

解决方案：基于随机过期时间的缓存失效解决方案

硬件规划

每日上亿流量，高峰QPS过1万

nginx部署，负载很重，16核32G，建议给3~5台以上，就非常充裕了，每台抗个几千QPS

缓存服务部署，4核8G，按照每台QPS支撑500，部署个10~20台

redis部署，每台给8核16G，根据数据量以及并发读写能力来看，部署5~10个master，每个master挂一个slave，主要是为了支撑更多数据量，1万并发读写肯定没问题了

#### 122\_如何将课程中的东西学以致用在自己目前的项目中去应用？

大家肯定学到了很多东西，技术，解决方案，架构

大家肯定会问，我怎么用？？？

提示一个思路，你学习到的任何课程，你如果要用这里的技术，你首先必须得对课程里的东西彻底给他嚼烂了

课程里那套东西吸收到自己独自里去

然后再看看自己的项目，有哪些地方是可以用到一些技术的，你的项目中有没有遇到我们讲解的各种问题

如果有可能，建议，将课程中学到的一些东西，如果是你之前不会的，尽量用到项目中去，将项目中的架构做的牛逼一些

缓存架构，java代码操作一下redis，都不考虑数据库+缓存双写的一致性问题

怎么将学到的东西应用到项目中去？

这块是这样子的，你肯定不是照搬，需要对你学到的各种解决方案进行一些改造和你的系统去融合

举个例子，你的MQ可能就不是kafka，可能就是ActiveMQ

分布式锁，可能不是基于zookeeper去做的，用你自己想要的锁就好

你要做实时的热点数据的统计，或者离线的热点数据的统计，也许量没那么大，不需要用storm，hadoop这种大数据的技术

你就直接用java系统自己封装自己写，不就得了

最重要的是什么？？？

我希望的是，这套课程学完以后，你对自己的系统多一些思考，多一些架构上的思考和设计

不要说出去的时候，就是各种技术整合在一起，就这么一个架构，而且各种技术还没用得多么的深

架构，考虑到各种高并发情况下的问题，然后你的架构是怎么设计的，怎么演化的，怎么去解决那些问题的

你的架构，明显比一般人的要好，思考的要多，你才能有机会拿到更高阶的技术岗位

#### 123\_如何带着课程中讲解的东西化为自己的技术并找一份更好的工作？

我做线上培训课程很长时间了，做java架构的课，没有出过

主要是出一些大数据spark的课程，其中有一个课程是一个纯企业级的大型的spark的大数据项目的课程

很多人学习了以后，有个问题

直接原封不动，直接就拿着课程里学到的东西，出去找工作，坑爹，很多人的简历，写自己做了个什么大数据的项目，几乎是一模一样

必死

你没做过电商的项目，结果你拿着这个电商项目出去，你就完蛋了

基本上，在行业里，一些项目实战类的课程，100%，都是用简化后的业务，来讲解各种技术和架构，方案

绝对不可能说一个几十个小时的课程，可以给你讲完一个大型项目所有的业务，带你做出来一个一模一样的项目

大项目，10+，做至少半年，5 \* 8 = 40 \* 4 = 160 ， 1000个小时，1w个小时

几十个小时，能给你讲清楚技术、架构和方案是足够的，但是也仅此而已了

任何项目实战类的课程，项目，简化业务，拿业务背景出来串起来所有的技术和知识，架构，让你在具体的业务背景中去学习

而不是简简单单的干学技术，写了一堆demo代码，没有这个技术在项目中如何运用的sense

你拿着一个项目实战的课程，原封不动出去找工作，你肯定完蛋了

你根本不了解这里面的业务，比如这个电商，你了解这个电商的业务吗？市面上任何一个课程讲解了吗？超级简化和简单的一个小电商？订单管理+购物车+首页

技巧

你肯定是要将项目课程中的技术嚼烂了，吞下去，变成自己的东西，然后呢？？？

跟自己的项目结合起来

项目实战课程，spark电商用户行为分析，很多人没有做过电商业务，银行信用卡系统，电信计费系统，网络运营系统，游戏运营系统

将一个课程中的技术全部提取出来，融合进你的项目的业务背景中

银行信用卡用户行为分析系统

电信用户短信通话行为分析系统

网络流量分析系统

游戏用户行为分析系统

技术提取出来，融入你自己的业务中去，业务你很清楚，门儿清，技术你也学到了

缓存架构

电商的缓存架构，商品详情页

银行信用卡系统的缓存架构，多级缓存，各种方案

电信计费系统的缓存架构

网络运营系统的缓存架构

游戏运营系统的缓存架构

#### 124\_大型电商网站的商品详情页的深入分析

之前，咱们也是说在讲解这个商品详情页系统的架构

缓存架构，高可用服务

商品详情页系统，我们只是抽取了其中一部分来讲解，而且还做了很大程度的简化

主要是为了用一个较为拟真的这么一个业务场景，重点是要讲解：缓存架构，高可用服务（hystrix）

在讲解完了之前的内容之后，相信大家也都掌握了一定的基础了，然后接下来我们就要去动手纯实战，去开发出来一个较为完整的亿级流量大型电商网站的商品详情页系统

升级内容：

（1）纯实战，我们不会过多的讲解一些技术（redis，缓存架构，hystrix高可用服务），redis和hystrix都是对技术的深入讲解

（2）技术讲解过多了，对于我们这样的一套单品课程来说，自然在第一版的时候，业务和完整架构自然讲解就少了

（3）这次升级，策略跟第一版不同，重点就是完整架构的项目实战，亿级流量电商详情页系统架构的完整架构的项目实战

（4）纯实战，做东西，不会深入讲解任何技术

（5）部署，简单的讲解，做

商品详情页介绍

商品详情页的多模板化

多套模板：聚划算、天猫超市、淘抢购、电器城

不同模板的元数据一样，只是展示方式不一样

不同的业务，商品详情页的个性化需求很多，数据来源也很多

商品详情页结构

时效性比较低的数据

一个商品详情包含了不同的维度

商品维度：标题，图片，属性，等等

主商品维度：商品介绍，规格参数，等等

分类维度

商家维度

时效性比较低的数据

其实后面会讲解，都是在一个商品详情页被访问到的时候，将数据动态直接渲染/填充到一个html模板中去的

在浏览器展现的时候，数据写死在html里面，直接就出来了

因为比如说，一个商品的数据变更了，可能是异步的去更新数据的，也许需要5分钟，或者10分钟的时间，才能将变更的数据反映的商品详情页中去

实时性较高的数据

实时价格、实时促销、广告词、配送至、预售、库存

ajax异步加载

在访问商品详情页的时候，价格、库存、促销活动、广告词，都没有直接写死到html中，直接是在html里放了一个js脚本

然后在html在浏览器显示出来的时候，js脚本运行，直接发送ajax请求到后端

后端接口，直接查询价格、库存、促销活动、广告词，最新的数据

只要你变更了数据，那么在下一次商品详情页展示的时候，一定可以将最新的数据展示出来

在淘宝网上展示一个通用商品模板，商品详情页结构拆解说明，分析一个商品详情页的多维度构成

亿级流量电商网站的商品详情页访问情况

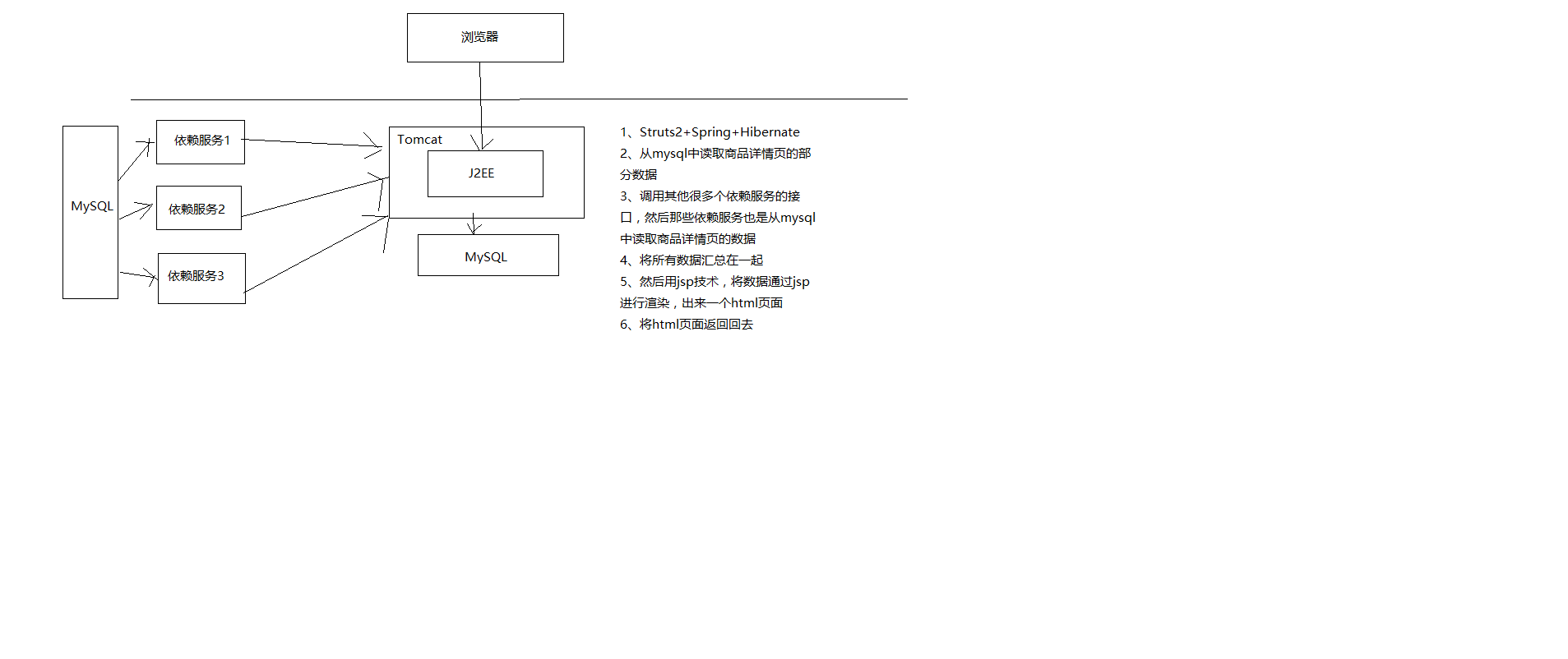
访问量：比如双11活动，商品详情页的pv至少达到几亿次，但是经过良好设计的详情页系统，响应时间小于几十ms

访问特点：离散访问多，热点数据少

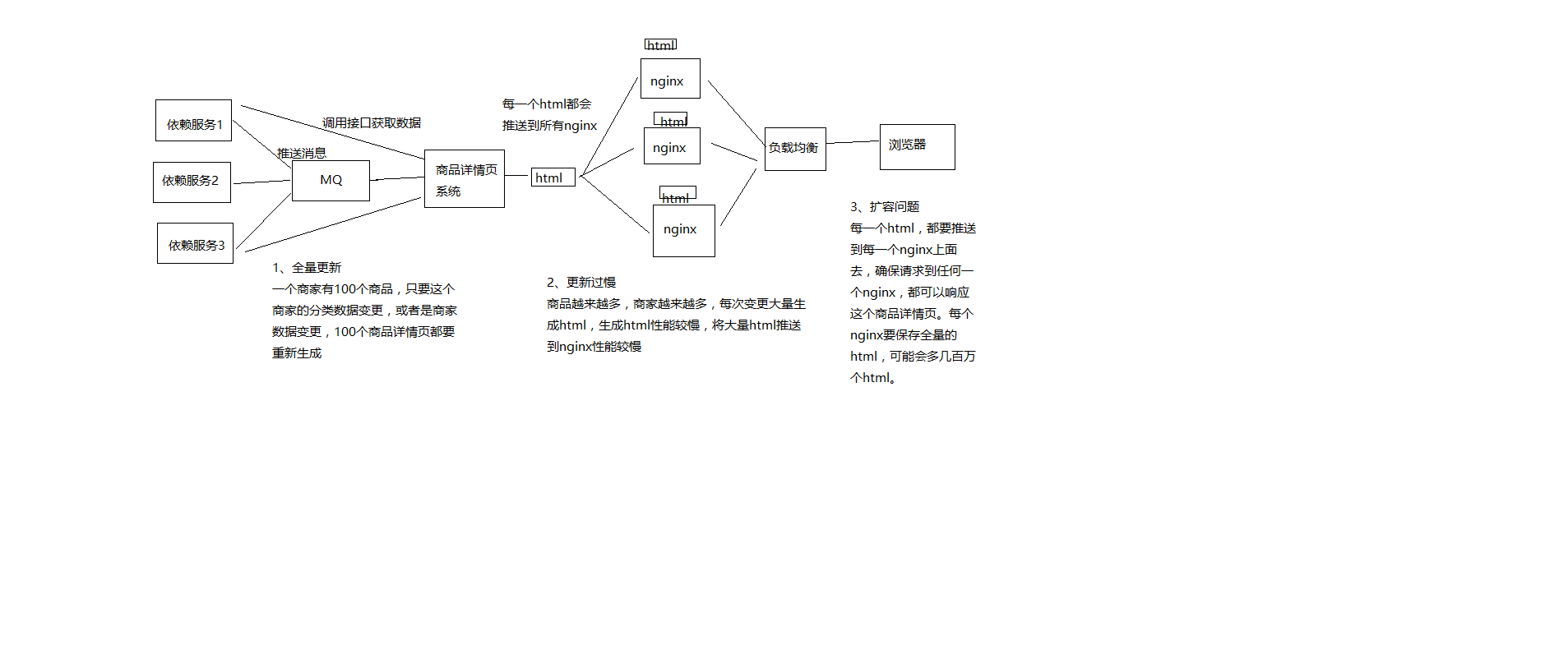
一般来说，访问的比较均匀，很少说集中式访问某个商品详情页，除非是那种秒杀活动，那是集中式访问某个商品详情页

