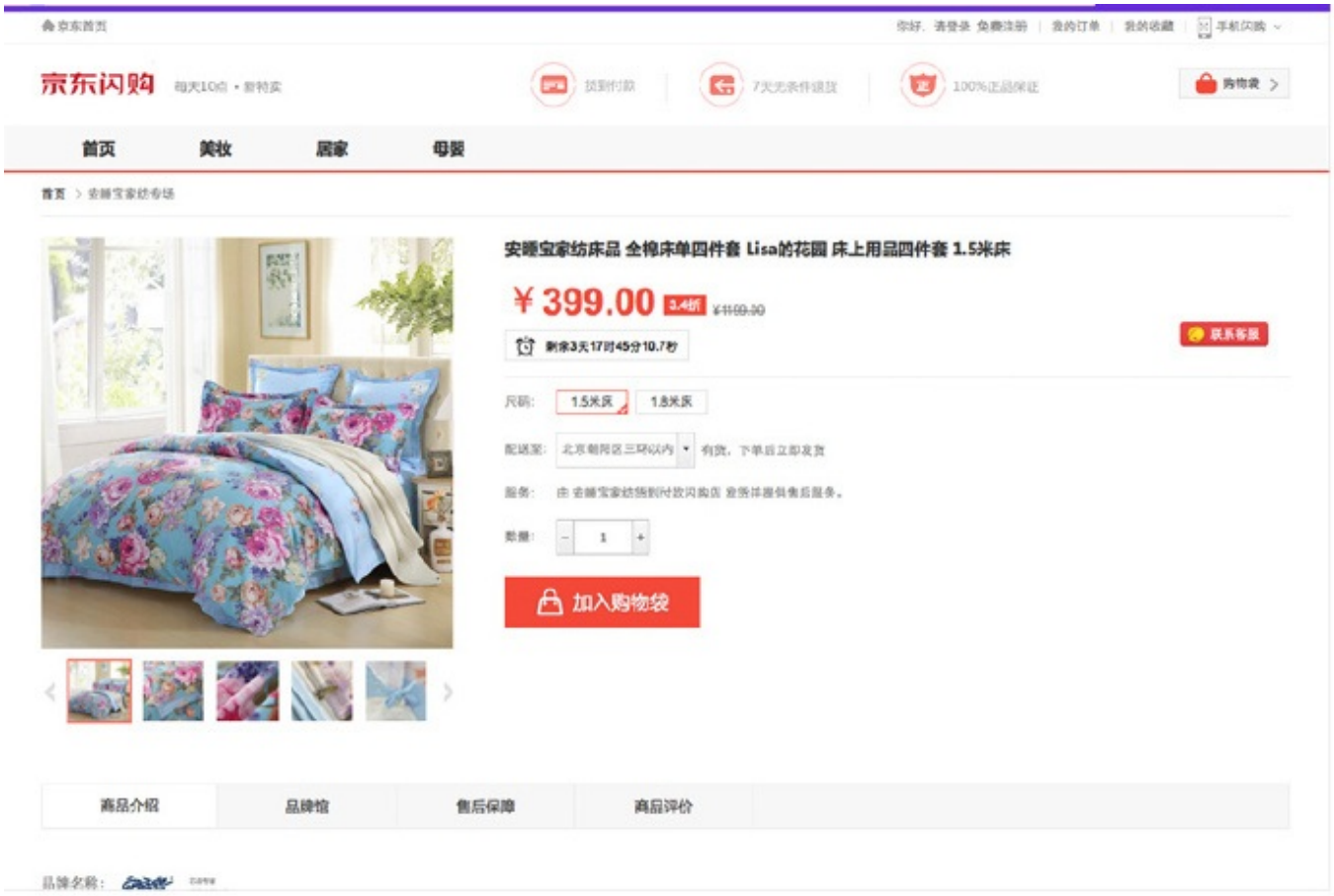
构建需求响应式亿级商品详情页

商品详情页是什么

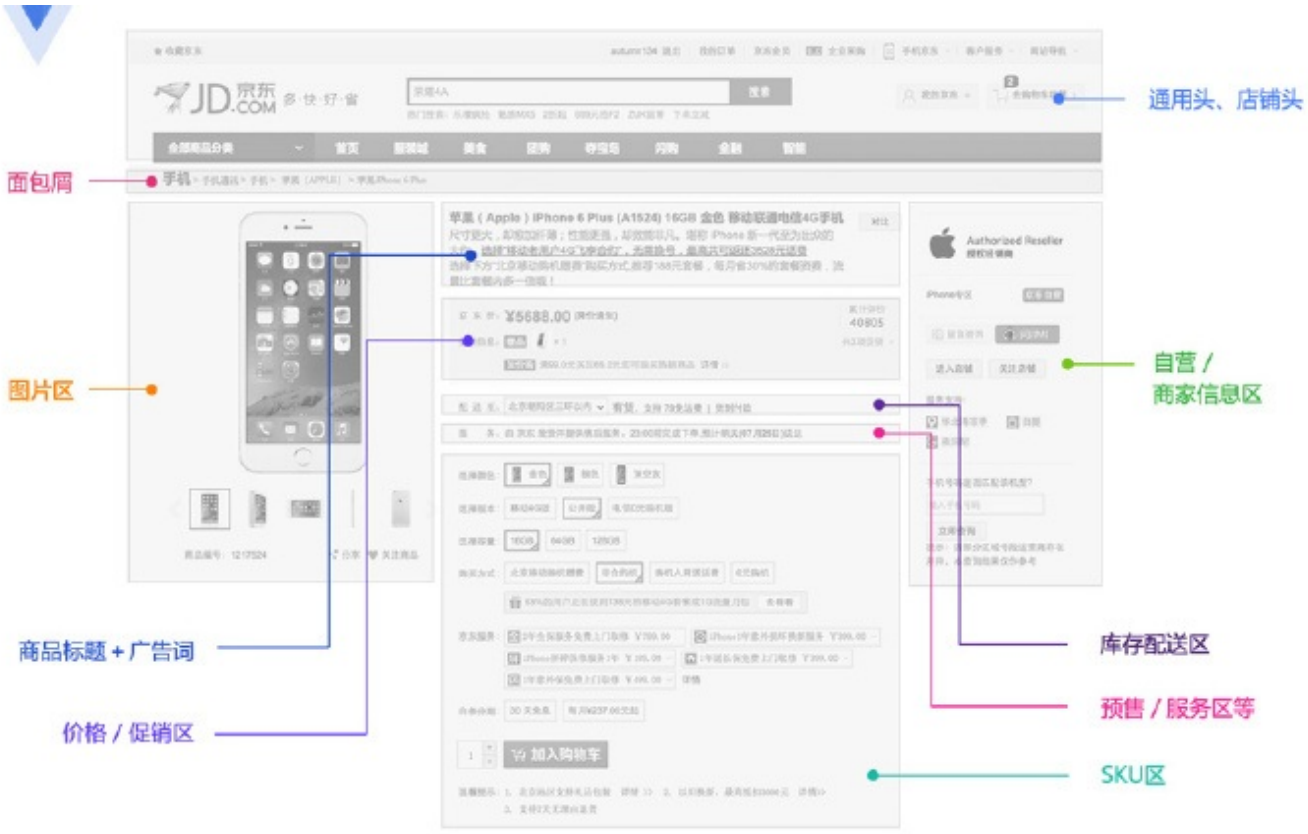
商品详情页是展示商品详细信息的一个页面，承载在网站的大部分流量和订单的入口。京东商城目前有通用版、全球购、闪购、易车、惠买车、服装、拼购、今日抄底等许多套模板。各套模板的元数据是一样的，只是展示方式不一样。目前商品详情页个性化需求非常多，数据来源也是非常多的，而且许多基础服务做不了的都放我们这，因此我们需要一种架构能快速响应和优雅的解决这些需求问题。因此我们重新设计了商品详情页的架构，主要包括三部分：商品详情页系统、商品详情页统一服务系统和商品详情页动态服务系统；商品详情页系统负责静的部分，而统一服务负责动的部分，而动态服务负责给内网其他系统提供一些数据服务。

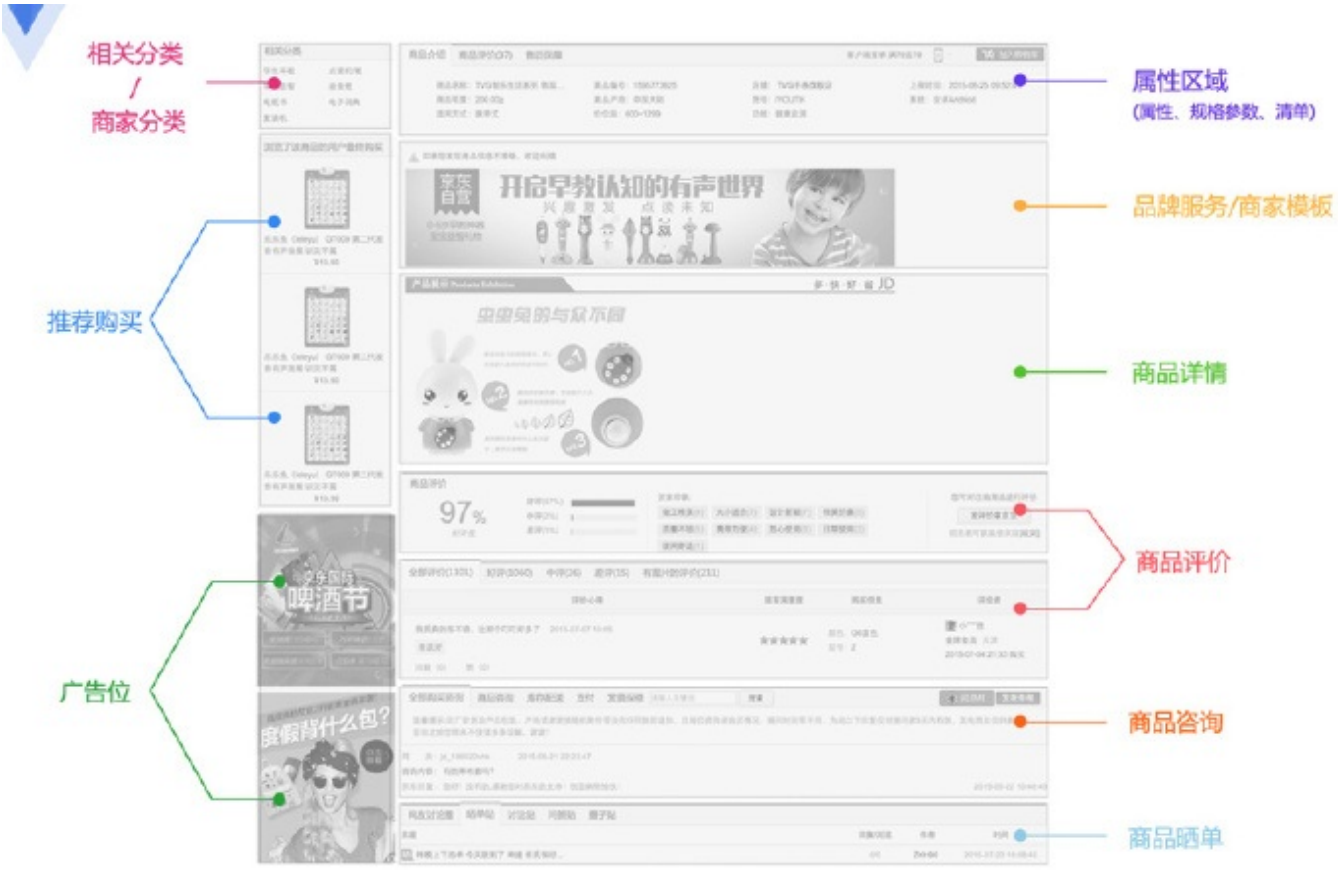


商品详情页前端结构

前端展示可以分为这么几个维度：商品维度(标题、图片、属性等)、主商品维度（商品介绍、规格参数）、分类维度、商家维度、店铺维度等；另外还有一些实时性要求比较高的如实时价格