#### 125\_大型电商网站的商品详情页系统架构是如何一步一步演进的

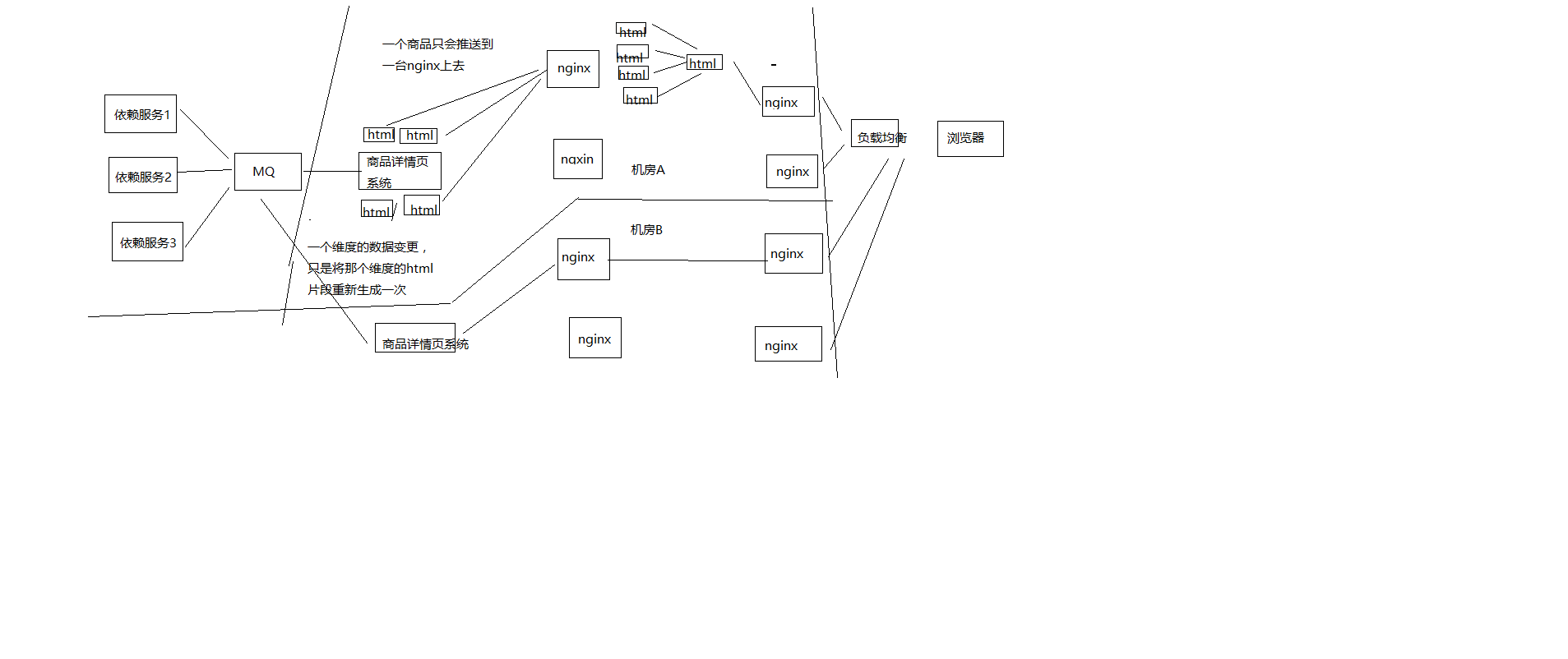
第一个版本的架构



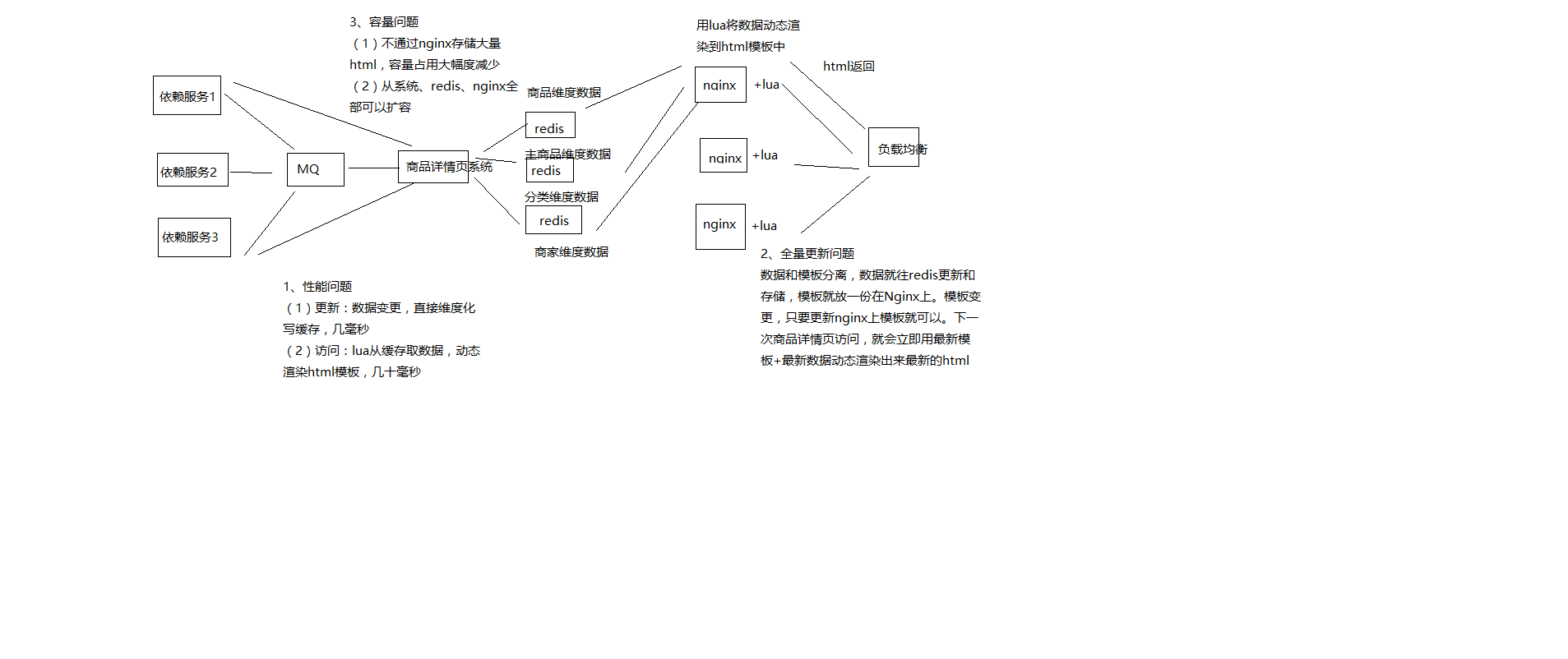
第二个版本架构以及其缺陷



第二个版本优化后的架构



第三个版本的架构



商品详情页系统架构演进历程

第一个版本

架构设计

J2EE+Tomcat+MySQL

动态页面，每次请求都要调用多个依赖服务的接口，从数据库里查询数据，然后通过类似JSP的技术渲染到HTML模板中，返回最终HTML页面

架构缺陷

每次请求都是要访问数据库的，性能肯定很差

每次请求都要调用大量的依赖服务，依赖服务不稳定导致商品详情页展示的性能经常抖动

第二个版本

架构设计

页面静态化技术

通过MQ得到商品详情页涉及到的数据的变更消息

通过Java Worker服务全量调用所有的依赖服务的接口，查询数据库，获取到构成一个商品详情页的完整数据，并通过velocity等模板技术生成静态HTML

将静态HTML页面通过rsync工具直接推送到多台nginx服务器上，每台nginx服务器上都有全量的HTML静态页面

nginx对商品详情页的访问请求直接返回本地的静态HTML页面

在nginx服务器前加一层负载均衡设备，请求打到任何一台应用nginx服务器上，都有全量的HTML静态页面可以返回

架构缺陷

全量更新问题

如果某一个商品分类、商家等信息变更了

那么那个分类、店铺、商家下面所有的商品详情页都需要重新生成静态HTML页面

更新速度过慢问题

分类、店铺、商家、商品越来越多

重新生成HTML的负载越来越高，rsync全量同步所有nginx的负载也越来越高

从数据变更到生成静态HTML，再到全量同步到所有nginx，时间越来越慢

扩容问题

因为每个商品详情页都要全量同步到所有的nginx上，导致系统无法扩容，无法增加系统容量

架构优化

解决全量更新问题

每次Java Worker收到某个维度的变更消息，不是拉去全量维度并生成完整HTML，而是按照维度拆分，生成一个变化维度的HTML片段

nginx对多个HTML片段通过SSI合并html片段然后输出一个完整的html

解决扩容问题

每个商品详情页不是全量同步到所有的nginx

而是根据商品id路由到某一台nginx上，同时接入层nginx按照相同的逻辑路由请求

更新速度过慢问题

增加更多机器资源

多机房部署，每个机房部署一套Java Worker+应用Nginx，所有机房用一套负载均衡设备，在每个机房内部完成全流程，不跨机房

架构优化后的缺陷

更新速度还是不够快的问题

商品的每个维度都有一个HTML片段，rsync推送大量的HTML片段，负载太高，性能较差

Nginx基于机械硬盘进行SSI合并，性能太差

还是存在全量更新的问题

虽然解决了分类、商家、店铺维度的变更，只要增量重新生产较小的HTML片段即可，不用全量重新生成关联的所有商品详情页的HTML

但是如果某个页面模板变更，或者新加入一个页面模板，还是会导致几亿个商品的HTML片段都要重新生成和rsync，要几天时间才能完成，无法响应需求

还是存在容量问题

nginx存储有限，不能无限存储几亿，以及增长的商品详情页的HTML文件

如果nginx存储达到极限，需要删除部分商品详情页的HTML文件，改成nginx找不到HTML，则调用后端接口，回到动态页面的架构

动态页面架构在高并发访问的情况下，会对依赖系统造成过大的压力，几乎扛不住

第三个版本

需要支持的需求

迅速响应各种页面模板的改版和个性化需求的新模板的加入

页面模块化，页面中的某个区域变化，只要更新这个区域中的数据即可

支持高性能访问

支持水平扩容的伸缩性架构

架构设计

系统架构设计

依赖服务有数据变更发送消息到MQ

数据异构Worker服务监听MQ中的变更消息，调用依赖服务的接口，仅仅拉取有变更的数据即可，然后将数据存储到redis中

数据异构Worker存储到redis中的，都是原子未加工数据，包括商品基本信息、商品扩展属性、商品其他信息、商品规格参数、商品分类、商家信息

数据异构Worker发送消息到MQ，数据聚合Worker监听到MQ消息

数据聚合Worker将原子数据从redis中取出，按照维度聚合后存储到redis中，包括三个维度

基本信息维度：基本信息、扩展属性

商品介绍：PC版、移动版

其他信息：商品分类、商家信息

nginx+lua，lua从redis读取商品各个维度的数据，通过nginx动态渲染到html模板中，然后输出最终的html

如何解决所有的问题

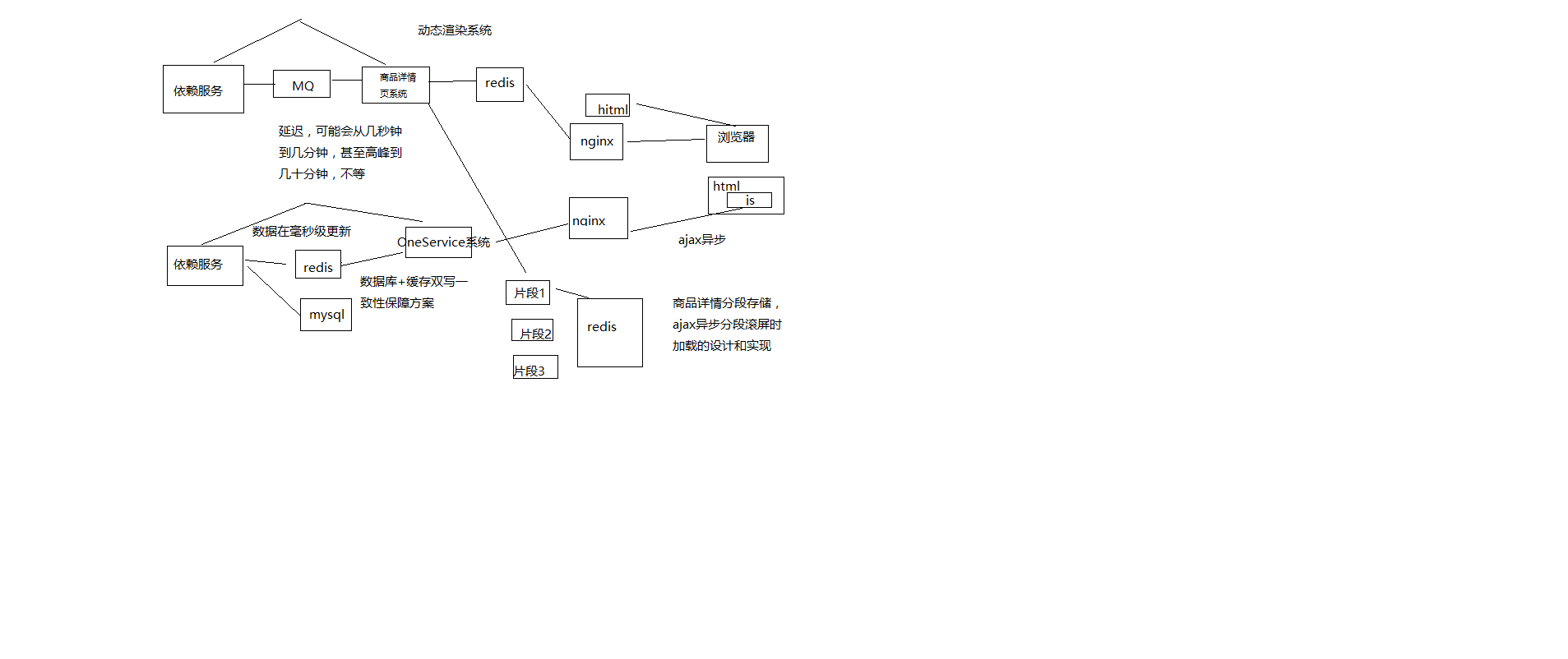
更新问题：不再是生成和推送html片段了，不再需要合成html，直接数据更新到redis，然后走动态渲染，性能大大提升

全量更新问题：数据和模板分离，数据更新呢就更新数据，模板更新直接推送模板到nginx，不需要重新生成所有html，直接走动态渲染

容量问题：不需要依赖nginx所在机器的磁盘空间存储大量的html，将数据放redis，html就存放模板，大大减少空间占用，而且redis集群可扩容

#### 126\_亿级流量大型电商网站的商品详情页系统架构的整体设计

商品详情页系统整体架构设计



商品详情页整体架构组成

动态渲染系统

将页面中静的数据，直接在变更的时候推送到缓存，然后每次请求页面动态渲染新数据

商品详情页系统（负责静的部分）：被动接收数据，存储redis，nginx+lua动态渲染

商品详情页动态服务系统（对外提供数据接口）

提供各种数据接口

动态调用依赖服务的接口，产生数据并且返回响应

从商品详情页系统处理出来的redis中，获取数据，并返回响应

OneService系统

动的部分，都是走ajax异步请求的，不是走动态渲染的

商品详情页统一服务系统（负责动的部分）

前端页面

静的部分，直接被动态渲染系统渲染进去了

动的部分，html一到浏览器，直接走js脚本，ajax异步加载

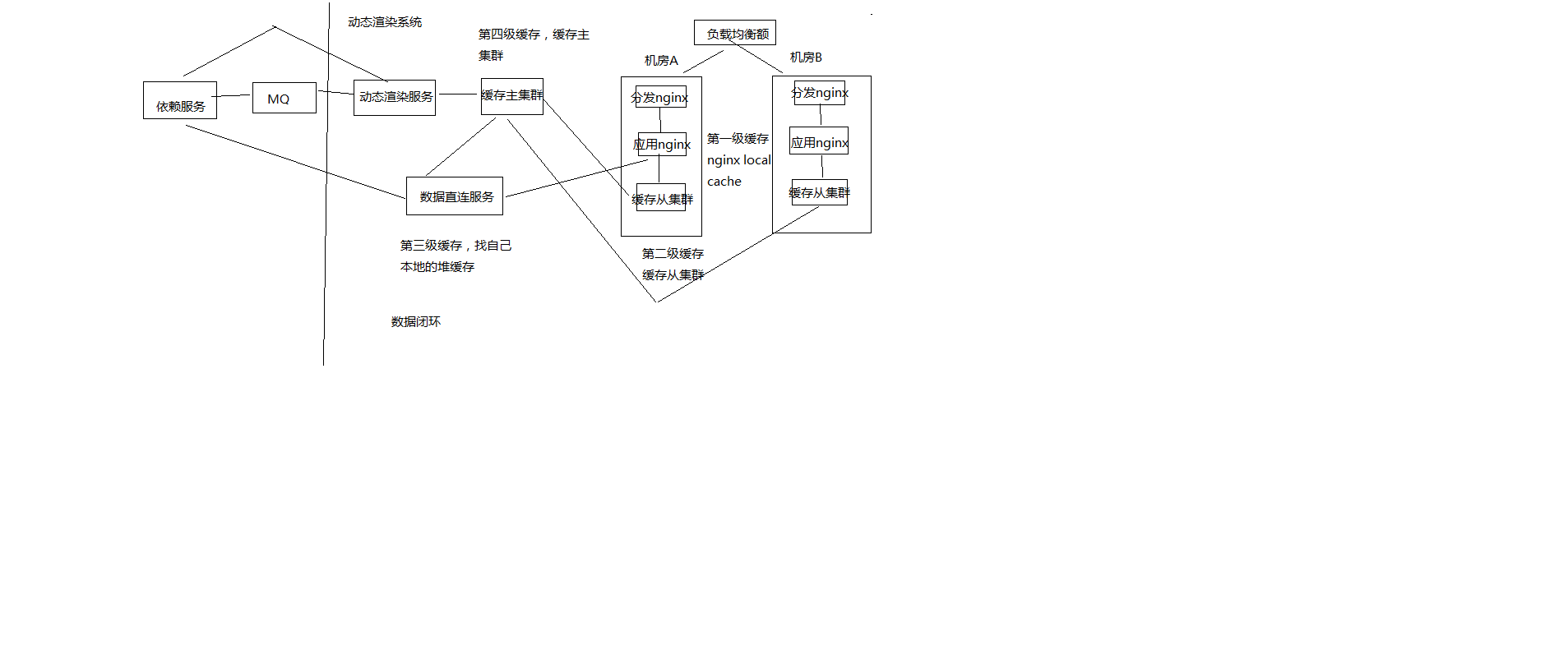
商品详情页，分段存储，ajax异步分屏加载

工程运维

限流，压测，灰度发布

#### 127\_商品详情页动态渲染系统：架构整体设计

动态渲染系统整体架构设计



我们先做动态渲染那套系统

（1）依赖服务 -> MQ -> 动态渲染服务 -> 多级缓存

（2）负载均衡 -> 分发层nginx -> 应用层nginx -> 多级缓存

（3）多级缓存 -> 数据直连服务

动态渲染系统

数据闭环

数据闭环架构

依赖服务：商品基本信息，规格参数，商家/店铺，热力图，商品介绍，商品维度，品牌，分类，其他

发送数据变更消息到MQ

数据异构Worker集群，监听MQ，将原子数据存储到redis，发送消息到MQ

数据聚合Worker集群，监听MQ，将原子数据按维度聚合后存储到redis，三个维度（商品基本信息、商品介绍、其他信息）

数据闭环，就是数据的自我管理，所有数据原样同步后，根据自己的逻辑进行后续的数据加工，走系统流程，以及展示k

数据形成闭环之后，依赖服务的抖动或者维护，不会影响到整个商品详情页系统的运行

数据闭环的流程：数据异构（多种异构数据源拉取），数据原子化，数据聚合（按照维度将原子数据进行聚合），数据存储（Redis）

数据维度化

商品基本信息：标题、扩展属性、特殊属性、图片、颜色尺码、规格参数

商品介绍

非商品维度其他信息：分类，商家，店铺，品牌

商品维度其他信息：采用ajax异步加载，价格，促销，配送至，广告，推荐，最佳组合，等等

采取ssdb，这种基于磁盘的大容量/高性能的kv存储，保存商品维度、主商品维度、商品维度其他信息，数据量大，不能光靠内存去支撑

采取redis，纯内存的kv存储，保存少量的数据，比如非商品维度的其他数据，商家数据，分类数据，品牌数据

一个完整的数据，拆分成多个维度，每个维度独立存储，就避免了一个维度的数据变更就要全量更新所有数据的问题

不同维度的数据，因为数据量的不一样，可以采取不同的存储策略

系统拆分

系统拆分更加细：依赖服务、MQ、数据异构Worker、数据同步Worker、Redis、Nginx+Lua

每个部分的工作专注，影响少，适合团队多人协作

异构Worker的原子数据，基于原子数据提供的服务更加灵活

聚合Worker将数据聚合后，减少redis读取次数，提升性能

前端展示分离为商品详情页前端展示系统和商品介绍前端展示系统，不同特点，分离部署，不同逻辑，互相不影响

异步化

异步化，提升并发能力，流量削峰

消息异步化，让各个系统解耦合，如果使用依赖服务调用商品详情页系统接口同步推送，那么就是耦合的

缓存数据更新异步化，数据异构Worker同步调用依赖服务接口，但是异步更新redis

动态化

数据获取动态化：nginx+lua获取商品详情页数据的时候，按照维度获取，比如商品基本数据、其他数据（分类、商家）

模板渲染实时化：支持模板页面随时变化，因为采用的是每次从nginx+redis+ehcache缓存获取数据，渲染到模板的方式，因此模板变更不用重新静态化HTML

重启应用秒级化：nginx+lua架构，重启在秒级

需求上线快速化：使用nginx+lua架构开发商品详情页的业务逻辑，非常快速

多机房多活

Worker无状态，同时部署在各自的机房时采取不同机房的配置，来读取各自机房内部部署的数据集群（redis、mysql等）

将数据异构Worker和数据聚合Worker设计为无状态化，可以任意水平扩展

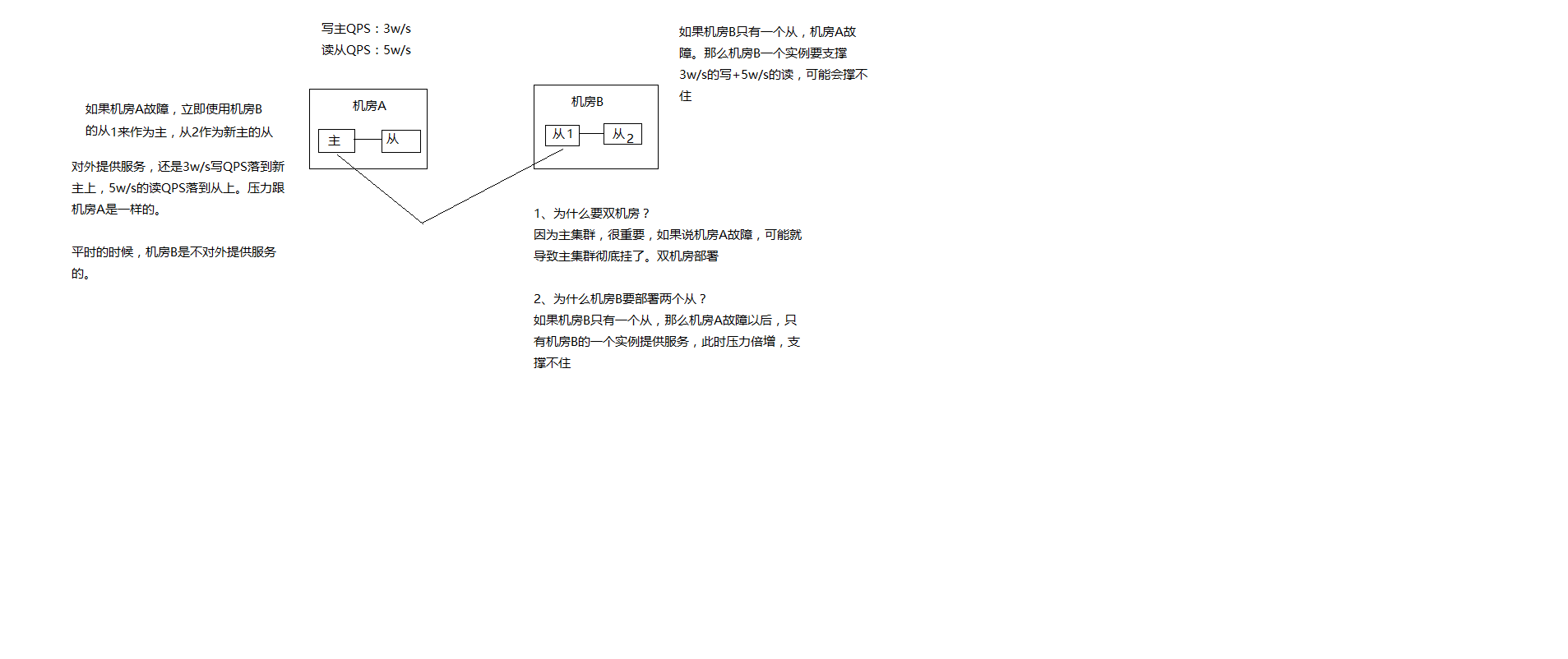
Worker无状态化，但是配置文件有状态，不同的机房有一套自己的配置文件，只读取自己机房的redis、ssdb、mysql等数据