、实时促销、广告词、配送至、预售等是通过异步加载。



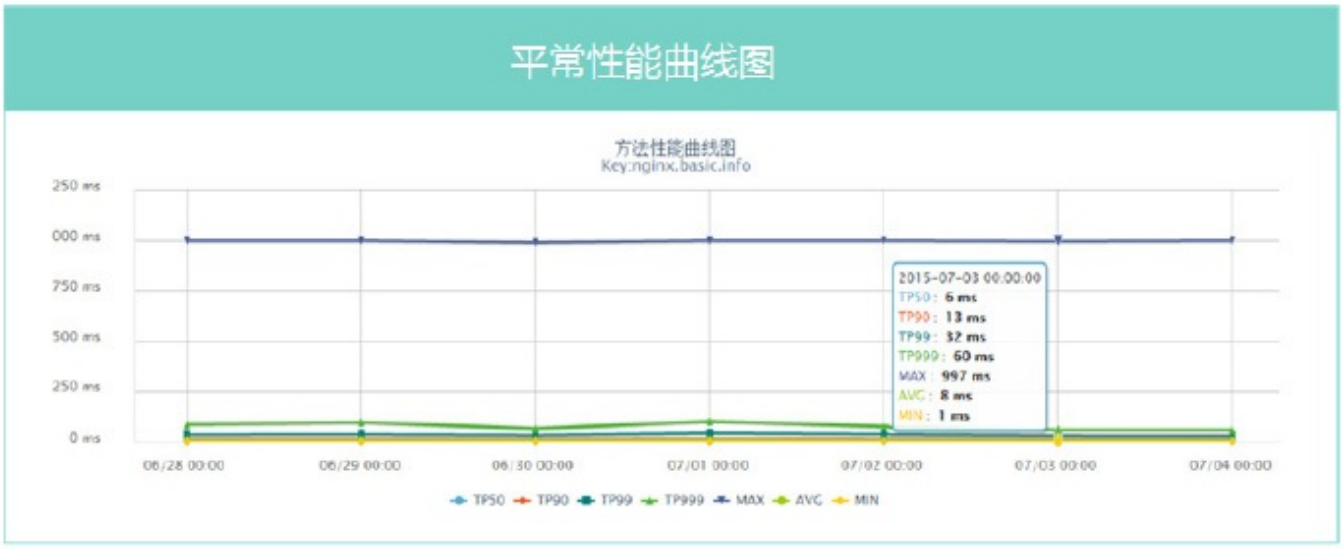


京东商城还有一些特殊维度数据：比如套装、手机合约机等，这些数据是主商品数据外挂的。

我们的性能数据

618当天PV数亿，618当天服务器端响应时间<38ms。此处我们用的是第1000次中第99次排

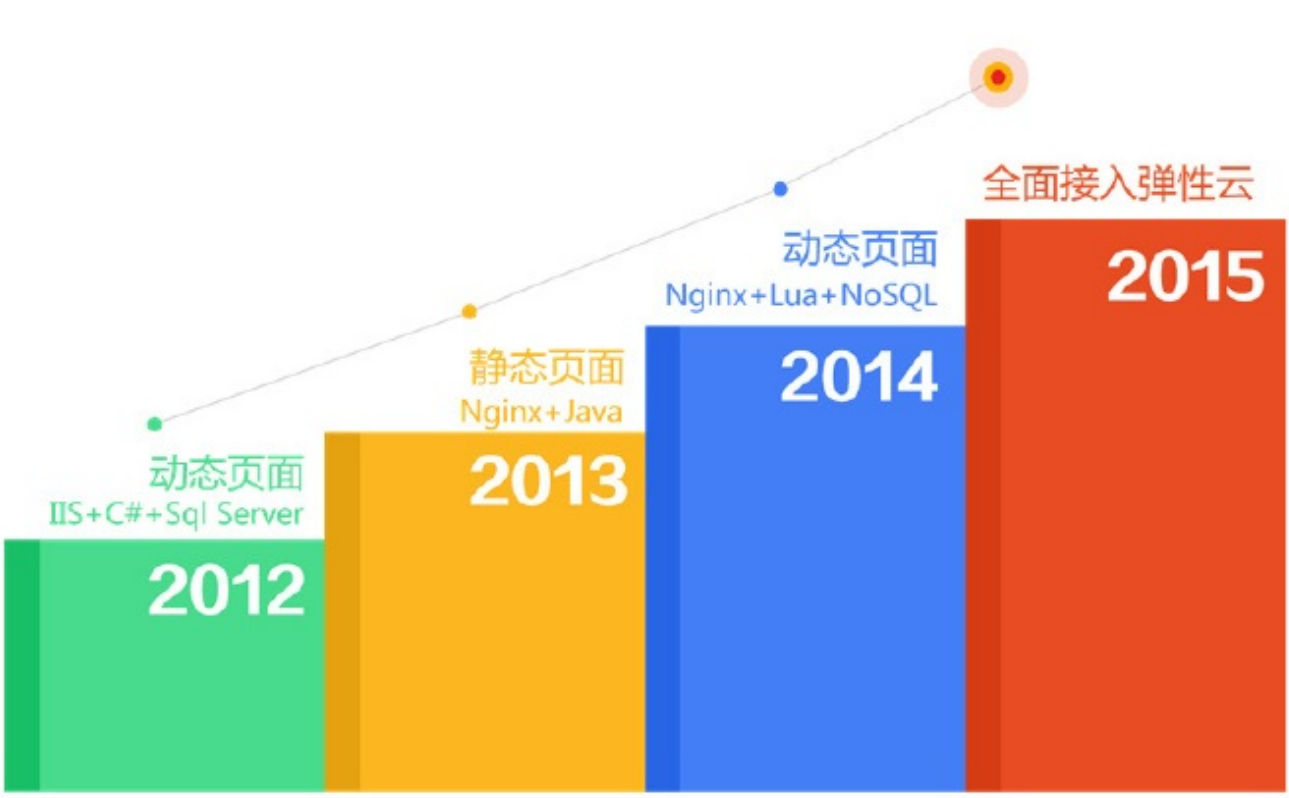
名的时间。



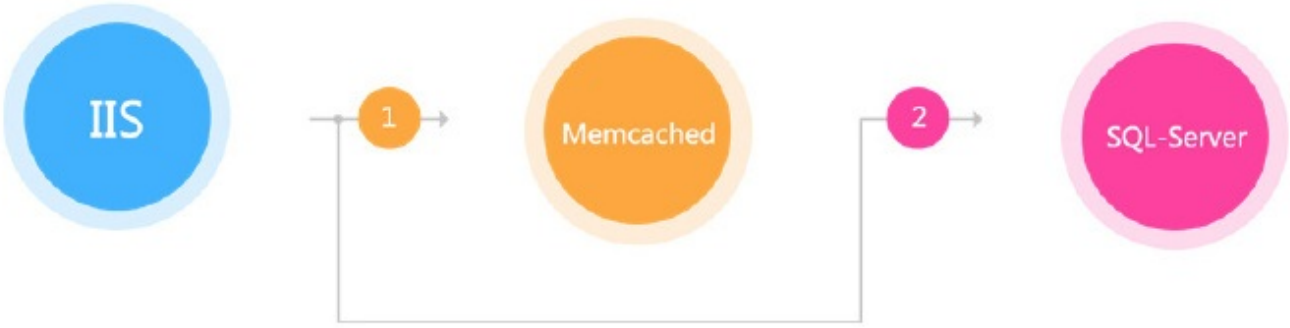
单品页流量特点

离散数据，热点少，各种爬虫、比价软件抓取。

单品页技术架构发展

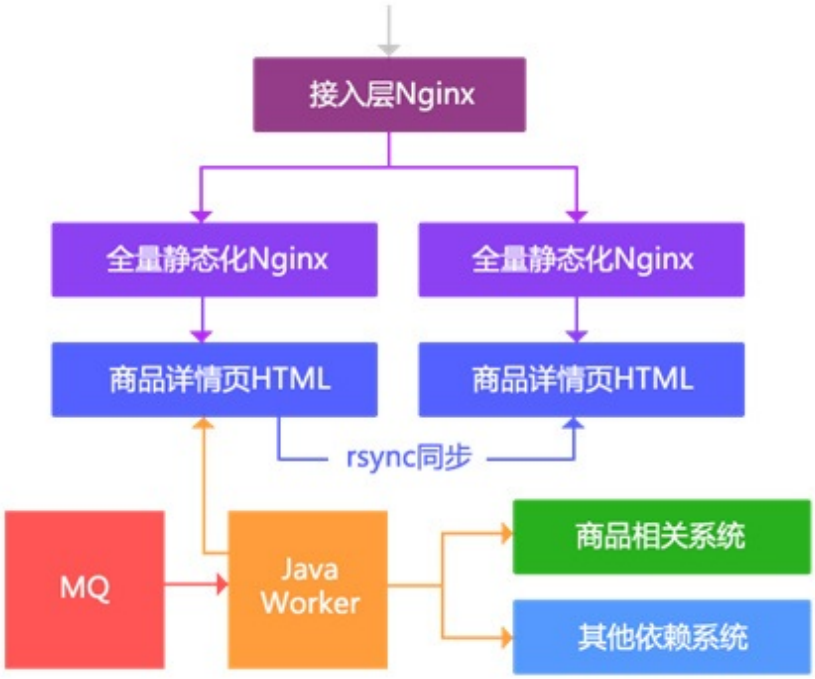


架构**1.0**



IIS+C#+Sql Server，最原始的架构，直接调用商品库获取相应的数据，扛不住时加了一层memcached来缓存数据。这种方式经常受到依赖的服务不稳定而导致的性能抖动。