每个机房配置全链路：接入nginx、商品详情页nginx+商品基本信息redis集群+其他信息redis集群、商品介绍nginx+商品介绍redis集群

部署统一的CDN以及LVS+KeepAlived负载均衡设备

#### 128\_商品详情页动态渲染系统：大型网站的多机房4级缓存架构设计

主集群的双机房一主三从部署架构



多级缓存架构

本地缓存

使用nginx shared dict作为local cache，http-lua-module的shared dict可以作为缓存，而且reload nginx不会丢失

也可以使用nginx proxy cache做local cache

双层nginx部署，一层接入，一层应用，接入层用hash路由策略提升缓存命中率

比如库存缓存数据的TP99为5s，本地缓存命中率25%，redis命中率28%，回源命中率47%

一次普通秒杀活动的命中率，本地缓存55%，分布式redis命中率15%，回源命中率27%

最高可以提升命中率达到10%

全缓存链路维度化存储，如果有3个维度的数据，只有其中1个过期了，那么只要获取那1个过期的数据即可

nginx local cache的过期时间一般设置为30min，到后端的流量会减少至少3倍

4级多级缓存

nginx本地缓存，抗热点数据，小内存缓存访问最频繁的数据

各个机房本地的redis从集群的数据，抗大量离线数据，采用一致性hash策略构建分布式redis缓存集群

tomcat中的动态服务的本地jvm堆缓存

支持在一个请求中多次读取一个数据，或者与该数据相关的数据

作为redis崩溃的备用防线

固定缓存一些较少访问频繁的数据，比如分类，品牌等数据

堆缓存过期时间为redis过期时间的一半

主redis集群

命中率非常低，小于5%

防止主从同步延迟导致的数据读取miss

防止各个机房的从redis集群崩溃之后，全量走依赖服务会导致雪崩，主redis集群是后备防线

主redis集群，采取多机房一主三从的高可用部署架构

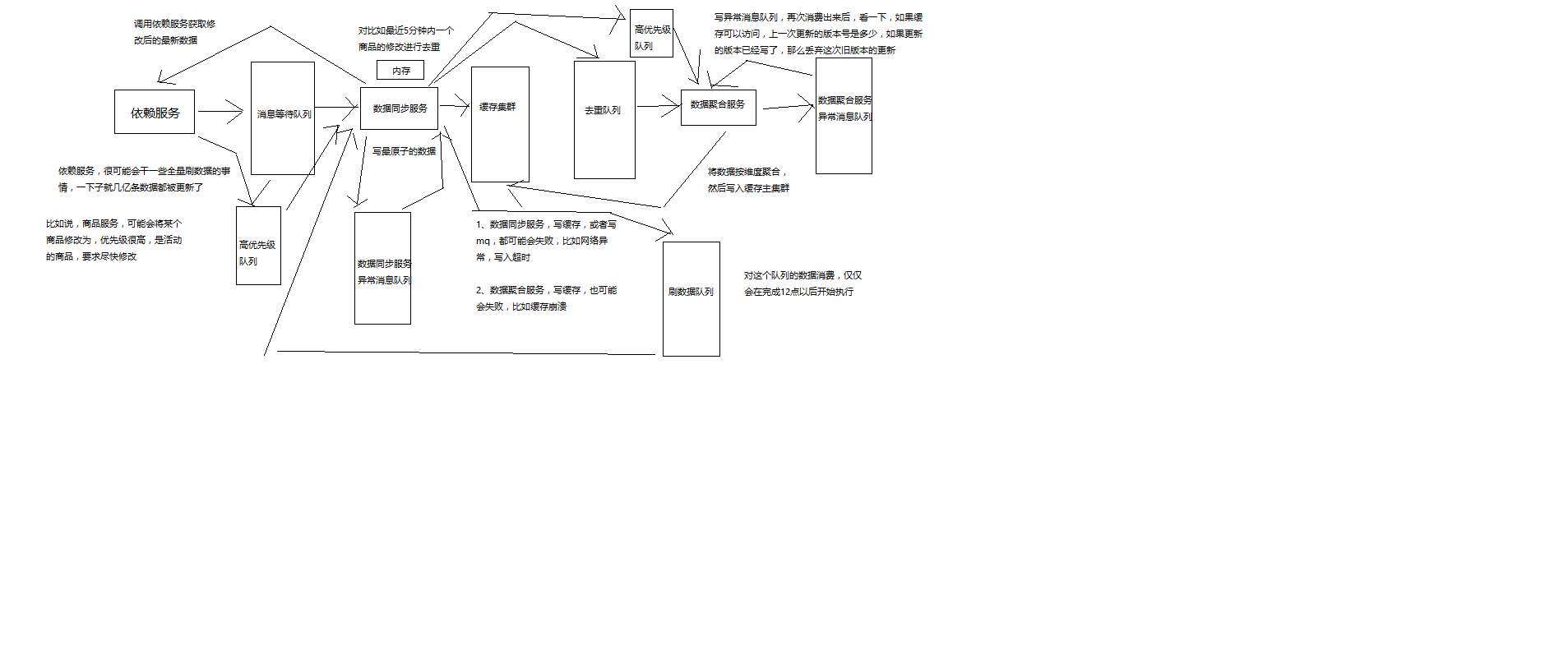
redis集群部署采取双机房一主三活的架构，机房A部署主集群+一个从集群，机房B部署一个从集群（从机房A主集群）+一个从集群（从机房B从集群）

双机房一主三活的架构，保证了机房A彻底故障的时候，机房B还有一套备用的集群，可以升级为一主一从

如果采取机房A部署一主一从，机房B一从，那么机房A故障时，机房B的一从承载所有读写压力，压力过大，很难承受

#### 129\_商品详情页动态渲染系统：复杂的消息队列架构设计

复杂的消息队列架构设计



队列化

任务等待队列

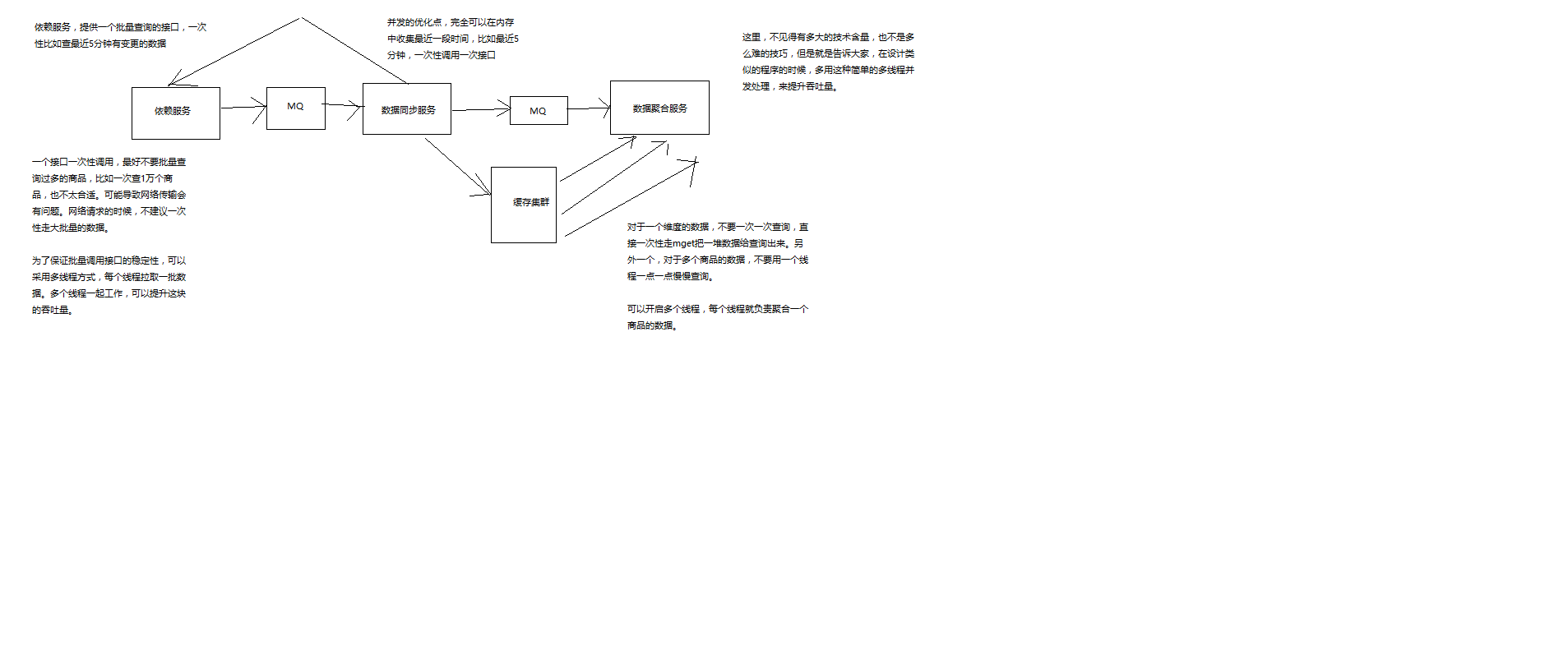
任务排重队列（异构Worker对一个时间段内的变更消息做排重）

失败任务队列（失败重试机制）

优先级队列，刷数据队列（依赖服务洗数据）、高优先级队列（活动商品优先级高）

#### 130\_商品详情页动态渲染系统：使用多线程并发提升系统吞吐量的设计

多线程提升吞吐量的设计



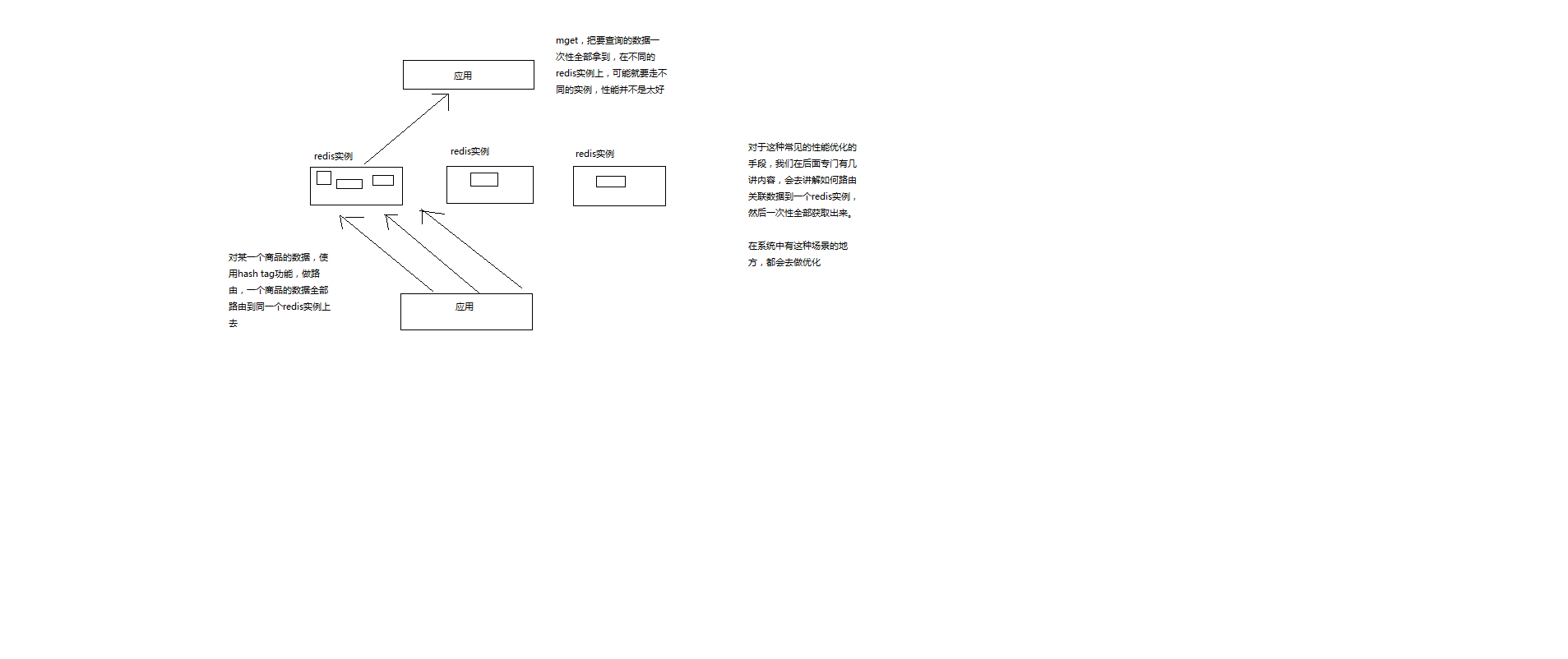
并发化

数据同步服务做并发化+合并，将多个变更消息合并在一起，调用依赖服务一次接口获取多个数据，采用多线程并发调用

数据聚合服务做并发化，每次重新聚合数据的时候，对多个原子数据用多线程并发从redis查询

#### 131\_商品详情页动态渲染系统：redis批量查询性能优化设计

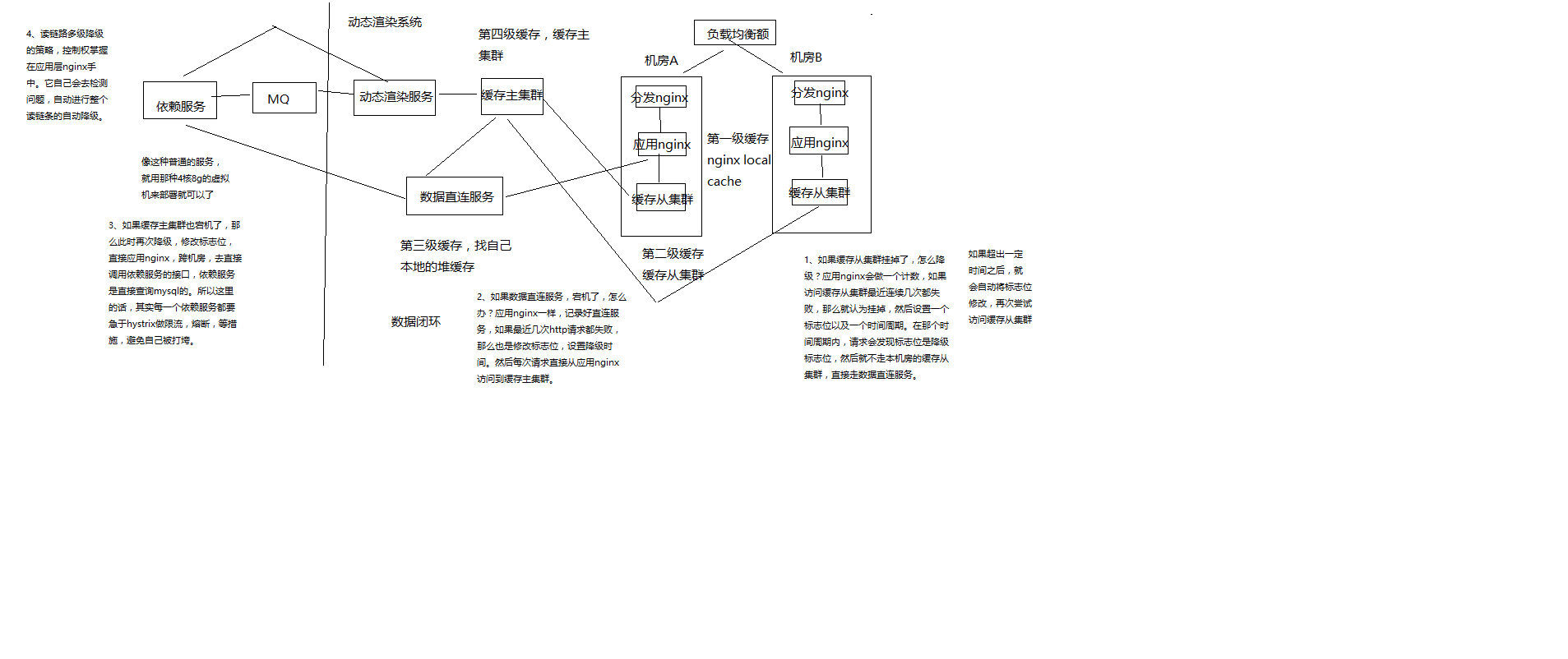
redis批量查询的性能优化



无

#### 132\_商品详情页动态渲染系统：全链路高可用架构设计

读链路多级降级设计



高可用设计

读链路多级降级：本机房从集群 -> 主集群 -> 直连

全链路隔离

基于hystrix的依赖调用隔离，限流，熔断，降级

普通服务的多机房容灾冗余部署以及隔离

#### 133\_商品详情页动态渲染系统：微服务架构设计

1、领域驱动设计：我们需要对这个系统涉及到的领域模型机进行分析，然后进行领域建模，最后的话，设计出我们对应的微服务的模型

2、spring cloud：微服务的基础技术架构，我们用spring cloud来做

3、持续交付流水线，jenkins+git+自动化持续集成+自动化测试+自动化部署

4、docker：大量的微服务的部署与管理

我们这个课程

一大块，真实的完整的亿级流量高并发高可用的电商详情页系统的架构实战

另一块，里面的服务，都会用微服务架构来做，相当于是在真实业务场景下的微服务项目实战

#### 134\_商品详情页动态渲染系统：机房与机器的规划

虚拟机，要弄几台，大概怎么来部署

负载均衡：2台机器，lvs+keepalived，双机高可用

两个机房，每个机房给1台机器，总共就是2台机器，分发层nginx+应用层nginx+缓存从集群

缓存主集群：模拟跟上面的两个机房部署在一起，在实际生产环境中，的确可能是在相同的机房，但是肯定在不同的机器上

我们这里不会有真正的机房，但是会模拟出来，有些机器会在某个机房里

缓存集群分片中间件，跟缓存集群部署在一起

rabbitmq和mysql：1台机器

5

#### 135\_商品详情页动态渲染系统：部署CentOS虚拟机集群

2台3G内存的虚拟机

1、在虚拟机中安装CentOS

（1）使用课程提供的CentOS 6.5镜像即可，CentOS-6.5-i386-minimal.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为eshop-cache01，选择操作系统为Linux，选择版本为Red Hat，分配1024MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。