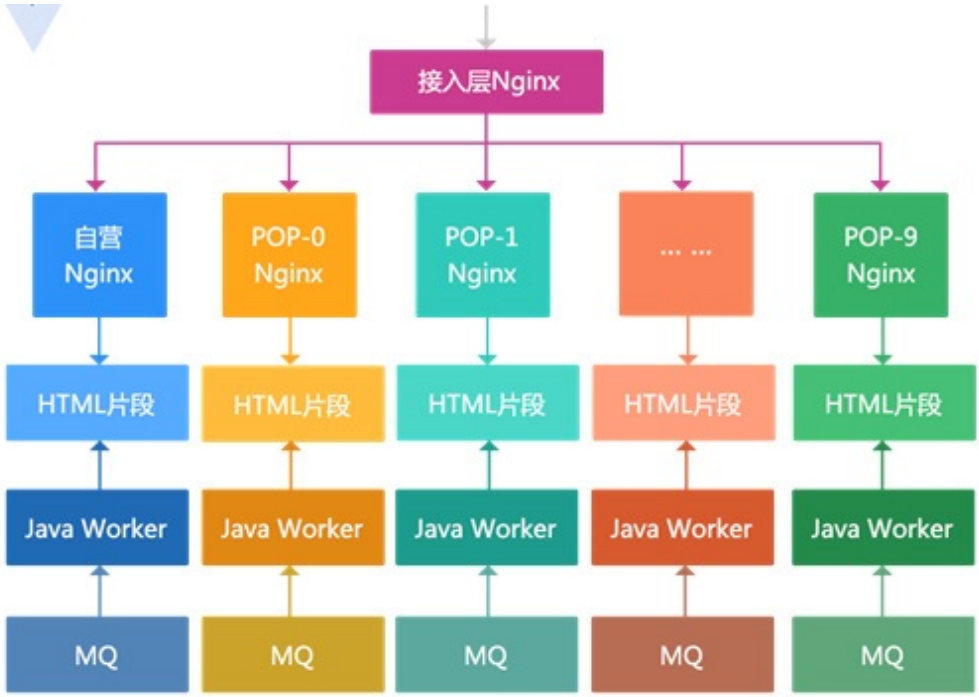
架构**2.0**



该方案使用了静态化技术，按照商品维度生成静态化HTML。主要思路： 1、通过MQ得到变更通知； 2、通过Java Worker调用多个依赖系统生成详情页HTML； 3、通过rsync同步到其他机器； 4、通过Nginx直接输出静态页； 5、接入层负责负载均衡。

该方案的主要缺点： 1、假设只有分类、面包屑变更了，那么所有相关的商品都要重刷； 2、随着商品数量的增加，rsync会成为瓶颈； 3、无法迅速响应一些页面需求变更，大部分都是通过JavaScript动态改页面元素。

随着商品数量的增加这种架构的存储容量到达了瓶颈，而且按照商品维度生成整个页面会存在如分类维度变更就要全部刷一遍这个分类下所有信息的问题，因此我们又改造了一版按照尾号路由到多台机器。



主要思路： 1、容量问题通过按照商品尾号做路由分散到多台机器，按照自营商品单独一台，第三方商品按照尾号分散到11台； 2、按维度生成HTML片段（框架、商品介绍、规格参数、面包屑、相关分类、店铺信息），而不是一个大HTML； 3、通过Nginx SSI合并片段输出； 4、接入层负责负载均衡； 5、多机房部署也无法通过rsync同步，而是使用部署多套相同的架构来实现。

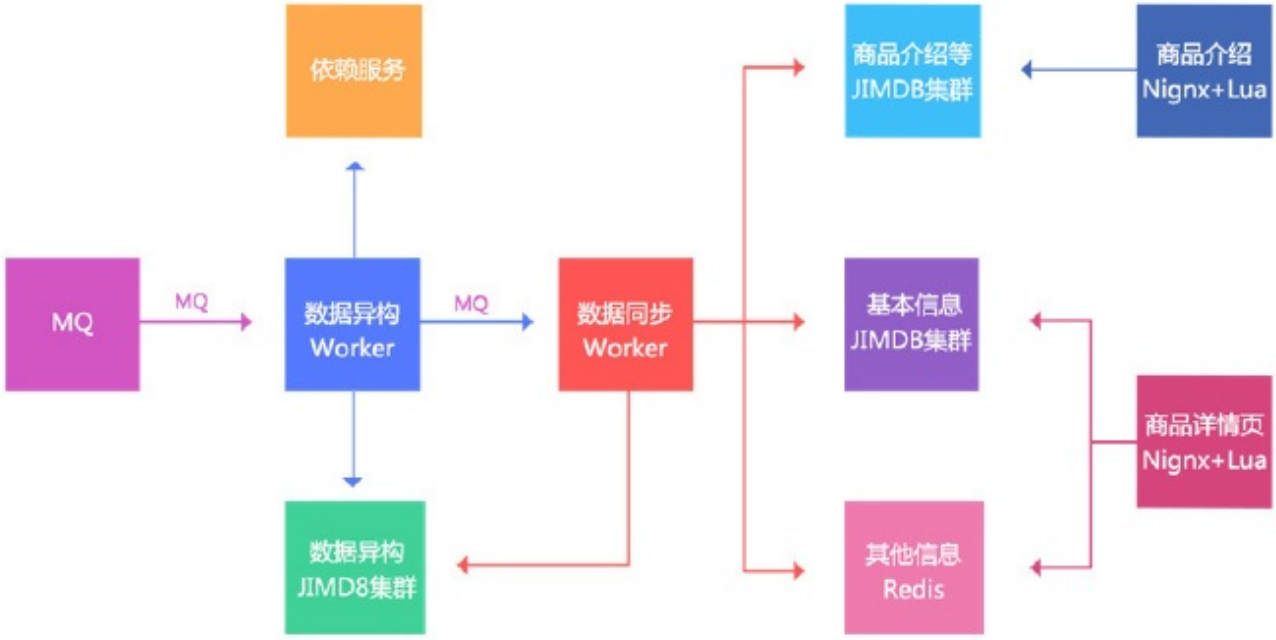
该方案主要缺点： 1、碎片文件太多，导致如无法rsync； 2、机械盘做SSI合并时，高并发时性能差，此时我们还没有尝试使用SSD； 3、模板如果要变更，数亿商品需要数天才能刷完； 4、到达容量瓶颈时，我们会删除一部分静态化商品，然后通过动态渲染输出，动态渲染系统在高峰时会导致依赖系统压力大，抗不住； 5、还是无法迅速响应一些业务需求。

我们的痛点

1、之前架构的问题存在容量问题，很快就会出现无法全量静态化，还是需要动态渲染；不过对于全量静态化可以通过分布式文件系统解决该问题，这种方案没有尝试； 2、最主要的问题是随着业务的发展，无法满足迅速变化、还有一些变态的需求。

架构**3.0**

我们要解决的问题： 1、能迅速响瞬变的需求，和各种变态需求； 2、支持各种垂直化页面改版； 3、页面模块化； 4、AB测试； 5、高性能、水平扩容； 6、多机房多活、异地多活。



主要思路： 1、数据变更还是通过MQ通知； 2、数据异构Worker得到通知，然后按照一些维度进行数据存储，存储到数据异构JIMDB集群（JIMDB：Redis+持久化引擎），存储的数据都是未加工的原子化数据，如商品基本信息、商品扩展属性、商品其他一些相关信息、商品规格参数、分类、商家信息等； 3、数据异构Worker存储成功后，会发送一个MQ给数据同步Worker，数据同步Worker也可以叫做数据聚合Worker，按照相应的维度聚合数据存储到相应的JIMDB集群；三个维度：基本信息（基本信息+扩展属性等的一个聚合）、商品介绍（PC版、移动版）、其他信息（分类、商家等维度，数据量小，直接Redis存储）； 4、前端展示分为两个：商品详情页和商品介绍，使用Nginx+Lua技术获取数据并渲染模板输出。

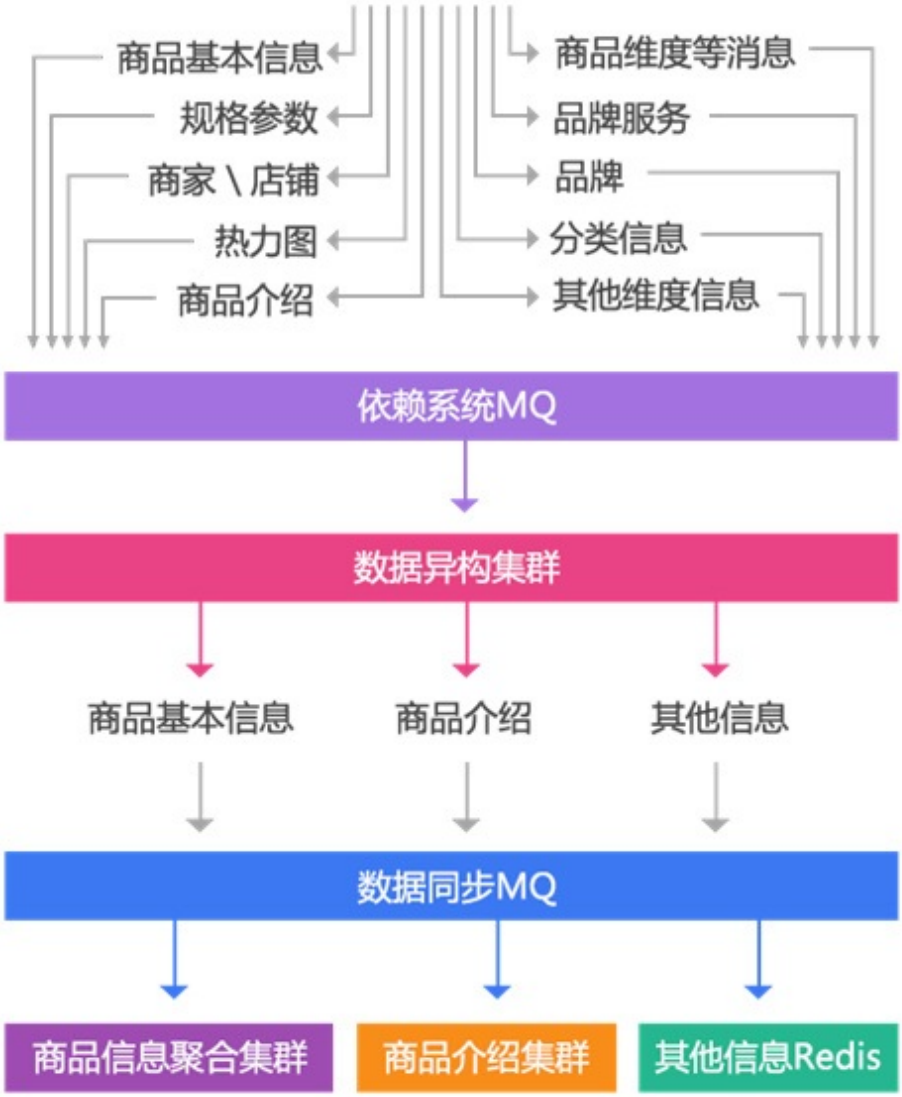
另外我们目前架构的目标不仅仅是为商品详情页提供数据，只要是Key-Value获取的而非关系的我们都可以提供服务，我们叫做动态服务系统。