（4）安装虚拟机中的CentOS 6.5操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 6.5镜像文件），选择第一项开始安装-Skip-欢迎界面Next-选择默认语言-Baisc Storage Devices-Yes, discard any data-主机名:spark2upgrade01-选择时区-设置初始密码为hadoop-Replace Existing Linux System-Write changes to disk-CentOS 6.5自己开始安装。

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

vi /etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=eth0

TYPE=Ethernet

ONBOOT=yes

BOOTPROTO=dhcp

service network restart

ifconfig

BOOTPROTO=static

IPADDR=192.168.0.X

NETMASK=255.255.255.0

GATEWAY=192.168.0.1

service network restart

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

比如后面我们要安装很多其他的一些东西，perl，java，redis，storm，复制一些命令直接去执行

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

service iptables stop

service ip6tables stop

chkconfig iptables off

chkconfig ip6tables off

vi /etc/selinux/config

SELINUX=disabled

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

yum clean all

yum makecache

yum install wget

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

2、在每个CentOS中都安装Java和Perl

（1）安装JDK

1、将jdk-7u60-linux-i586.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-7u65-linux-i586.rpm

3、配置jdk相关的环境变量

vi .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

（2）安装Perl

yum install -y gcc

wget http://www.cpan.org/src/5.0/perl-5.16.1.tar.gz

tar -xzf perl-5.16.1.tar.gz

cd perl-5.16.1

./Configure -des -Dprefix=/usr/local/perl

make && make test && make install

perl -v

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

3、在另外一个虚拟机中安装CentOS集群

（1）按照上述步骤，再安装1台一模一样环境的linux机器

（2）另外三台机器的hostname分别设置为eshop-detail02

（3）安装好之后，在每台机器的hosts文件里面，配置好所有的机器的ip地址到hostname的映射关系

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

4、配置2台CentOS为ssh免密码互相通信

（1）首先在三台机器上配置对本机的ssh免密码登录

ssh-keygen -t rsa

生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下

cd /root/.ssh

cp id\_rsa.pub authorized\_keys

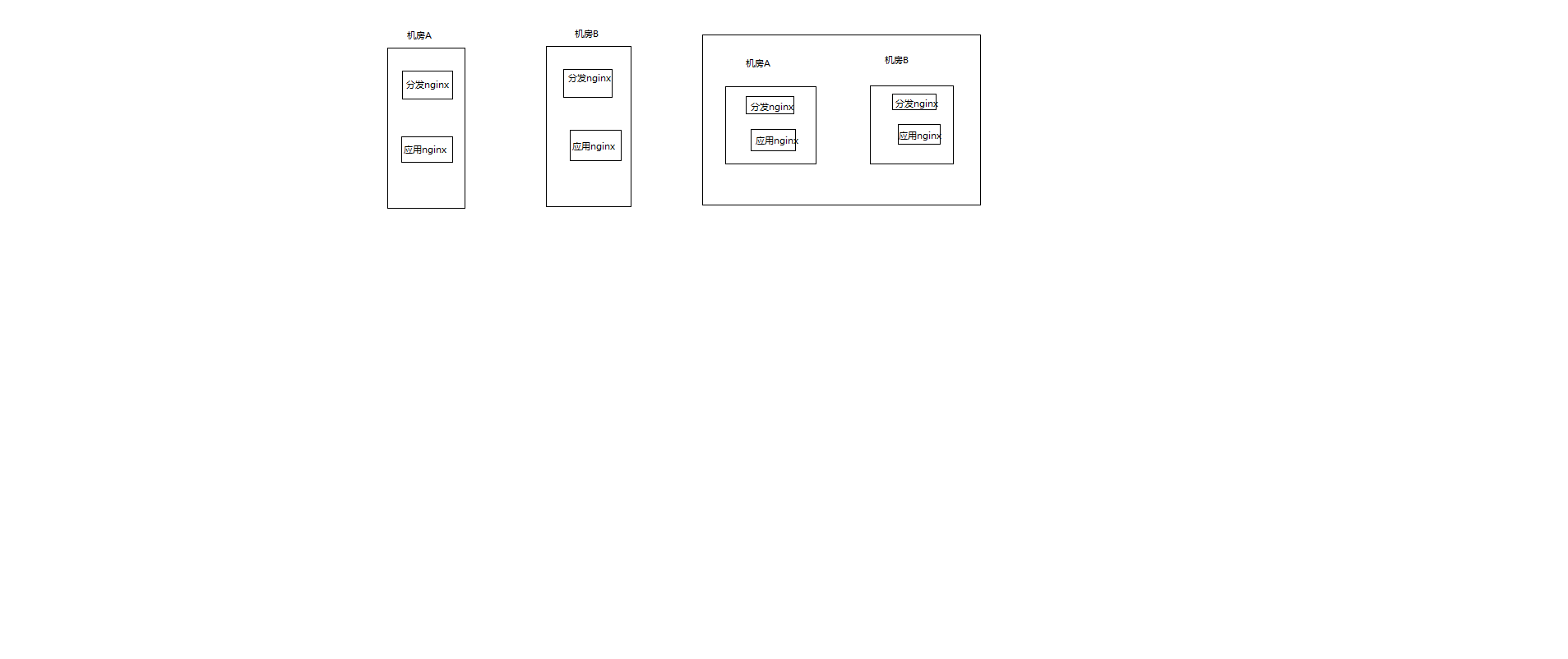
将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了

（2）接着配置三台机器互相之间的ssh免密码登录

使用ssh-copy-id -i hostname命令将本机的公钥拷贝到指定机器的authorized\_keys文件中

#### 136\_商品详情页动态渲染系统：双机房部署接入层与应用层Nginx+Lua

部署结构图说明



openresty，我们之前都给大家讲解过了

部署了两台虚拟机

模拟的场景是什么，假设这两台虚拟机分别在不同的机房中，每个机房里都有一台机器，所以按照我们之前讲解的那套双机房的四级缓存架构

部署nginx，虚拟机，每台机器上，部署两个nginx，一个是分发层nginx，一个是应用层nginx

在实际生产环境中

1、部署第一个nginx

（1）部署openresty

mkdir -p /usr/servers

cd /usr/servers/

yum install -y readline-devel pcre-devel openssl-devel gcc

wget http://openresty.org/download/ngx\_openresty-1.7.7.2.tar.gz

tar -xzvf ngx\_openresty-1.7.7.2.tar.gz

cd /usr/servers/ngx\_openresty-1.7.7.2/

cd bundle/LuaJIT-2.1-20150120/

make clean && make && make install

ln -sf luajit-2.1.0-alpha /usr/local/bin/luajit

cd bundle

wget https://github.com/FRiCKLE/ngx\_cache\_purge/archive/2.3.tar.gz

tar -xvf 2.3.tar.gz

cd bundle

wget https://github.com/yaoweibin/nginx\_upstream\_check\_module/archive/v0.3.0.tar.gz

tar -xvf v0.3.0.tar.gz

cd /usr/servers/ngx\_openresty-1.7.7.2

./configure --prefix=/usr/servers --with-http\_realip\_module --with-pcre --with-luajit --add-module=./bundle/ngx\_cache\_purge-2.3/ --add-module=./bundle/nginx\_upstream\_check\_module-0.3.0/ -j2

make && make install

cd /usr/servers/

ll

/usr/servers/luajit

/usr/servers/lualib

/usr/servers/nginx

/usr/servers/nginx/sbin/nginx -V

启动nginx: /usr/servers/nginx/sbin/nginx

（2）nginx+lua开发的hello world

vi /usr/servers/nginx/conf/nginx.conf

在http部分添加：

lua\_package\_path "/usr/servers/lualib/?.lua;;";

lua\_package\_cpath "/usr/servers/lualib/?.so;;";

/usr/servers/nginx/conf下，创建一个lua.conf

server {

listen 80;

server\_name \_;

}

在nginx.conf的http部分添加：

include lua.conf;

验证配置是否正确：

/usr/servers/nginx/sbin/nginx -t

在lua.conf的server部分添加：

location /lua {

default\_type 'text/html';

content\_by\_lua 'ngx.say("hello world")';

}

/usr/servers/nginx/sbin/nginx -t

重新nginx加载配置

/usr/servers/nginx/sbin/nginx -s reload

访问http: http://192.168.31.187/lua

vi /usr/servers/nginx/conf/lua/test.lua

ngx.say("hello world");

修改lua.conf

location /lua {

default\_type 'text/html';

content\_by\_lua\_file conf/lua/test.lua;

}

查看异常日志

tail -f /usr/servers/nginx/logs/error.log

（3）工程化的nginx+lua项目结构

项目工程结构

hello

hello.conf

lua

hello.lua

lualib

\*.lua

\*.so

放在/usr/hello目录下

/usr/servers/nginx/conf/nginx.conf

worker\_processes 2;

error\_log logs/error.log;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type text/html;

lua\_package\_path "/usr/hello/lualib/?.lua;;";

lua\_package\_cpath "/usr/hello/lualib/?.so;;";

include /usr/hello/hello.conf;

}

/usr/hello/hello.conf

server {

listen 80;

server\_name \_;

location /lua {

default\_type 'text/html';

lua\_code\_cache off;

content\_by\_lua\_file /usr/example/lua/test.lua;

}

}

#### 137\_商品详情页动态渲染系统：为什么是twemproxy+redis而不是redis cluster？

1、LVS那块不讲解

LVS+KeepAlived，负载均衡那块，讲一讲，还是不讲了，意义不是太大

MySQL+Atlas，分库分表，鸡肋

单课，聚焦，围绕一个主题去讲解，太发散了以后，什么都讲，没有围绕主题去讲解，意义不是太大

商品详情页系统，亿级流量大电商，核心的东西

随着课程不断讲解，可能会有10%的出入，砍掉或者调整一些细枝末节，大的思路是ok的，不会改变的

对课程，更加深入的思考

每一讲的标题会不断的变更

2、redis cluster的问题

twemproxy+redis去做集群，redis部署多个主实例，每个主实例可以挂载一些redis从实例，如果将不同的数据分片，写入不同的redis主实例中，twemproxy这么一个缓存集群的中间件

redis cluster

（1）不好做读写分离，读写请求全部落到主实例上的，如果要扩展写QPS，或者是扩展读QPS，都是需要扩展主实例的数量，从实例就是一个用做热备+高可用

（2）不好跟nginx+lua直接整合，lua->redis的client api，但是不太支持redis cluster，中间就要走一个中转的java服务

（3）不好做树状集群结构，比如redis主集群一主三从双机房架构，redis cluster不太好做成那种树状结构

（4）方便，相当于是上下线节点，集群扩容，运维工作，高可用自动切换，比较方便

3、twemproxy+redis

（1）上线下线节点，有一些手工维护集群的成本

（2）支持redis集群+读写分离，就是最基本的多个redis主实例，twemproxy这个中间件来决定的，java/nginx+lua客户端，是连接twemproxy中间件的。每个redis主实例就挂载了多个redis从实例，高可用->哨兵，redis cluster读写都要落到主实例的限制，你自己可以决定写主，读从，等等

（3）支持redis cli协议，可以直接跟nginx+lua整合

（4）可以搭建树状集群结构

4、如何选择？

（1）看你是否一定需要那3点了，如果不需要，那么用redis cluster也ok，大多数情况下，很多应用用redis就是比较简单的，做一个缓存

（2）如果你的架构里很需要那3点，那么用twemproxy比较好，商品详情页系统的整套架构

#### 138\_商品详情页动态渲染系统：redis复习以及twemproxy基础知识讲解

1、部署redis

tar -zxvf redis-2.8.19.tar.gz --版本过旧，实际在做类似这种nginx+lua生产环境的部署的时候，不一定用最新的版本就是最好，老版本一般比较稳定，nginx+lua整合，用老点的版本，会比较保险一些

cd redis-2.8.19

make

nohup /usr/local/redis-test/redis-2.8.19/src/redis-server /usr/local/redis-test/redis-2.8.19/redis.conf &

ps -aux | grep redis

/usr/local/redis-test/redis-2.8.19/src/redis-cli -p 6379

set k1 v1

get k1

2、twemproxy部署

yum install -y autoconf automake libtool

yum remove -y autoconf --直接将autoconf和automake、libtool都删除掉了

wget ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz

tar -zxvf autoconf-2.69.tar.gz

cd autoconf-2.69

./configure --prefix=/usr

make && make install

wget http://ftp.gnu.org/gnu/automake/automake-1.14.tar.gz

tar -zxvf automake-1.14.tar.gz

cd automake-1.14

./bootstrap.sh

./configure --prefix=/usr

make && make install

wget http://ftpmirror.gnu.org/libtool/libtool-2.4.2.tar.gz

tar -zxvf libtool-2.4.2.tar.gz

cd libtool-2.4.2

./configure --prefix=/usr

make && make install

tar -zxvf twemproxy-0.4.0.tar.gz

cd twemproxy-0.4.0

autoreconf -fvi

./configure && make

vi /usr/local/twemproxy-test/twemproxy-0.4.0/conf/nutcracker.yml

server1:

listen: 127.0.0.1:1111

hash: fnv1a\_64

distribution: ketama

redis: true

servers:

- 127.0.0.1:6379:1

/usr/local/twemproxy-test/twemproxy-0.4.0/src/nutcracker -d -c /usr/local/twemproxy-test/twemproxy-0.4.0/conf/nutcracker.yml

ps -aux | grep nutcracker

/usr/local/redis-test/redis-2.8.19/src/redis-cli -p 1111

get k1

set k1 v2

get k1

3、redis配置

redis.conf

port 6379

logfile ""

maxmemory 100mb

maxmemory-policy volatile-lru

maxmemory-samples 10

4、redis主从

cp redis.conf redis\_6379.conf

cp redis.conf redis\_6380.conf

两份文件分别将端口设置为6379和6380

分别启动两个redis实例

nohup /usr/local/redis-2.8.19/src/redis-server /usr/local/redis-2.8.19/redis\_6379.conf &

nohup /usr/local/redis-2.8.19/src/redis-server /usr/local/redis-2.8.19/redis\_6380.conf &

ps -aux | grep redis

通过从实例客户端挂载主从

/usr/local/redis-2.8.19/src/redis-cli -p 6380

slaveof 127.0.0.1 6379

info replication

/usr/local/redis-2.8.19/src/redis-cli -p 6379

set k2 v2

/usr/local/redis-2.8.19/src/redis-cli -p 6380

get k2

5、twemproxy讲解

eshop-detail-test:

listen: 127.0.0.1:1111

hash: fnv1a\_64

distribution: ketama

timeout:1000

redis: true

servers:

- 127.0.0.1:6379:1 test-redis-01

- 127.0.0.1:6380:1 test-redis-02

eshop-detail-test: redis集群的逻辑名称

listen：twemproxy监听的端口号

hash：hash散列算法

distribution：分片算法，一致性hash，取模，等等

timeout：跟redis连接的超时时长

redis：是否是redis，false的话是memcached

servers：redis实例列表，一定要加别名，否则默认使用ip:port:weight来计算分片，如果宕机后更换机器，那么分片就不一样了，因此加了别名后，可以确保分片一定是准确的

你的客户端，java/nginx+lua，连接twemproxy，写数据的时候，twemproxy负责将数据分片，写入不同的redis实例

如果某个redis机器宕机，需要自动从一致性hash环上摘掉，等恢复后自动上线

auto\_eject\_hosts: true，自动摘除故障节点

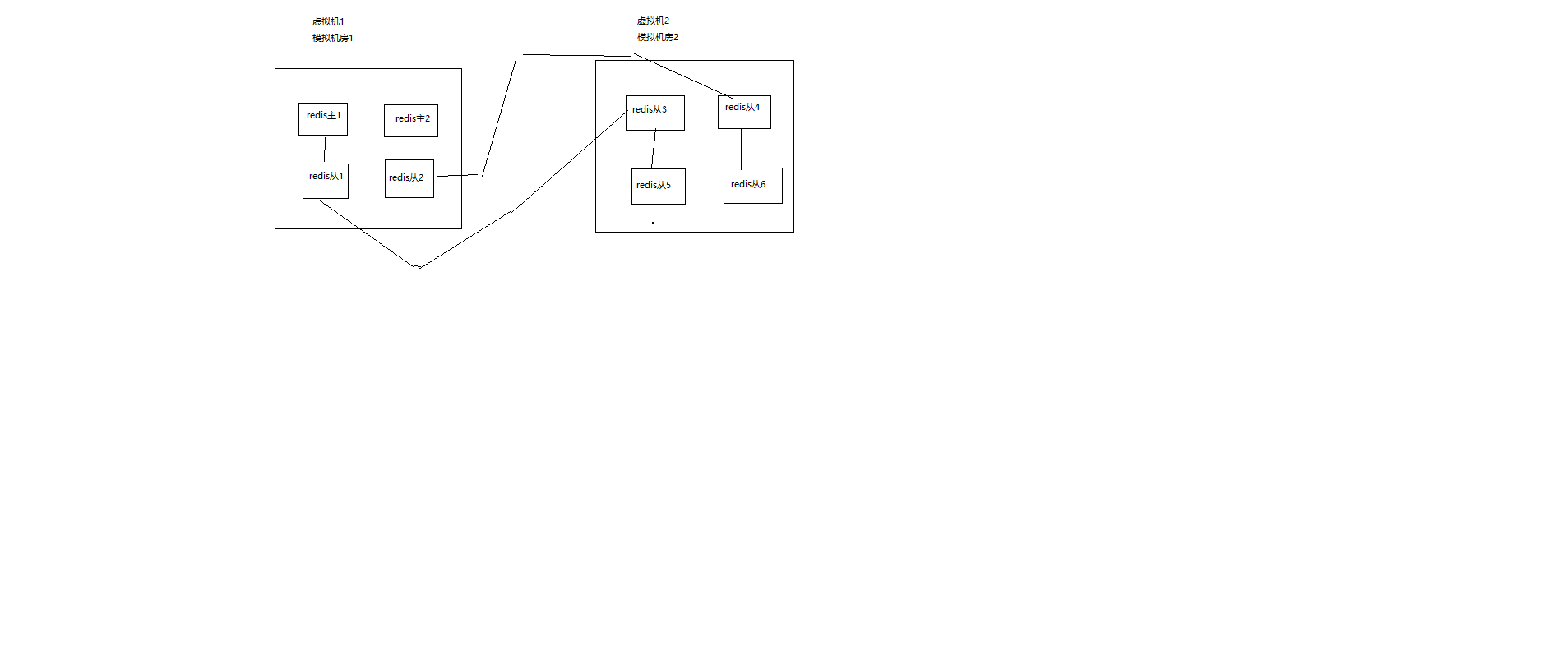
server\_retry\_timeout: 30000，每隔30秒判断故障节点是否正常，如果正常则放回一致性hash环

server\_failure\_limit: 2，多少次无响应，就从一致性hash环中摘除

一致性hash，前面讲解果了

#### 139\_商品详情页动态渲染系统：部署双机房一主三从架构的redis主集群

部署架构



在第一台虚拟机上，部署两个redis主实例+两个redis从实例，模拟一个机房的情况

在第二台虚拟机上，部署两个redis从实例，挂载到第一台虚拟机的redis从实例上; 再部署两个redis从实例，挂载到第二台虚拟机的从实例上

tar -zxvf redis-2.8.19.tar.gz

cd redis-2.8.19

make

master01.conf

master02.conf

slave01.conf

slave02.conf

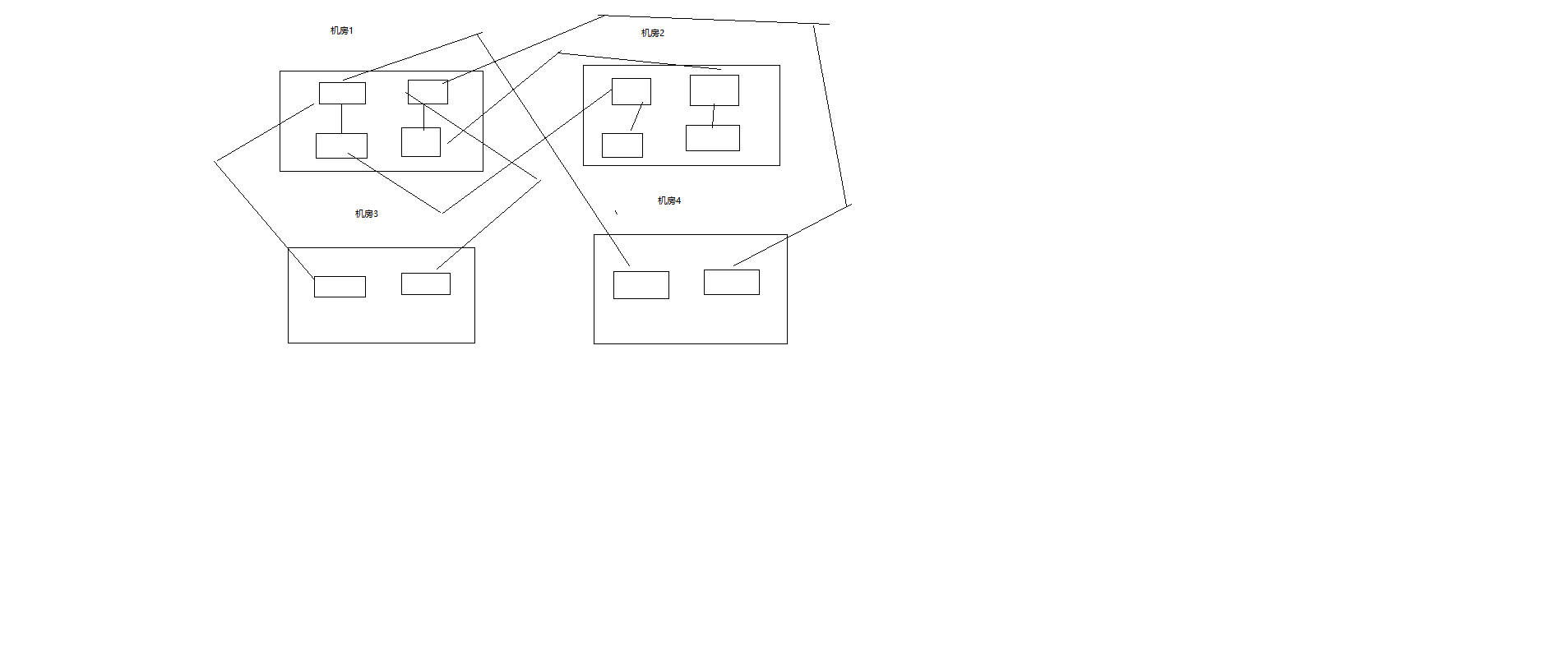
nohup src/redis-server ../redis.conf &

src/redis-cli -p 6379

slaveof

#### 140\_商品详情页动态渲染系统：给每个机房部署一个redis从集群

部署架构



空

#### 141\_商品详情页动态渲染系统：为redis主集群部署twemproxy中间件

yum install -y autoconf automake libtool

yum remove -y autoconf --直接将autoconf和automake、libtool都删除掉了

wget ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz

tar -zxvf autoconf-2.69.tar.gz

cd autoconf-2.69

./configure --prefix=/usr

make && make install

wget http://ftp.gnu.org/gnu/automake/automake-1.14.tar.gz

tar -zxvf automake-1.14.tar.gz

cd automake-1.14

./bootstrap.sh

./configure --prefix=/usr

make && make install

wget http://ftpmirror.gnu.org/libtool/libtool-2.4.2.tar.gz

tar -zxvf libtool-2.4.2.tar.gz

cd libtool-2.4.2

./configure --prefix=/usr

make && make install

tar -zxvf twemproxy-0.4.0.tar.gz

cd twemproxy-0.4.0

autoreconf -fvi

./configure && make

vi conf/nutcracker.yml

server1:

listen: 127.0.0.1:1111

hash: fnv1a\_64

distribution: ketama

redis: true

servers:

- 127.0.0.1:6379:1

src/nutcracker -d -c ../conf/nutcracker.yml

ps -aux | grep nutcracker

src/redis-cli -p 1111

get k1

set k1 v2

get k1

#### 142\_商品详情页动态渲染系统：为每个机房的redis从集群部署twemproxy中间件

yum install -y autoconf automake libtool

yum remove -y autoconf --直接将autoconf和automake、libtool都删除掉了

wget ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz

tar -zxvf autoconf-2.69.tar.gz

cd autoconf-2.69

./configure --prefix=/usr

make && make install

wget http://ftp.gnu.org/gnu/automake/automake-1.14.tar.gz

tar -zxvf automake-1.14.tar.gz

cd automake-1.14

./bootstrap.sh

./configure --prefix=/usr

make && make install

wget http://ftpmirror.gnu.org/libtool/libtool-2.4.2.tar.gz

tar -zxvf libtool-2.4.2.tar.gz

cd libtool-2.4.2

./configure --prefix=/usr

make && make install

tar -zxvf twemproxy-0.4.0.tar.gz

cd twemproxy-0.4.0

autoreconf -fvi

./configure && make

vi conf/nutcracker.yml

server1:

listen: 127.0.0.1:1111

hash: fnv1a\_64

distribution: ketama

redis: true

servers:

- 127.0.0.1:6379:1

src/nutcracker -d -c ../conf/nutcracker.yml

ps -aux | grep nutcracker

src/redis-cli -p 1111

get k1

set k1 v2

get k1

#### 143\_商品详情页动态渲染系统：部署RabbitMQ消息中间件

我之前是想用activemq，但是实际上，现在最近几年，企业里比较流行的还是rabbitmq，因为性能等各方面都比传统的activemq要好很多

安装rabbitmq 3.6.12，当前为止，最新的rabbitmq的版本

1、安装编译工具

yum install -y ncurses ncurses-base ncurses-devel ncurses-libs ncurses-static ncurses-term ocaml-curses ocaml-curses-devel

yum install -y openssl-devel zlib-devel

yum install -y make ncurses-devel gcc gcc-c++ unixODBC unixODBC-devel openssl openssl-devel

2、安装erlang

下载erlang：http://erlang.org/download/otp\_src\_20.0.tar.gz

tar -zxvf otp\_src\_20.0.tar.gz

cd otp\_src\_20.0

./configure --prefix=/usr/local/erlang --with-ssl -enable-threads -enable-smmp-support -enable-kernel-poll --enable-hipe --without-javac

make && make install

ln -s /usr/local/erlang/bin/erl /usr/local/bin/erl

vi ~/.bashrc

ERLANG\_HOME=/usr/local/erlang

PATH=$ERLANG\_HOME/bin:$PATH

source ~/.bashrc

erl

3、安装rabbitmq

http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.12/rabbitmq-server-generic-unix-3.6.12.tar.xz

yum install -y xz

xz -d rabbitmq-server-generic-unix-3.6.12.tar.xz

tar -xvf rabbitmq-server-generic-unix-3.6.12.tar

mv rabbitmq\_server-3.6.1 rabbitmq-3.6.12

开启管理页面的插件

cd rabbitmq-3.6.1/sbin/

./rabbitmq-plugins enable rabbitmq\_management

后台启动rabbitmq server

./rabbitmq-server -detached

关闭rabbitmq server

./rabbitmqctl stop

添加管理员账号

./rabbitmqctl add\_user rabbitadmin 123456

./rabbitmqctl set\_user\_tags rabbitadmin administrator

进入管理页面

15672端口号，输入用户名和密码

#### 144\_商品详情页动态渲染系统：部署MySQL数据库

用最简单的方式装一个mysql数据库，然后后面的话，就有数据库可以用来开发了

yum install -y mysql-server

chkconfig mysqld on

service mysqld start

mysql -u root

set password for root@localhost=password('root');

mysql -uroot -proot

#### 145\_商品详情页动态渲染系统：声音小问题&课程代码二次开发&商品服务需求

前置性的东西

直到144讲，把mysql给装好，我其实用的是我的一台最近新买的电脑录制的，那个电脑比较好，内存也有24个G，录制出来的声音也很响

有很多同学都反馈，之前说视频的声音很小，我有一台老的笔记本电脑，4年前买，换过几次内存条，SSD，录制的东西，我自己听是没有任何问题的

但是我就是不知道为什么上传到龙果就是声音很小，后来龙果找了一些办法，发现是每个同学将自己的声卡升级一下，就可以保证声音响亮了

但是如果还是不行的话，有其他同学总结过几种办法，可以到我们的学员群，有个共享空间，找一下，也可以找客服老师去要，如何解决声音小的问题

win10，自动拉取一个很庞大的更新，4g，下到一半，电脑就卡死了，直接就跳转，导致我的电脑无法启动，然后只能用戴尔自己的恢复功能，恢复到出厂的设置

那个电脑很多东西就弄丢了，而且到目前为止，还是有点问题，没有办法用那个新电脑直接讲课，为了保证升级内容的录制进度

那个新电脑就慢慢拿去修了，后面所有的内容，全部用4年的老电脑继续讲解

我这边周末花了整天的时间，在老电脑上，搭建了一套升级内容的环境

唯一的问题，我怀疑可能录制出来的视频，放到龙果，还是会有问题的，声音会小，有很多办法，有很多同学用了那些办法解决了声音小的问题

如果实在不行，你只能插耳机，在比较安静的环境下去听视频，声音开到最大，那样的话，也可以听，用户体验差一些了

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

之前我们已经将课程需要的一些基础性的环境都搭建好了，第一个阶段，设计了这次完整的一套商品详情页系统架构; 第二个阶段，搭建了基础设施

可以正式开始进入业务系统的开发了

第一个版本的内容，实际上，相对来说，没有出来一个完整的商品详情页架构，这次我们要做一个完整的架构，涉及到很多的服务，商品详情页的数据，都是从各个服务来的

商品服务，价格服务，库存服务，促销服务，广告服务，推荐服务。。。可能多大十几个服务，甚至是二十多个服务

动手去开发几个正儿八经的依赖服务，商品服务，价格服务，库存服务

我这边还是要说明一点，我们尽量用真实的流程和架构去做，但是完全真实复杂的电商业务，没可能，业务代码是特别多，不可能通过一个课程去讲解一个完整的电商业务的，上百人做几年，做出来一套复杂的业务

做业务出来，也没什么意义

讲课，我希望给大家屡清楚一点，讲多少业务，讲到什么程度，我觉得业务肯定要有，要讲，但是肯定要做阉割和简化，业务讲解到能用真实的场景，将整个项目流程，和架构串起来，完全跑出来，业务讲解就到位了

很多人问我，是不是可以直接拿来做二次开发，我只能这么说，你要怎么二次开发？你是要拿到完全真实的一套电商业务系统，上百万行代码，二次开发，你请绕远路，自己去外面花钱买，10万起步，几十万，买一套源码回来。

但是我们的课程讲解出来的最后的代码，也是绝对支持二次开发，我们整个架构完全出来，环境，基础设施，开发流程，部署，架构的每个环节，填充了一些简单的业务，拿到手以后，一样可以二次开发，直接就是在这个架构里面填入你自己的系统的业务就可以了

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

商品服务，管理商品的数据

分类管理：增删改查

品牌管理：增删改查

商品基本信息管理：增删改查

商品规格管理：增删改查

商品属性管理：增删改查

商品介绍管理：编辑

写出来，这块很定是要基于数据库去做得，然后重点就是演示，这种服务如何跟商品详情页系统的架构串接起来

#### 146\_商品详情页动态渲染系统：工程师的why-how-what思考方法&价格服务说明

复杂电商里面，商品的价格，比较复杂的事情

（1）调整自己商品的价格，但是这个时候你可能需要引入很多的策略，比如说做各种限制，价格是不是可以为负数，在活动来临之际，突然提价再打折，你是不是要通过专门的价格服务去检测和限制，网络加盟商，你跟品牌之间有没有一些价格上的限制

（2）每个商品可能是可以有多个价格的，什么属性，土豪金，可能就会比较贵，范围内，你可能可以打包多个东西放一起，那么就是一个价格，可能你购买的是去年款的，那么会便宜一些，电商上，价格是个区间（256元~356元）

（3）但是在我们的课程里，我不打算做这些业务

大公司里面，会给工程师培训一些软素质，比如如何正确的思考

（1）错误的思考过程：what -> how -> why，你做了很多事情，你也一直在考虑如何做到这个事情，结果你做到之后，有一天，你突然问问自己，我为什么要做这个事情？好像没有必要啊。。。

（2）正确的思考过程：why -> how -> what，考虑一个事情，为什么要做？如何去做？具体做什么？

价格服务，很多复杂的业务

我要做一个真实的可以二次开发的复杂的价格服务的业务，我要听中华石杉老师讲这块业务，没有这个，我退钱，这个课，没有价值了

为什么要做真实的价格业务？对你有什么好处？对你没什么帮助

（1）如果你是一个做电商行业系统的一个从业人员，这些业务对你来说是小儿科，产品经理有很完善的需求文档

（2）如果你不是一个做电商行业系统的从业人员，你了解到了细枝末节的业务，对你也没什么用，你自己做了电商系统，你只有对自己所在的公司所处的行业，你工作了至少2~3年以后，才能说对一块业务是熟悉的，博客，论坛，demo级的项目

（3）培训机构，j2ee就业课程，项目，OA系统，进销存系统，里面就挑选真实系统的1%的模块，花10天时间给你讲一讲，你出去敢说自己做了OA系统？进销存系统？

对课程来说，有什么帮助？

业务，对我们，商品详情页系统，我告诉你，对这个系统来说，是么有太大的意义，因为其实无论的价格怎么变，最终就是变化之后，反馈到商品详情页面里去，让用户可以尽快看到最新的价格，至于价格是怎么变化的，我们不关心

how，修改商品价格

给一个简单的接口，可以修改商品的价格，落地到数据库中，价格跟后面的商品详情页系统架构，串接起来，时效性比较高的服务，去讲解

商品详情页上，部分时效性要求很高的数据，比如价格，库存，是通过ajax异步加载的

what，价格服务，提供一个接口，可以修改某个商品的价格，落地到数据库中，可以跟商品详情页系统架构，串接起来

#### 147\_商品详情页动态渲染系统：库存服务的场景介绍以及课程需求说明

why-how-what思考法，库存服务

要不要去做复杂的库存服务，对你个人的价值，对课程的价值，根本不需要

商品服务，之所以要做一些增删改查的操作，是因为那些东西跟商品详情页系统的影响较大，做那样的一些操作

库存服务 = 价格服务，不需要做复杂的业务，库存变化了，反应到数据库中，跟商品详情页系统架构串接起来，就ok了

库存的介绍，业务：

（1）事务性关联很大，商品的购买，就要修改库存

（2）保证库存的递减一定是事务的，不能失败，不能出错，最怕的就是系统里面库存不准确，比如一个商家都没有库存了，但是还是出了bug，导致超售，用户道歉，退款

（3）电商，退货，退款，库存增加

（4）进销存，物流等系统打通，进货，退货，增减库存

主要关注库存的增加和减少，最终的结果就ok了，我们只要关注库存显示到商品详情页上去就ok了

how&what，服务，接口，修改商品的商品库存，反应数据库中，跟商品详情页系统架构打通

#### 148\_商品详情页动态渲染系统：微服务与Spring Cloud基本介绍

微服务，龙果，在我这个升级之前，就有2个课程了

dubbo的课程，通用架构给搭建了起来，dubbo去做微服务+activemq+redis+mysql+持续集成，把一个比较通用的完整的微服务的技术架构讲解了出来

spring cloud课程，那个课程会主要讲解spring cloud的技术

升级，微服务实战，spring cloud，还有各种DevOps，docker容器，持续交付流水线，结合在这个真实的电商详情页系统项目实战中去做微服务架构

微服务的项目实战，spring cloud微服务的项目实战

另外一个重点，弥补第一个版本课程的遗憾，要将一个完整的电商详情页系统架构搭建出来，跑通，从前到后，全部搞定，架构->二次开发

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

传统架构的问题

（1）单块应用，耦合严重

（2）开发速度慢，新需求

（3）不易于扩展和重构

（4）不易于技术升级

一个java web工程，多个工程，maven整合起来，spring mvc -> spring -> mybatis

一个工程，可能就包含几十万行代码

各种业务模块，全部耦合在一个大的工程中，公用了很多的基础代码

开发速度慢，开发流程，代码管理工具，签出最新的代码副本到本地，运行一下，保证可以运行

做自己的开发，你的代码跟其他人的代码都耦合在了一起，然后可能还需要涉及到跟其他的代码要做持续集成，然后放在一起统一测试，统一回归，然后统一部署和发布

每次部署都非常重，因为你要考虑到别人的代码，依赖的一些基础环境是否ok，打包和部署，都特别慢

很可能不小心别人的代码bug，导致你的系统部署失败，回滚，你还要通知别人，跟别人一起去调试和找到它的bug在哪儿

多人写作开发一个工程，经常涉及到各种代码的冲突，解决冲突，搞出来一些bug，重新运行多大几千个，上万个单元测试，重新经历后续的所有测试环境

软件迭代开发的速度很慢，可能迭代一个小功能，就要一周时间，一个模块，半个月的时间

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

微服务架构的几大特征：

（1）足够单一的职责与功能

（2）非常的微型

（3）面向服务的思想

（4）独立开发：团队，技术选型，前后端分离，存储分离，独立部署

（5）自动化开发流程：编码，自动化测试，持续集成，自动化部署

用最简单的话来说，比如之前，可能就一个单块应用，几十个兄弟在一个代码上开发，商品模块，价格模块，库存模块，促销模块，o2o模块，全部放一起了

微服务，把几十万行的单块应用，拆分出多个服务，每个服务对应一个工程，每个工程就几百行到几千行代码

每个服务，职责很单一，负责一块事情，商品数据的管理，商品服务; 价格服务，管理复杂的价格变更的业务; 库存服务，管理复杂的库存变更的业务

微型：几百行~几千行代码

面向服务的思想：每个服务暴露出来一堆接口，然后其他人都是依赖你的服务在开发

独立开发：工程上完全独立了， 那就可以给不同的服务配置不同的团队，或者工程师去开发。商品服务是3个哥儿们在维护，价格服务是1个应届生在做，库存服务是2个哥儿们在做