该动态服务分为前端和后端，即公网还是内网，如目前该动态服务为列表页、商品对比、微信单品页、总代等提供相应的数据来满足和支持其业务。

详情页架构设计原则

1、数据闭环 2、数据维度化 3、拆分系统 4、Worker无状态化+任务化 5、异步化+并发化 6、多级缓存化 7、动态化 8、弹性化 9、降级开关 10、多机房多活 11、多种压测方案

数据闭环



数据闭环即数据的自我管理，或者说是数据都在自己系统里维护，不依赖于任何其他系统，去依赖化；这样得到的好处就是别人抖动跟我没关系。

数据异构，是数据闭环的第一步，将各个依赖系统的数据拿过来，按照自己的要求存储起来；

数据原子化，数据异构的数据是原子化数据，这样未来我们可以对这些数据再加工再处理而响应变化的需求；

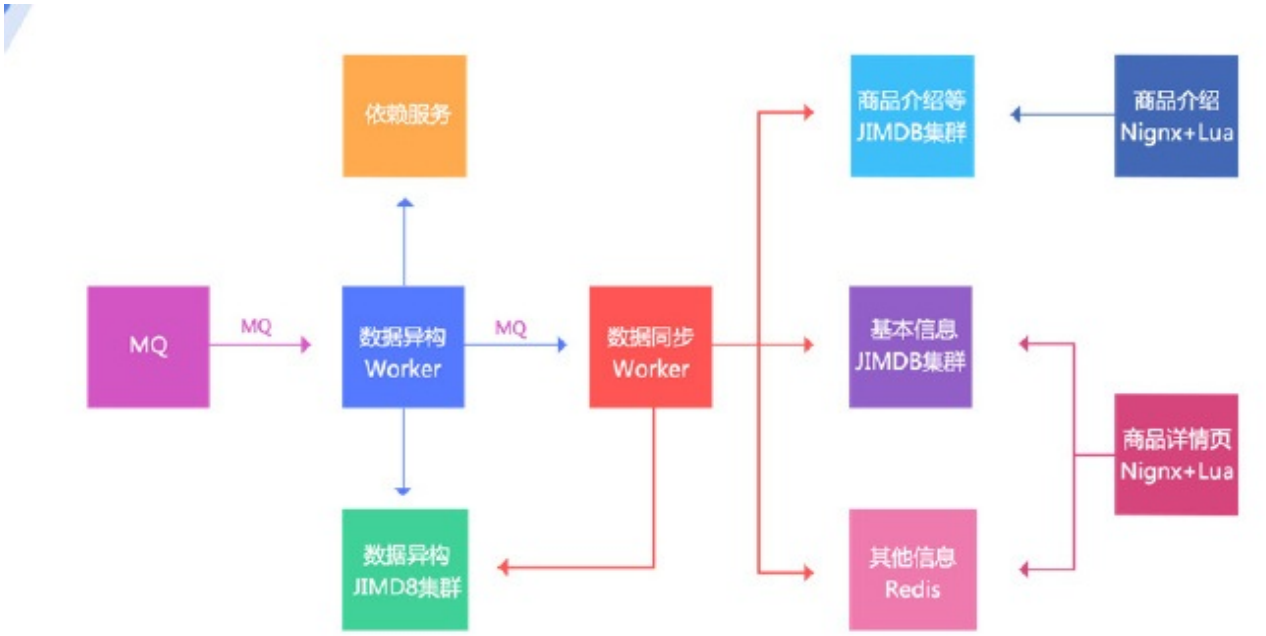
数据聚合，将多个原子数据聚合为一个大JSON数据，这样前端展示只需要一次get，当然要考虑系统架构，比如我们使用的Redis改造，Redis又是单线程系统，我们需要部署更多的Redis来支持更高的并发，另外存储的值要尽可能的小；

数据存储，我们使用JIMDB，Redis加持久化存储引擎，可以存储超过内存N倍的数据量，我们目前一些系统是Redis+LMDB引擎的存储，目前是配合SSD进行存储；另外我们使用Hash Tag机制把相关的数据哈希到同一个分片，这样mget时不需要跨分片合并。