不同的人就做不同的工程，维护自己不同的代码，spring mvc + spring + mybatis，php，go，c++

技术选型：mysql，mongodb，memcached，redis，hbase

每个服务都是自己的存储，单独对接自己的前端工程师，独立的部署在自己的机器上

独立开发，跟其他人没关系

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

微服务的强大作用：

（1）迭代速度：你只要管好自己的服务就行了，跟别人没关系，随便你这么玩儿，修改代码，测试，部署，都是你自己的事情，不用考虑其他人，没有任何耦合

（2）复用性：拆分成一个一个服务之后，就不需要写任何重复的代码了，有一个功能别人做好了，暴露了接口出来，直接调用不就ok了么

（3）扩展性：独立，扩展，升级版本，重构，更换技术

（4）完全克服了传统单块应用的缺点

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

微服务的缺点

（1）服务太多，难以管理

（2）微服务 = 分布式系统，你本来是一个系统，现在拆分成多块，部署在不同的服务器上，一个请求要经过不同的服务器上不同的代码逻辑处理，才能完成，这不就是分布式系统

（3）分布式一致性，分布式事务，故障+容错

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

微服务的技术栈

（1）领域驱动设计：微服务建模

你的任何业务系统都有自己独特的复杂的业务，但是这个时候就是有一个问题，怎么拆分服务？拆成哪些服务？拆成多大？每个服务负责哪些功能？

微服务的建模，模型怎么设计

领域驱动的设计思想，可以去分析系统，完成建模的设计

这里不讲解了，一定是要拿超级复杂的业务来讲解，你才能听懂，业务采取的还是比较简单的，领域驱动

至少如果你真的很了解你的业务的话，你大概也知道应该如何去拆分这个服务

（2）Spring Cloud：基础技术架构

各个服务之间怎么知道对方在哪里 -> 服务的注册和发现

服务之间的调用怎么处理，rpc，负载均衡

服务故障的容错

服务调用链条的追踪怎么做

多个服务依赖的统一的配置如何管理

（3）DevOps：自动化+持续集成+持续交付+自动化流水线，将迭代速度提升到极致

如果要将微服务的开发效率提升到最高，DevOps，全流程标准化，自动化，大幅度提升你的开发效率

（4）Docker：容器管理大量服务

微服务，一个大型的系统，可以涉及到几十个，甚至是上百个服务，比较坑，怎么部署，机器怎么管理，怎么运维

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

整个微服务技术架构，全部涉及到，全部结合我们的实际的项目，完成整套微服务架构的项目实战

#### 149\_商品详情页动态渲染系统：Spring Boot与微服务的关系以及开发回顾

第一版的内容里，spring boot也用了，但是几乎没什么介绍，主要是什么原因呢？

龙果是很多课都之前都有了，spring boot专门有了一套课程去见解的，所以我当时就没多说什么

spring cloud，很多人之前都说基于spring boot的微服务

1、Spring Boot的特点

（1）快速开发spring应用的框架

spring mvc+spring+mybatis，首先配置一大堆xml配置文件，其次部署和安装tomcat，jetty等容器，跟java web打交道

跟servlet，listener，filter，打交道

手工部署到tomcat或者jetty等容器中，发布一个web应用

spring boot，简单来说，就是看中了这种java web应用繁琐而且重复的开发流程，采用了spring之上封装的一套框架，spring boot，简化整个这个流程

尽可能提升我们的开发效率，让我们专注于自己的业务逻辑即可

（2）内嵌tomcat和jetty容器，不需要单独安装容器，jar包直接发布一个web应用

（3）简化maven配置，parent这种方式，一站式引入需要的各种依赖

（4）基于注解的零配置思想

（5）和各种流行框架，spring web mvc，mybatis，spring cloud无缝整合

2、Spring Boot和微服务

（1）spring boot不是微服务技术

（2）spring boot只是一个用于加速开发spring应用的基础框架，简化工作，开发单块应用很适合

（3）如果要直接基于spring boot做微服务，相当于需要自己开发很多微服务的基础设施，比如基于zookeeper来实现服务注册和发现

（4）spring cloud才是微服务技术

3、Spring Boot的入门开发

参照之前的库存服务的spring boot，整合搭建一个spring boot

#### 150\_商品详情页动态渲染系统：Spring Cloud之Eureka注册中心

我们大概会通过七八讲的时间，来讲解一下spring cloud这个东西，不会太深入，spring cloud课

1、什么是注册中心

（1）就是首先有一个eureka server，服务的注册与发现的中心

（2）你如果写好了一个服务，就可以将其注册到eureka server上去

（3）然后别人的服务如果要调用你的服务，就可以从eureka server上查找你的服务所在的地址，然后调用

2、Eureka基本原理

（1）服务都会注册到eureka的注册表

（2）eureka有心跳机制，自动检测服务，故障时自动从注册表中摘除

（3）每个服务也会缓存euraka的注册表，即使eureka server挂掉，每个服务也可以基于本地注册表缓存，与其他服务进行通信

（4）只不过是如果eureka server挂掉了，那么无法发布新的服务了

实验步骤

（1）启动和发布一个eureka server，注册中心，web界面可以看到所有注册的服务

（2）写一个hello world服务，注册到eureka server上去

3、eureka server

（1）pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!--eureka server -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

<!-- spring boot test-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RC1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

（2）Application

@EnableEurekaServer

@SpringBootApplication

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

（3）application.yml

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

（4）访问8761端口

4、eureka client

（1）pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RC1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

（2）Application

@SpringBootApplication

@EnableEurekaClient

@RestController

public class SayHelloServiceApplication {

public static void main(String[] args) {

SpringApplication.run(SayHelloServiceApplication.class, args);

}

@Value("${server.port}")

String port;

@RequestMapping("/sayHello")

public String home(@RequestParam String name) {

return "hello, " + name + " from port:" +port;

}

}

（3）application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8762

spring:

application:

name: service-say-hello

（4）查看eureka server界面

（5）http://localhost:8762/sayHello?name=leo

#### 151\_商品详情页动态渲染系统：Spring Cloud之Ribbon+Rest调用负载均衡

上一讲，我们已经知道如何发布一个eureka注册中心，然后也知道如何发布一个服务注册到eureka server

这一讲，就来学习，如何开发另外一个服务，来通过eureka server发现其他服务，并且调用其他服务，通过ribbon+rest，RestTemplate调用rest服务接口，ribbon多个服务实例的负载均衡

1、将say-hello-service的port修改为8673，再启动一个实例

在生产环境中，肯定是一个服务会发布在多台机器上，每个机器上发布的服务，就是一个服务实例，多个服务实例实际上就组成了一个集群

2、创建一个新的工程，叫做greeting-service

（1）pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RC1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

（2）application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8764

spring:

application:

name: greeting-service

（3）Application

// @EnableDiscoveryClient，向eureka注册自己作为一个服务的调用client

// 之前的服务，EnableEurekaClient，代表的是向eureka注册自己，将自己作为一个服务

@SpringBootApplication

@EnableDiscoveryClient

public class GreetingServiceApplication {

public static void main(String[] args) {

SpringApplication.run(GreetingServiceApplication.class, args);

}

// 在spring容器中注入一个bean，RestTemplate，作为rest服务接口调用的客户端

// @LoadBalanced标注，代表对服务多个实例调用时开启负载均衡

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

（4）调用say-hello-service

// 写一个服务，注入RestTemplate服务调用客户端

@Service

public class GreetingService {

@Autowired

private RestTemplate restTemplate;

// 用SAY-HELLO-SERVICE这个服务名替代实际的ip地址

// ribbon负载在多个服务实例之间负载均衡，每次调用随机挑选一个实例，然后替换服务名

public String greeting(String name) {

return restTemplate.getForObject("http://SAY-HELLO-SERVICE/sayHello?name="+name, String.class);

}

}

（5）controller

@RestController

public class GreetingControler {

@Autowired

private GreetingService greetingService;

@RequestMapping(value = "/greeting")

public String greeting(@RequestParam String name){

return greetingService.greeting(name);

}

}

（6）多次访问http://localhost:8764/greeting?name=leo，发现每次访问的端口都不一样，在多个服务实例之间负载均衡了

#### 152\_商品详情页动态渲染系统：Spring Cloud之Fegion声明式服务调用

ribbon+rest是比较底层的调用方式，其实一般不常用

fegion，声明式的服务调用，类似于rpc风格的服务调用，默认集成了ribbon做负载均衡，集成eureka做服务发现

用fegion来重构greeting-service

1、pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RC1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

2、application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8764

spring:

application:

name: greeting-service

3、Application

@SpringBootApplication

@EnableDiscoveryClient

@EnableFeignClients

public class GreetingServiceApplication {

public static void main(String[] args) {

SpringApplication.run(GreetingServiceApplication.class, args);

}

}

4、声明远程服务接口

@FeignClient(value = "say-hello-service")

public interface SayHelloService {

@RequestMapping(value = "/sayHello",method = RequestMethod.GET)

String sayHello(@RequestParam(value = "name") String name);

}

5、controller

@RestController

public class GreetingController {

@Autowired

private SayHelloService sayHelloService;

@RequestMapping(value = "/greeting", method = RequestMethod.GET)

public String greeting(@RequestParam String name){

return sayHelloService.sayHello(name);

}

}

#### 153\_商品详情页动态渲染系统：Spring Cloud之Hystrix熔断降级

微服务架构，很重要的就是多个服务之间互相调用，很可能某个服务就死了，然后依赖它的其他服务调用大量超时，最后耗尽资源，继续死，最终导致整个系统崩盘

hystrix去做资源隔离，限流，熔断，降级

1、让greeting-service支持hystrix

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

（2）application.xml

feign.hystrix.enabled=true

（3）SayHelloService

@FeignClient(value = "say-hello-service", fallback = SayHelloServiceFallback.class)

public interface SayHelloService {

@RequestMapping(value = "/sayHello", method = RequestMethod.GET)

public String sayHello(@RequestParam(value = "name") String name);

}

（4）SayHelloServiceFallback

@Component

public class SayHelloServiceFallback implements SayHelloService {

@Override

public String sayHello(String name) {

return "error, " + name;

}

}

（5）先保持SayHelloService启动，可以访问; 关闭SayHelloSerivce，再访问，调用失败直接走降级

包括限流，自动熔断，调用失败，都会走降级

2、hystrix dashboard

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

（2）Application

@EnableHystrixDashboard

@EnableCircuitBreaker

（3）http://localhost:8764/hystrix

输入http://localhost:8764/hystrix.stream和title

访问接口，会在hystrix dashboard看到访问请求

3、改造say-hello-service支持hystrix

将一个服务多个实例的指标聚合起来看，改造say-hello-service

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

（2）Application

@SpringBootApplication

@EnableEurekaClient

@RestController

@EnableHystrix

@EnableHystrixDashboard

@EnableCircuitBreaker

public class SayHelloServiceApplication {

public static void main(String[] args) {

SpringApplication.run(SayHelloServiceApplication.class, args);

}

@Value("${server.port}")

private String port;

@RequestMapping("/sayHello")

@HystrixCommand(fallbackMethod = "sayHelloFallback")

public String sayHello(String name) {

return "hello, " + name + " from port: " + port;

}

public String sayHelloFallback(String name) {

return "error, " + name

}

}

（3）locahost:8762/hystrix

输入locahost:8762/hystrix.stream，2000，title

访问这个接口

4、创建turbin工程，hystrix-turbine-server

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

（2）Application

@SpringBootApplication

@EnableTurbine

public class HystrixTurbineServer {

public static void main(String[] args) {

new SpringApplicationBuilder(HystrixTurbineServer.class).web(true).run(args);

}

}

（3）application.yml

spring:

application.name: hystrix-terbine-server

server:

port: 8765

security.basic.enabled: false

turbine:

aggregator:

clusterConfig: default

appConfig: say-hello-service

clusterNameExpression: new String("default")

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

（4）对say-hello-service每个服务实例都访问几次

http://localhost:8762/hystrix

stream输入：http://localhost:8765/turbine.stream

在dashboard可以看到两个服务实例聚合起来的指标

#### 154\_商品详情页动态渲染系统：Spring Cloud之Zuul网关路由

常规的spring cloud的微服务架构下

前端请求先通过nginx走到zuul网关服务，zuul负责路由转发、请求过滤等网关接入层的功能，默认和ribbon整合实现了负载均衡

比如说你有20个服务，暴露出去，你的调用方，如果要跟20个服务打交道，是不是很麻烦

所以比较好的一个方式，就是开发一个通用的zuul路由转发的服务，根据请求api模式，动态将请求路由转发到对应的服务

你的前端，主要考虑跟一个服务打交道就可以了

1、创建zuul-server工程

2、pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RC1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

3、Application

@EnableZuulProxy

@EnableEurekaClient

@SpringBootApplication

public class ServiceZuulApplication {

public static void main(String[] args) {

SpringApplication.run(ServiceZuulApplication.class, args);

}

}

4、application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8766

spring:

application:

name: zuul-server

zuul:

routes:

say-hello:

path: /say/hello/\*\*

serviceId: say-hello-service

greeting:

path: /greeting/\*\*

serviceId: greeting-service

5、修改代码

在greeting-service中的返回值加入自己的标识

6、运行，依次走两种不同的api接口，zuul会路由到不同的服务上去

7、请求过滤

@Component

public class UserLoginFilter extends ZuulFilter {

private static Logger logger = LoggerFactory.getLogger(UserLoginFilter.class);

// pre，routing，post，error

@Override

public String filterType() {

return "pre";

}

// 顺序

@Override

public int filterOrder() {

return 0;

}

// 根据逻辑判断是否要过滤

@Override

public boolean shouldFilter() {

return true;

}

@Override

public Object run() {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest request = ctx.getRequest();

log.info(String.format("%s >>> %s", request.getMethod(), request.getRequestURL().toString()));

Object userId = request.getParameter("userId");

if(userId == null) {

log.warn("userId is empty");

ctx.setSendZuulResponse(false);

ctx.setResponseStatusCode(401);

try {

ctx.getResponse().getWriter().write("userId is empty");

}catch (Exception e){}

return null;

}

log.info("ok");

return null;

}

}

#### 155\_商品详情页动态渲染系统：Spring Cloud之Config统一配置中心

多个服务共享相同的配置，举个例子，数据库连接，redis连接，还有别的一些东西，包括一些降级开关，等等

用config统一配置中心

1、创建工程config-server

2、pom.xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Camden.SR6</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

2、Application

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

3、公开的git仓库

spring cloud config，配置文件，用的是properties的格式，基于git去做

账号：roncoo-eshop

密码：roncoo123456

仓库地址：https://github.com/roncoo-eshop/roncoo-eshop-config

git怎么用，我不讲解了，自己百度或者找资料

3、application.properties

spring.application.name=config-server

server.port=8767

spring.cloud.config.server.git.uri=https://github.com/roncoo-eshop/roncoo-eshop-config

spring.cloud.config.server.git.searchPaths=config-file

spring.cloud.config.label=master

spring.cloud.config.server.git.username=roncoo-eshop

spring.cloud.config.server.git.password=roncoo123456

4、访问http://localhost:8767/name/dev

5、重构greeting-service

配置一个默认的name，如果前端没有传递name参数，直接取用默认的name

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

（2）application.yml -> bootstrap.properties

spring.application.name=config-client

spring.cloud.config.label=master

spring.cloud.config.profile=dev

spring.cloud.config.uri= http://localhost:8767/

server.port=8764

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

feign.hystrix.enabled=true

（3）controller

@Value("${defaultName}")

private String defaultName;

没有传递name的时候，默认用spring cloud config中配置的name

#### 156\_商品详情页动态渲染系统：Spring Cloud之Sleuth调用链路追踪

在一个微服务系统中，一个请求过来，可能会经过一个很复杂的调用链路，经过多个服务的依次处理，才能完成

在这个调用链路过程中，可能任何一个环节都会出问题，所以如果要进行一些问题的定位，那么就要对每个调用链路进行追踪

sleuth

1、搭建sleuth server

（1）创建工程：sleuth-server

（2）pom.xml

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-server</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-ui</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Camden.SR6</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

（3）Application

@SpringBootApplication

@EnableZipkinServer

public class SleuthServer {

public static void main(String[] args) {

SpringApplication.run(SleuthServer.class, args);

}

}

（4）application.yml

server.port=9411

2、在say-hello-service和greeting-service中加入sleuth支持

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

（2）application.yml

spring.zipkin.base-url=http://localhost:9411

3、调用接口，查看http://localhost:9411

#### 157\_商品详情页动态渲染系统：Spring Cloud之Eureka Server安全认证

1、pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

2、application.yml

security:

basic:

enabled: true

user:

name: admin

password: 123456

3、访问http://localhost:8761/，需要输入用户名和密码

基于spring cloud提供的一整套技术：服务注册与发现，声明式的服务调用，熔断降级，网关路由，统一配置，链路追踪

来开发我们的商品详情页系统

#### 158\_商品详情页动态渲染系统：完成Spring Boot+Spring Cloud+MyBatis整合

接下来就要进入正式的业务系统开发，各种设计思路，各种基础环境的搭建（课程中的生产环境），基础知识（spring boot，spring cloud）

体会到我之前的苦心，为什么第一版的内容，一开始做的时候，没有想要做一个特别大而全的系统架构，就是找准里面几个点深入去讲解呢？因为完整的架构涉及到的东西太多了。要讲解的细枝末节特别多，满足各位同学的心愿，微服务的项目实战，商品详情页系统就涉及到很多服务，服务之间的高可用。

但是，我们尽可能将需要的环境，比如说mysql，rabbitmq，redis，尽量在本地都部署一份，平时开发，就在本地玩儿就可以了

我这个是老电脑，12G内存，2台虚拟机启动，6g，再启动一个eclipse，里面跑一大堆的服务，12g内存几乎吃满

尽可能讲课的过程中，不要又是启动虚拟机，又是启动eclipse跑服务，尽可能就是要么在本机开发，测试，相当于是我们平时工作中的开发环境

要么就是最终完全课程讲完之后，全部使用微服务的方式部署各种服务到虚拟机模拟的生产环境中去，跑通整个流程

0、在windows上安装mysql数据库，方便测试和开发

我本来这个老电脑上，本机就有一个mysql数据库，大家自己装的话，百度一下，官网下载个安装包，作为windows服务启动即可

再下载一个SQLYog数据库管理工具

spring boot+spring cloud，本身自带了spring web mvc的支持，mybatis整合起来，可以操作数据库

1、pom.xml

<dependency>

<groupId>mysql</groupId>

<artifactId>artifactId</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.18</version>

</dependency>

2、application.yml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource

platform: mysql

url: jdbc:mysql://192.168.31.223:3306/eshop

username: root

password: root

3、User

public class User {

private Long id;

private String name;

private Integer age;

}

4、UserMapper

@Mapper

public interface UserMapper {

@Select("select \* from users")

List<User> findAllUsers();

}

5、UserService

@Service

public class UserService {

@Autowired

private UserMapper userMapper;

public List<User> findAllUsers() {

return userMapper.findAllUsers();

}

}

6、UserController

@RestController

public class UserController {

@Autowrite

private UserService userService;

@RequestMapping("/findAllUsers")

@ResponseEntity

public List<User> findAllUsers() {

return userService.findAllUsers();

}

}

跟mybatis整合成功，可以操作数据库

#### 159\_商品详情页动态渲染系统：基于Spring Cloud开发商品服务（一）

1、基于spring cloud搭建一个商品服务，跑通基本架构

spring boot + spring cloud + spring mvc + spring + mybatis

2、在里面进行业务代码开发

CRUD，而且我还做了很多的简化，这里说一点，有些同学也是正儿八经的做这个电商系统行业的，所以本身其实就对电商业务很了解

如果你对这个电商业务很了解的话，业务简化也没关系

不少人提，为什么没有真实的业务，课程，几十个小时，核心

吐槽，为什么不带着我们做复杂的电商数据库的建模，好像是有人提过，玩儿那种特别复杂的电商业务建模，上百张表

这个课程，架构，技术，商品详情页系统，电商系统

分类管理：增删改查

品牌管理：增删改查

商品基本信息管理：增删改查

商品规格管理：增删改查

商品属性管理：增删改查

商品介绍管理：编辑

#### 160\_商品详情页动态渲染系统：基于Spring Cloud开发商品服务（二）

包括后面要讲解的商品介绍，分段存储和分段ajax加载，也不讲解图片，存储的都是一些图片的名字，有专门的图片服务器

商品属性管理：增删改查

商品规格管理：增删改查

商品介绍管理：编辑

在很多的大公司里，业务的价值，在于你对这个领域的理解，你是一个领域专家，对业务的理解，技术促进业务

实际上，对于工程师而言，在一个大的系统中，可能就是架构师设计架构，玩儿的是技术含量最高的，你在里面写业务代码，CRUD

#### 161\_商品详情页动态渲染系统：基于Spring Cloud开发价格服务

单开服务，修改商品价格

#### 162\_商品详情页动态渲染系统：基于Spring Cloud开发库存服务

单开服务，修改商品库存

#### 163\_商品详情页动态渲染系统：windows部署rabbitmq作为开发测试环境

之前在虚拟机linux上安装了rabbitmq，作为生产环境

在本地windows上，我们需要去搭建一套本地的开发和测试环境，数据库已经安装了，rabbitmq

从下一讲开始，我们其实就可以去将数据写入rabbitmq，然后开发同步服务，聚合服务，等等从rabbitmq里面消费数据，写入redis

1、安装erlang

下载erlang：http://www.erlang.org/downloads，otp\_win64\_20.0.exe，直接安装

在windows设置环境变量

ERLANG\_HOME=/usr/local/erlang

PATH=$ERLANG\_HOME/bin:$PATH

erl

3、安装rabbitmq

http://www.rabbitmq.com/download.html

cd rabbitmq-3.6.1/sbin/

rabbitmq-plugins.bat enable rabbitmq\_management

后台启动rabbitmq server

添加管理员账号

./rabbitmqctl add\_user rabbitadmin 123456

./rabbitmqctl set\_user\_tags rabbitadmin administrator

进入管理页面

15672端口号，输入用户名和密码

#### 164\_商品详情页动态渲染系统：windows部署redis作为开发测试环境

我们之前讲解过了，咱们因为是在我的旧电脑上玩儿，12g内存，每次都开几台虚拟机，然后启动eclipse，启动一大堆spring cloud服务，内存吃不消

所以开发、测试环境与生产部署环境隔离，虚拟机上部署的一套作为生产环境

windows上，我们启动eclipse+一大堆服务，同时将rabbitmq，mysql，redis，等，全部在windows上部署，作为咱们的测试环境

这一讲，在windows上部署一个最基本的单实例的redis，可以用就行了，可以写kv存储就可以了

上生产环境，也是直接连接twemproxy，连接twemproxy中间件+redis集群，跟本地连接单个的redis实例，是一个意思

原生的redis是不支持windows的，但是微软搞了一个可以在windows上部署的redis版本，供我们学习、开发以及测试来使用的

#### 165\_商品详情页动态渲染系统：依赖服务将数据变更消息写入rabbitmq或双写redis

1、基于spring boot整合rabbitmq的发送与消费

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

（2）application.yml

spring:

rabbitmq:

host: localhost

port: 15672

username: rabbitadmin

password: 123456

（3）生产者

@Component

public class RabbitMQSender {

@Autowired

private AmqpTemplate rabbitTemplate;

public void send(String message) {

this.rabbitTemplate.convertAndSend("my-queue", message);

}

}

（4）消费者

@Component

@RabbitListener(queues = "my-queue")

public class RabbitMQReceiver {

@RabbitHandler

public void process(String message) {

System.out.println("从my-queue队列接收到一条消息：" + message);

}

}

（5）在rabbitmq管理界面中创建队列

rabbitmqctl set\_permissions -p / rabbitadmin '.\*' '.\*' '.\*'

在web界面中，到admin下，点击一下那个用户，set一下permission，就可以创建queue了

2、spring boot整合redis

（1）pom.xml

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.1.43</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

（2）Application

@Bean

public Jedis jedis() {

JedisPoolConfig config = new JedisPoolConfig();

config.setMaxActive(100);

config.setMaxIdle(5);

config.setMaxWait(1000 \* 100);

config.setTestOnBorrow(true);

pool = new JedisPool(config, "localhost", 6379);

}

3、商品服务数据变更，将消息写入rabbitmq

时效性比较低的数据，走rabbitmq，然后后面就接着整套动态渲染系统去玩儿

4、价格服务和库存服务，数据变更，直接将数据双写到redis中

时效性比较高的数据，直接mysql+redis双写，不走冬天渲染系统，写到redis之后，后面走OneService服务提供页面的ajax调用

#### 166\_商品详情页动态渲染系统：基于Spring Cloud开发数据同步服务

1、基于spring boot整合rabbitmq的发送与消费

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

（2）application.yml

spring:

rabbitmq:

host: localhost

port: 15672

username: rabbitadmin

password: 123456

（3）生产者

@Component

public class RabbitMQSender {

@Autowired

private AmqpTemplate rabbitTemplate;

public void send(String message) {

this.rabbitTemplate.convertAndSend("my-queue", message);

}

}

（4）消费者

@Component

@RabbitListener(queues = "my-queue")

public class RabbitMQReceiver {

@RabbitHandler

public void process(String message) {

System.out.println("从my-queue队列接收到一条消息：" + message);

}

}

（5）在rabbitmq管理界面中创建队列

2、spring boot整合redis

（1）pom.xml

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.1.43</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

（2）Application

@Bean

public Jedis jedis() {

return new Jedis("127.0.0.1", 6379);

}

3、基于spring cloud开发数据同步服务

（1）接收到增删改消息

（2）直接基于Fegion调用依赖服务接口，拉取数据，对redis原子数据进行增删改操作

（3）再将数据变更消息按照维度发送到rabbitmq

4、基于spring cloud开发数据聚合服务

（1）接收到数据变更消息

（2）按照维度从redis中获取数据，聚合成一个维度数据，写入redis中维度聚合数据

#### 167\_商品详情页动态渲染系统：基于Spring Cloud开发数据聚合服务

是这样，数据同步服务，就是去接收各个依赖服务发送过来的某个原子数据的变更消息，然后将原子数据通过fegion调用依赖服务的接口拉取过来

然后写入redis中

接着再将某个维度数据的变更消息发送到另外一个queue中

数据聚合服务，就是去监听另外一个queue，得到某个维度变化的消息之后，就从redis中将这个维度的数据全部读取出来，然后拼成一个大的聚合json串

1、基于spring boot整合rabbitmq的发送与消费

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

（2）application.yml

spring:

rabbitmq:

host: localhost

port: 15672

username: rabbitadmin

password: 123456

（3）生产者

@Component

public class RabbitMQSender {

@Autowired

private AmqpTemplate rabbitTemplate;

public void send(String message) {

this.rabbitTemplate.convertAndSend("my-queue", message);

}

}

（4）消费者

@Component

@RabbitListener(queues = "my-queue")

public class RabbitMQReceiver {

@RabbitHandler

public void process(String message) {

System.out.println("从my-queue队列接收到一条消息：" + message);

}

}

（5）在rabbitmq管理界面中创建队列

2、spring boot整合redis

（1）pom.xml

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.1.43</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

（2）Application

@Bean

public Jedis jedis() {

return new Jedis("127.0.0.1", 6379);

}

3、基于spring cloud开发数据同步服务

（1）接收到增删改消息

（2）直接基于Fegion调用依赖服务接口，拉取数据，对redis原子数据进行增删改操作

（3）再将数据变更消息按照维度发送到rabbitmq

4、基于spring cloud开发数据聚合服务

（1）接收到数据变更消息

（2）按照维度从redis中获取数据，聚合成一个维度数据，写入redis中维度聚合数据

#### 168\_商品详情页动态渲染系统：完成数据同步服务与数据聚合服务的测试

2310879776

#### 169\_商品详情页动态渲染系统：消息队列架构升级之去重队列

已经做好了什么

依赖服务，一个商品服务走动态渲染系统，另外两个价格服务和库存服务走mysql+redis双写+OneService系统+页面Ajax

商品服务（增删改查各种数据） -> 发送数据变更消息到queue -> 数据同步服务+原子数据更新到redis中 -> 发送维度数据变更消息到queue -> 数据聚合服务+将原子数据从redis中查询出来按照维度聚合后写入redis

对这个里面的一些细节做一些架构上的优化和升级

消息队列，rabbitmq，去重队列

动态渲染系统，就说明了，数据更新之后，要反馈到页面中，时效性并不是太高，可以接受几秒钟甚至是几分钟的延迟

在这里为了减少后面的系统，比如说数据聚合服务的压力，可以做一些优化，去重队列

数据同步服务里面，完全可以不用每次立即发送维度数据变更消息，可以将维度数据变更消息采用set的方式，在内存中先进行去重

开启一条后台线程，每隔5分钟，每隔1分钟，每隔5秒钟，将set中的数据拿出来，发送到下一个queue中，然后set中的数据清除掉

比如某一个维度数据，product，商品属性，商品规格，商品本身，短时间内变更了，可能就会发送3条一模一样的维度数据变更消息到下一个queue

数据聚合服务短时间内要执行3次聚合操作，压力比较大， 给redis带来的压力也比较大

如果做了去重之后，3个维度数据变更消息会在数据同步服务的内存中先去重，然后几分钟之后才会发送一条消息到下一个queue，数据聚合服务执行一次聚合操作即可

2310879776

#### 170\_商品详情页动态渲染系统：消息队列架构升级之刷数据与高优先级队列

刷数据的问题

业务系统，特别是在快速迭代过程中，可能会因为一些代码的bug，或者要上线一些新功能，导致需要对全量的数据刷一遍数据

本来有个字段，status，0，1，2这样的值，但是现在要将所有数据，status这个字段的值刷为OPEN，CLOSED，SEND之类的状态

刷数据，一般是在晚上凌晨的时候执行的，依赖服务会大量的更新数据，大量的刷数据的请求会到我们的消息队列中

此时我们的系统压力会非常的大

甚至可能会影响夜间一些正常用户的购买行为，等等

所以一般对这个问题是这样的，我们会针对刷数据的问题，单独开出来队列，专门处理刷数据的请求，对这些队列的消费，通常来说，只会在凌晨0点之后才开始执行

这样的话呢，好处在于，正常的消息不会跟刷数据的消息混杂在一个队列中，可以拆分到不同的队列中

高优先级问题

大家可以这么想，有些数据，比如说，一些特别紧急的活动对应的数据，需要尽快的反应到页面中，那么此时，一个道理，如果这种高优先级的数据

跟其他的消息混杂的一个队列中，势必需要去等待队列中其他的普通消息先处理完了，才能轮到自己

所以一般来说，也会单开高优先级的队列，然后如果业务系统有高优先级的消息，直接写到高优先的队列中，这样的话呢，后续流程全部单独处理

2310879776

#### 171\_商品详情页动态渲染系统：吞吐量优化之批量调用依赖服务接口

2310879776

#### 172\_商品详情页动态渲染系统：吞吐量优化之redis mget批量查询数据

2310879776

#### 173\_商品详情页动态渲染系统：在分发层nginx部署流量分发的lua脚本

用之前做好的lua脚本修改一下

2310879776

#### 174\_商品详情页动态渲染系统：完成应用层nginx的lua脚本的编写与部署

app.conf

lua\_shared\_dict my\_cache 128m;

local function close\_redis(red)

if not red then

return

end

local pool\_max\_idle\_time = 10000

local pool\_size = 100

local ok, err = red:set\_keepalive(pool\_max\_idle\_time, pool\_size)

if not ok then

ngx.say("set keepalive error : ", err)

end

end

local redis = require("resty.redis")

local red = redis:new()

red:set\_timeout(1000)

local ip = "127.0.0.1"

local port = 6660

local ok, err = red:connect(ip, port)

if not ok then

ngx.say("connect to redis error : ", err)

return close\_redis(red)

end

local resp, err = red:get("msg")

if not resp then

ngx.say("get msg error : ", err)

return close\_redis(red)

end

if resp == ngx.null then

resp = ''

end

ngx.say("msg : ", resp)

close\_redis(red)

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http\_headers.lua

wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http.lua

cd /usr/hello/lualib/resty/

wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template.lua

mkdir /usr/hello/lualib/resty/html

cd /usr/hello/lualib/resty/html

wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template/html.lua

在app.conf的server中配置模板位置

set $template\_location "/templates";

set $template\_root "/usr/servers/distribution\_nginx/distribution\_app/templates";

product id: {\* productId \*}<br/>

product name: {\* productName \*}<br/>

product picture list: {\* productPictureList \*}<br/>

product specification: {\* productSpecification \*}<br/>

product service: {\* productService \*}<br/>

product color: {\* productColor \*}<br/>

product size: {\* productSize \*}<br/>

shop id: {\* shopId \*}<br/>

shop name: {\* shopName \*}<br/>

shop level: {\* shopLevel \*}<br/>

shop good cooment rate: {\* shopGoodCommentRate \*}<br/>

2310879776

#### 175\_商品详情页动态渲染系统：基于Spring Cloud开发数据直连服务

如果nginx本地，走nginx local cache没有，在本机房的通过twemproxy读本机房的从集群，如果还是没有，则发送http请求给数据直连服务

数据直连服务，先在自己本地读取ehcache（有的时候也可以忽略，我这里就不做了，因为之前已经做过了），读redis主集群，通过fegion拉取依赖服务的接口

将数据写入主集群中，主集群会同步到各个机房的从集群，同时数据直连服务将获取到的数据返回给nginx，nginx会写入自己本地的local cache

2310879776

#### 176\_商品详情页动态渲染系统：完成多级缓存全链路的测试多个bug修复

为什么单独拉这么一讲，测试出来，这次讲解，有好几讲，都是单独的测试，调试

因为上次在我们的群里，好多同学居然跟我提出抗议，之前讲解的时候，每次遇到的问题，直接就暂停，一会儿就好了？？？？

他们没看到我调试的过程，是一种损失

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

</head>

<body>

product info: {\* productInfo \*}

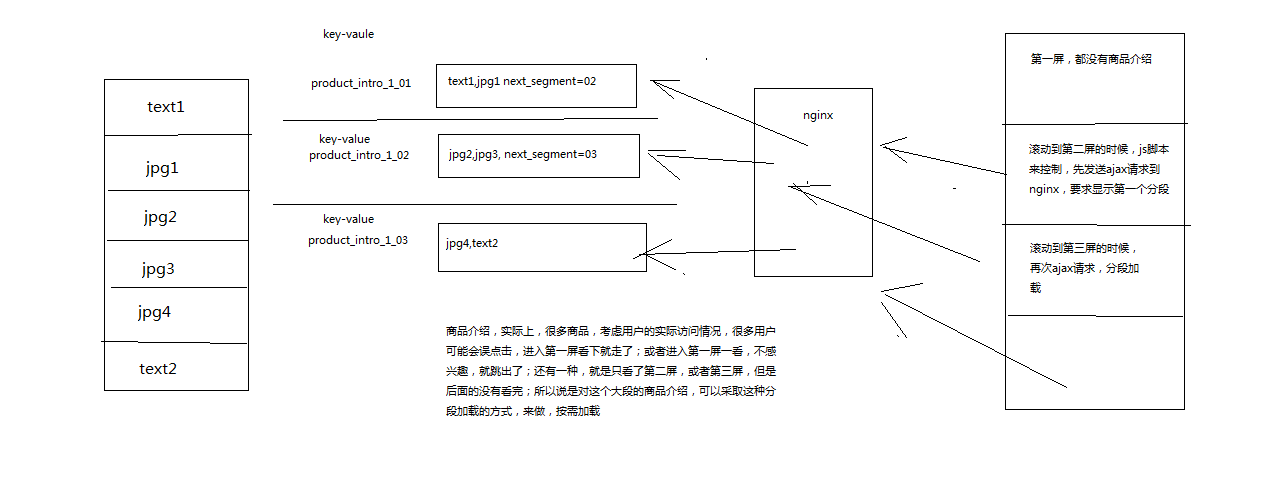
</body>

</html>

2310879776

#### 177\_商品详情页动态渲染系统：商品介绍分段存储以及分段加载的介绍

商品介绍分段加载思路介绍



商品介绍，product\_intro，里面可能包含大段的文字，还有大量的图片

存储的时候，完全可以将大段的东西，分段来存储，因为一般最好不要将一个特别大的value存储到redis中

2310879776

#### 178\_商品详情页动态渲染系统：高可用架构优化之读链路多级降级思路介绍

读链路：nginx local cache -> 本机房redis从集群 -> 数据直连服务的jvm堆缓存（之前讲解，这次没做） -> 其他机房redis主集群 -> 依赖服务

读链路的降级

本机房redis从集群可能会挂掉，可能性会大一些：降级为直接连数据直连服务

数据直连服务也可能会挂掉：降级为跨机房直接连redis主集群

t1 = os.time();

t2 = os.time();

os.difftime(t2, t1)

if not resp then

ngx.say("request error :", err)

return

end

2310879776

#### 179\_商品详情页动态渲染系统：高可用架构优化之hystrix隔离与降级

这块说，hystrix之前已经深入讲解过了，在spring cloud里面，fegion去掉用其他服务接口的操作，在这里肯定是要将hystrix跟fegion整合起来的

这样的话，对后端的依赖服务的接口才能做资源隔离，不至于说某一个依赖服务故障，拖垮整个服务，有一个服务故障的时候，可以自动降级

hystrix，还可以自动做限流

依赖服务故障过多，限流，熔断，降级

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 5000

2310879776

#### 180\_商品详情页动态渲染系统：部署jenkins持续集成服务器

上传到github上面去，然后部署jenkins持续集成服务器，在jenkins上配置自动化部署的脚本

每次部署，都是在jenkins上执行构建，jenkins自动从github上面拉取代码，用maven打包，打成jar包，创建一个docker镜像，通过docker容器来部署spring cloud微服务

然后就可以通过这种自动化流水线的方式部署所有的微服务

所有的微服务通过这种方式部署好了之后，可以跟我们的虚拟机中的类生产环境打通，mysql，redis，rabbitmq，全部都用虚拟机中的生产环境

相当于是我们修改商品数据，那个数据直接会进入各个机房的redis从集群

我们在nginx上访问，不就可以从自己本机房的redis从集群中获取数据了么，数据直连服务都把多级缓存打通了

再装一个2g的虚拟机，用centos 7，java 8

http://updates.jenkins-ci.org/download/war/

2.78

将jenkins.war放到/usr/local中

执行java -jar httpPort=8080 jenkins.war，启动jenkins服务器

访问8080端口，可以看到jenkins的界面，需要输入秘钥

在控制台可以找到秘钥，也可以再/root/.jenkins/secrets/initialAdminPassword中找到

复制秘钥，进入界面

install suggested plugins，安装推荐的插件

创建一个管理员账户，save and finish

2310879776

#### 181\_商品详情页动态渲染系统：在CentOS 6安装和部署Docker

1、初步安装和启动docker

yum update -y

yum install -y yum-utils

yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

yum -y install docker-ce

systemctl start docker

2、设置镜像

vi /etc/docker/daemon.json

{

"registry-mirrors": ["https://aj2rgad5.mirror.aliyuncs.com"]

}

3、开放管理端口映射

vi /lib/systemd/system/docker.service

将第11行的ExecStart=/usr/bin/dockerd，替换为：

ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock -H tcp://0.0.0.0:7654

2375是管理端口，7654是备用端口

在~/.bashrc中写入docker管理端口

export DOCKER\_HOST=tcp://0.0.0.0:2375

source ~/.bashrc

4、重启docker

systemctl daemon-reload && service docker restart

5、测试docker是否正常安装和运行

docker run hello-world

Hello from Docker!

This message shows that your installation appears to be working correctly.

2310879776

#### 182\_商品详情页动态渲染系统：在CentOS 6安装maven、git以及推送github

1、安装maven

wget http://mirror.bit.edu.cn/apache/maven/maven-3/3.5.0/binaries/apache-maven-3.5.0-bin.tar.gz

tar -zxvf apache-maven-3.5.0-bin.tar.gz

vi ~/.bashrc

export MAVEN\_HOME=/usr/local/apache-maven-3.5.0

export PATH=$PATH:$MAVEN\_HOME/bin

source ~/.bashrc

mvn -version

cd /usr/local/apache-maven-3.5.0/conf

vi settings.xml

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>\*</mirrorOf>

<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>

2、安装git

yum install -y git

git --version

3、修改课程中的所有服务的配置为基于虚拟机中的生产环境

4、将代码推送到github中

2310879776

#### 183\_商品详情页动态渲染系统：通过jenkins+docker部署eureka服务

1、新建一个任务

2、构建一个自由风格的软件项目

3、配置Github，包括github地址，用户名和密码，分支

4、配置构建环境

增加构建步骤 -> invoker top-level-Maven targets

Maven version: maven3.5.0

Goals: clean package

POM: pom.xml

5、增加构建步骤 -> execute shell

#!/bin/bash

REGISTRY\_URL=192.168.189.54:2375

WORK\_DIR=/root/work\_build

PROJECT\_NAME=eureka-server

PROJECT\_VERSION=0.0.1-SNAPSHOT

if [ ! -e ${WORK\_DIR}/${PROJECT\_NAME} ] && [ ! -d ${WORK\_DIR}/${PROJECT\_NAME} ]; then

mkdir -p ${WORK\_DIR}/${PROJECT\_NAME}

echo "Create Dir: ${WORK\_DIR}/${PROJECT\_NAME}"

fi

if [ -e ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile ]; then

rm -rf ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile

echo "Remove File: ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile"

fi

cp /root/.jenkins/workspace/eureka-server/Dockerfile ${WORK\_DIR}/${PROJECT\_NAME}/

cp /root/.jenkins/workspace/eureka-server/target/\*.jar ${WORK\_DIR}/${PROJECT\_NAME}/

cd ${WORK\_DIR}/${PROJECT\_NAME}/

docker build -t ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION} .

docker push ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION}

if docker ps -a | grep ${PROJECT\_NAME}; then

docker rm -f ${PROJECT\_NAME}

echo "Remove Docker Container: ${PROJECT\_NAME}"

fi

docker run -d -p 8761:8761 --name ${PROJECT\_NAME} ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION}

6、执行一些修改

（1）修改1：在系统配置中设置maven，然后在配置中选择自己配置的maven版本，解决cannot run mvn program的错误

（2）修改2：编写DockerFile

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD eureka-server-0.0.1-SNAPSHOT.jar app.jar

#RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

EXPOSE 8761

7、执行构建

8、访问http://192.168.31.253:8761/，可以看到eureka-server的页面

2310879776

#### 184\_商品详情页动态渲染系统：twemproxy hash tag+mget优化思路介绍

我们其实twemproxy就是将Redis集群做成了很多个分片，相当于是部署了很多个redis主实例

然后通过twemproxy中间件，将数据散列存储到多个redis实例中去，每个redis实例中存储一部分的数据

但是还记得，我们用了mget那个优化操作么

product\_1

product\_property\_1

product\_specification\_1

这是3个key，可能会散列到不同的redis实例上去

此时直接用mget很坑，可能要走多次网络请求，连接多个redis实例才能获取到所有数据

所以这里的优化思路，就是用hash tag这个功能

通过hash tag，将一个商品关联的数据（可能会被mget一下子一起批量获取的数据），全部路由到同一个redis实例上去

这样的话呢，后面在获取数据的时候，直接就会路由到同一个redis实例，一下子mget出来多个数据

就不需要连接多次到不同的redis实例上去

1、twemproxy中配置一个东西，hash\_tag

server1:

listen: 127.0.0.1:1111

hash: fnv1a\_64

distribution: ketama

redis: true

hash\_tag: "::"

servers:

- 127.0.0.1:6660:1 server1

- 127.0.0.1:6661:1 server2

2、写和读，都按照hash\_tag格式来

product:1:

product\_property:1:

product\_specification:1:

写的时候，不是按照product\_1这种完整的串来计算hash值和路由的

是按照两个冒号中间的值来计算hash值和路由的

比如上面3个值，两个冒号中间都是1，那么计算出来的hash值一定是一样的，然后三个key-value对一定会路由到同一个redis实例上去

在读的时候

mget product:1: product\_property:1: product\_specification:1:

读的时候，同样是根据两个冒号中间的值来路由和读取，这样的话，3个key会路由到同一个redis实例上去，一次性全部读出来

2310879776

#### 185\_商品详情页动态渲染系统：所有服务最终修改以及jenkins+docker部署

把所有服务全部修改为生产环境的配置，然后全部代码推送带github，全部在jenkins中配置基于docker的自动化部署

整个流程串起来搞一遍看看

1、新建一个任务

2、构建一个自由风格的软件项目

3、配置Github，包括github地址，用户名和密码，分支

4、配置构建环境

增加构建步骤 -> invoker top-level-Maven targets

Maven version: maven3.5.0

Goals: clean package

POM: pom.xml

5、增加构建步骤 -> execute shell

#!/bin/bash

REGISTRY\_URL=192.168.189.54:2375

WORK\_DIR=/root/work\_build

PROJECT\_NAME=eshop-datalink-service

PROJECT\_VERSION=0.0.1-SNAPSHOT

if [ ! -e ${WORK\_DIR}/${PROJECT\_NAME} ] && [ ! -d ${WORK\_DIR}/${PROJECT\_NAME} ]; then

mkdir -p ${WORK\_DIR}/${PROJECT\_NAME}

echo "Create Dir: ${WORK\_DIR}/${PROJECT\_NAME}"

fi

if [ -e ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile ]; then

rm -rf ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile

echo "Remove File: ${WORK\_DIR}/${PROJECT\_NAME}/Dockerfile"

fi

cp /root/.jenkins/workspace/eshop-datalink-service/Dockerfile ${WORK\_DIR}/${PROJECT\_NAME}/

cp /root/.jenkins/workspace/eshop-datalink-service/target/\*.jar ${WORK\_DIR}/${PROJECT\_NAME}/

cd ${WORK\_DIR}/${PROJECT\_NAME}/

docker build -t ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION} .

docker push ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION}

if docker ps -a | grep ${PROJECT\_NAME}; then

docker rm -f ${PROJECT\_NAME}

echo "Remove Docker Container: ${PROJECT\_NAME}"

fi

docker run -d -p 8767:8767 --network="host" --name ${PROJECT\_NAME} ${REGISTRY\_URL}/eshop-detail/${PROJECT\_NAME}:${PROJECT\_VERSION}

6、执行一些修改

（1）修改1：在系统配置中设置maven，然后在配置中选择自己配置的maven版本，解决cannot run mvn program的错误

（2）修改2：编写DockerFile

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD eureka-server-0.0.1-SNAPSHOT.jar app.jar

#RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

EXPOSE 8761

hystrix有熔断的作用，所以一开始调用失败后，后面几次可能经常会失败，所以大家可以跟我一样，在datasync-service里加一个测试的接口，手动去调用一下product-service的接口，这样让接口调通了以后，就ok了

7、执行构建

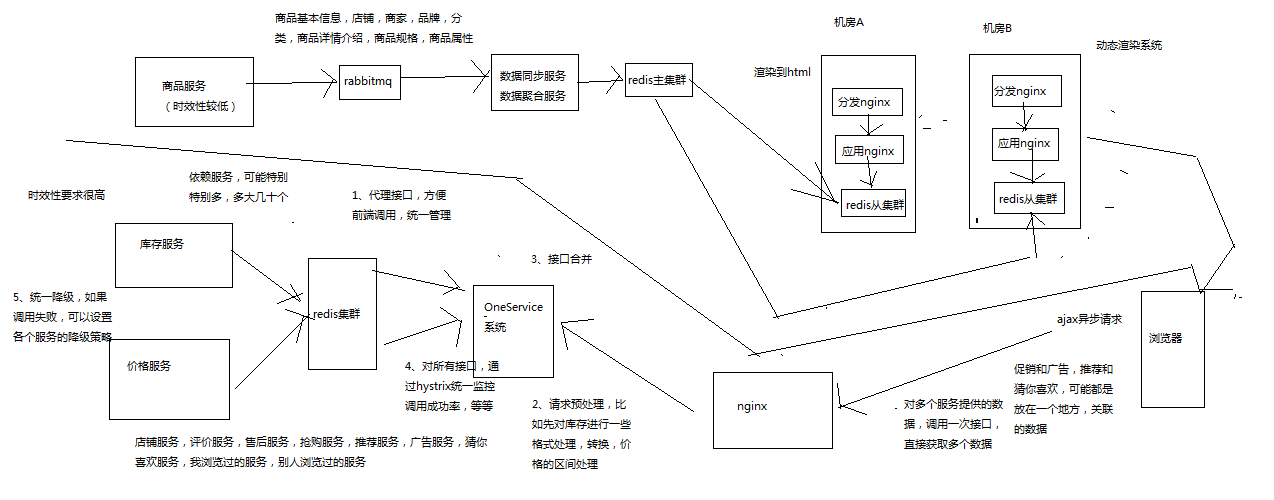
8、访问http://192.168.31.253:8761/，可以看到eureka-server的页面

2310879776

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'root' WITH GRANT OPTION;

#### 186\_商品详情页OneService系统：整体架构设计

商品详情页整体架构图



OneService系统

商品详情页依赖的服务达到数十个，甚至是上百个

需要给一个统一的入口，打造服务闭环

请求预处理

合并接口调用，减少ajax异步加载次数

统一监控

统一降级

2310879776

#### 187\_商品详情页OneService系统：基于Spring Cloud构建OneService服务

2310879776

#### 188\_商品详情页OneService系统：库存服务与价格服务的代理接口开发

统一入口，方便用户调用

2310879776

#### 189\_商品详情页OneService系统：请求预处理功能设计介绍

我这么跟大家说，我们是不可能完全真实的开发出一个完整业务的OneService系统的

因为那样做的话，需要构建出来大量的各种各样的依赖服务，才能将这个业务场景模拟出来，没必要

我这边，把一个基本的OenService系统的架子，基于spring cloud做出来，大家就能体验到一个完整的架构是怎么做的

至于里面一些具体的业务，就由我来给大家说明和介绍一下就可以了

请求预处理

比如库存状态（有货/无货）的转换，第三方运费的处理，第三方配送时效（多少天发货）的处理

处理主商品与配件关系，比如说iphone可能就搭载着耳机，充电器，等等

商家运费动态计算

请求预处理，先做一些简单的，薄薄的一层的封装和代理，先做点业务逻辑的判断和处理

这样的话，就可以给后端的服务传递更多的参数，或者简化后端服务的计算逻辑

2310879776

#### 190\_商品详情页OneService系统：多服务接口合并设计介绍

之前我们都跟大家说过了，可能页面中需要相关联的几份数据，就不用一次又一次的发送ajax请求来获取多份数据

直接就是一次请求发给一个one service系统的大接口，然后那个接口统一调用各个后端服务的接口就可以

减少浏览器和后端系统之间的交互次数

将哪些接口合并为一个接口呢？如何来设计接口的合并呢？

请求合并

促销和广告，合并成一个接口，一次性发送请求过来，然后调用促销服务和广告服务，获取两份数据

库存服务，配送服务，合并成一个接口，一次性过来，获取当前有多少库存，如何配送

组合推荐服务+配件推荐服务+套装推荐服务，三个服务并发调用，合并结果

2310879776

#### 191\_商品详情页OneService系统：基于hystrix进行接口统一降级

1、让greeting-service支持hystrix

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

（2）application.xml

feign.hystrix.enabled=true

（3）SayHelloService

@FeignClient(value = "say-hello-service", fallback = SayHelloServiceFallback.class)

public interface SayHelloService {

@RequestMapping(value = "/sayHello", method = RequestMethod.GET)

public String sayHello(@RequestParam(value = "name") String name);

}

（4）SayHelloServiceFallback

@Component

public class SayHelloServiceFallback implements SayHelloService {

@Override

public String sayHello(String name) {

return "error, " + name;

}

}

（5）先保持SayHelloService启动，可以访问; 关闭SayHelloSerivce，再访问，调用失败直接走降级

包括限流，自动熔断，调用失败，都会走降级

2310879776

#### 192\_商品详情页OneService系统：基于hystrix dashboard进行统一监控

2、hystrix dashboard

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

（2）Application

@EnableHystrixDashboard

@EnableCircuitBreaker

（3）http://localhost:8764/hystrix

输入http://localhost:8764/hystrix.stream和title

访问接口，会在hystrix dashboard看到访问请求

3、改造say-hello-service支持hystrix

将一个服务多个实例的指标聚合起来看，改造say-hello-service

（1）pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

（2）Application

@SpringBootApplication

@EnableEurekaClient

@RestController

@EnableHystrix

@EnableHystrixDashboard

@EnableCircuitBreaker

public class SayHelloServiceApplication {

public static void main(String[] args) {

SpringApplication.run(SayHelloServiceApplication.class, args);

}

@Value("${server.port}")

private String port;

@RequestMapping("/sayHello")

@HystrixCommand(fallbackMethod = "sayHelloFallback")

public String sayHello(String name) {

return "hello, " + name + " from port: " + port;

}

public String sayHelloFallback(String name) {

return "error, " + name

}

}

（3）locahost:8762/hystrix

输入locahost:8762/hystrix.stream，2000，title

访问这个接口

4、创建turbin工程，hystrix-turbine-server

（1）pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

（2）Application

@SpringBootApplication

@EnableTurbine

public class HystrixTurbineServer {

public static void main(String[] args) {

new SpringApplicationBuilder(HystrixTurbineServer.class).web(true).run(args);

}

}

（3）application.yml

spring:

application.name: hystrix-terbine-server

server:

port: 8765

security.basic.enabled: false

turbine:

aggregator:

clusterConfig: default

appConfig: say-hello-service

clusterNameExpression: new String("default")

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

（4）对say-hello-service每个服务实例都访问几次

http://localhost:8762/hystrix

stream输入：http://localhost:8765/turbine.stream

在dashboard可以看到两个服务实例聚合起来的指标

2310879776

#### 193\_商品详情页OneService系统：基于jenkins+docker部署OneService服务

2310879776

#### 194\_商品详情页OneService系统：基于jenkins+docker部署hystrix terbine服务

2310879776

#### 195\_商品详情页前端介绍&课程总结&Java架构师展望

咱们的最后一讲了

我们最后，相当于我们已经有了两套系统

第一套：商品服务+动态渲染系统

第二套：库存/价格服务+OneService系统

第三部分：前端页面

（1）时效性比较低的数据

更新的时候发送消息到mq，专门有一套数据同步服务+数据聚合服务来进行数据的加工和处理

前端页面，请求商品详情页的时候，nginx会走多级缓存策略（nginx local cache -> 本机房redis集群 -> 数据直连服务 -> 本地jvm cache -> redis主集群 -> 依赖服务），将时效性比较低的数据，全部加载到内存中，然后动态渲染到html中

前端html展示出来的时候，上来就有一些动态渲染出来的数据

（2）时效性比较高的数据

依赖服务每次更新数据库的时候，直接就更新redis缓存了，mysql+redis双写

前端html在展示出来以后，立即会对时效性要求比较高的数据，比如库存，价格，促销，推荐，广告，发送ajax请求到后盾

后端nginx接收到请求之后，就会将请求转发给one service系统，one service系统代理了所有几十个服务的接口，统一代理，统一降级，预处理，合并接口，统一监控

由one service系统发送请求给后端的一些服务，那些服务优先读redis，如果没有则读mysql，然后再重新刷入redis

（3）商品介绍

写的时候，采取的是分段存储策略，之前介绍过了

读的时候，也是在用户滚屏的时候，动态的异步ajax加载，分段加载商品介绍，不要一次性将所有的商品介绍都加载出来

总结

第一版：深入redis，缓存架构，hystrix高可用

第二版：完整的亿级流量商品详情页的系统架构，spring cloud+jenkins+docker的微服务项目实战

单课，是不可能真的将所有东西讲的面面俱到的

职业生涯去考虑，比如说从月薪十几k的中级工程师到月薪二十多k的高级工程师，或者从高级工程师到月薪三十多k，四十多k的架构师

至少1~2年的时间，非常系统的去学习完整的架构师的知识，才能够达到

我的这个架构班，内容一定是非常多，1~1.5年

2310879776