我们目前的异构数据时键值结构的，用于按照商品维度查询，还有一套异构时关系结构的用于关系查询使用。

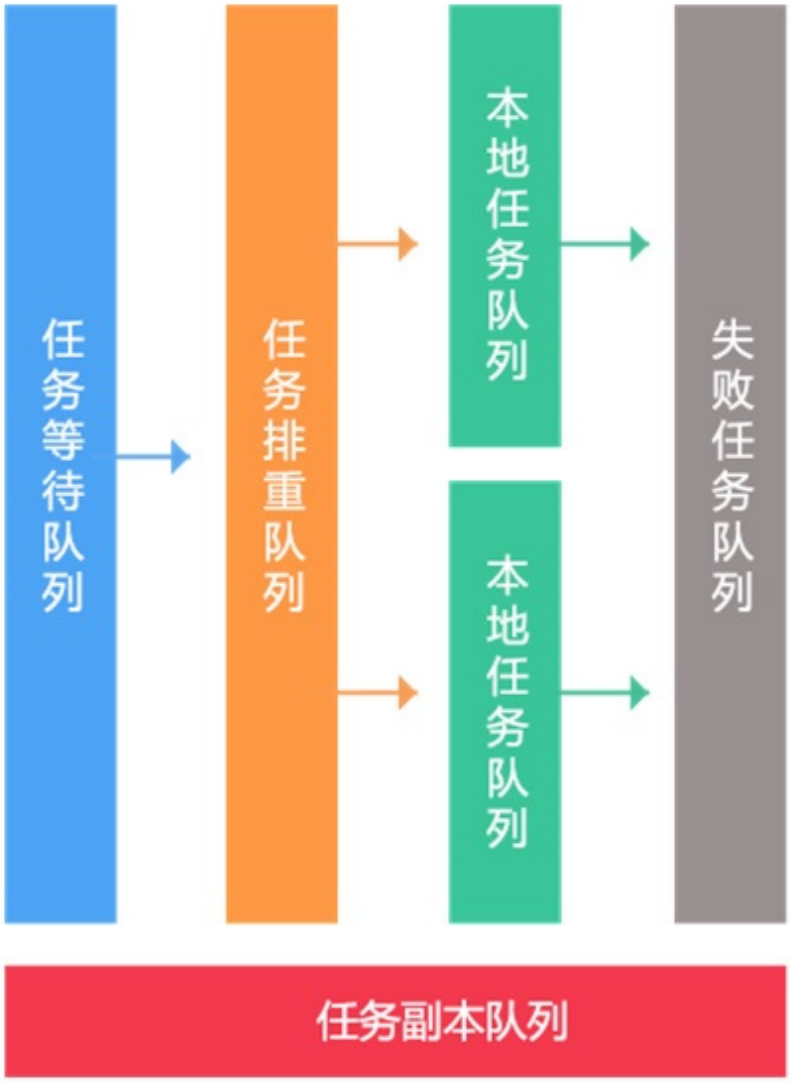
详情页架构设计原则 / 数据维度化 对于数据应该按照维度和作用进行维度化，这样可以分离存储，进行更有效的存储和使用。我们数据的维度比较简单： 1、商品基本信息，标题、扩展属性、特殊属性、图片、颜色尺码、规格参数等； 2、商品介绍信息，商品维度商家模板、商品介绍等； 3、非商品维度其他信息，分类信息、商家信息、店铺信息、店铺头、品牌信息等； 4、商品维度其他信息（异步加载），价格、促销、配送至、广告词、推荐配件、最佳组合等。

拆分系统



将系统拆分为多个子系统虽然增加了复杂性，但是可以得到更多的好处，比如数据异构系统存储的数据是原子化数据，这样可以按照一些维度对外提供服务；而数据同步系统存储的是聚合数据，可以为前端展示提供高性能的读取。而前端展示系统分离为商品详情页和商品介绍，可以减少相互影响；目前商品介绍系统还提供其他的一些服务，比如全站异步页脚服务。

Worker无状态化**+**任务化



1、数据异构和数据同步Worker无状态化设计，这样可以水平扩展； 2、应用虽然是无状态化的，但是配置文件还是有状态的，每个机房一套配置，这样每个机房只读取当前机房数据； 3、任务多队列化，等待队列、排重队列、本地执行队列、失败队列； 4、队列优先级化，分为：普通队列、刷数据队列、高优先级队列；例如一些秒杀商品会走高优先级队列保证快速执行； 5、副本队列，当上线后业务出现问题时，修正逻辑可以回放，从而修复数据；可以按照比如固定大小队列或者小时队列设计； 6、在设计消息时，按照维度更新，比如商品信息变更和商品上下架分离，减少每次变更接口的调用量，通过聚合Worker去做聚合。

异步化+并发化

我们系统大量使用异步化，通过异步化机制提升并发能力。首先我们使用了消息异步化 进行系统解耦合，通过消息通知我变更，然后我再调用相应接口获取相关数据；之前老系统使用同步推送机制，这种方式系统是紧耦合的，出问题需要联系各个负责人重新推送还要考虑失败重试机制。数据更新异步化 ，更新缓存时，同步调用服务，然后异步更新缓存。可并行任务并发化，商品数据系统来源有多处，但是可以并发调用聚合，这样本来串行需要1s的经过这种方式我们提升到300ms之内。异步请求合并，异步请求做合并，然后一次请求调用就能拿到所有数据。前端服务异步化/聚合，实时价格、实时库存异步化， 使用如线程或协程机制将多个可并发的服务聚合。异步化还一个好处就是可以对异步请求做合并，原来N次调用可以合并为一次，还可以做请求的

排重。

多级缓存化

浏览器缓存，当页面之间来回跳转时走local cache，或者打开页面时拿着Last-Modified去CDN验证是否过期，减少来回传输的数据量； CDN缓存，用户去离自己最近的CDN节点拿数据，而不是都回源到北京机房获取数据，提升访问性能； 服务端应用本地缓存，我们使用Nginx+Lua架构，使用HttpLuaModule模块的shared dict做本地缓存（ reload不丢失）或内存级Proxy Cache，从而减少带宽； 另外我们还使用使用一致性哈希（如商品编号/分类）做负载均衡内部对URL重写提升命中率； 我们对mget做了优化，如去商品其他维度数据，分类、面包屑、商家等差不多8个维度数据，如果每次mget获取性能差而且数据量很大，30KB以上；而这些数据缓存半小时也是没有问题的，因此我们设计为先读local cache，然后把不命中的再回源到remote cache获取，这个优化减少了一半以上的remote cache流量； 服务端分布式缓存，我们使用内存+SSD+JIMDB持久化存储。

动态化

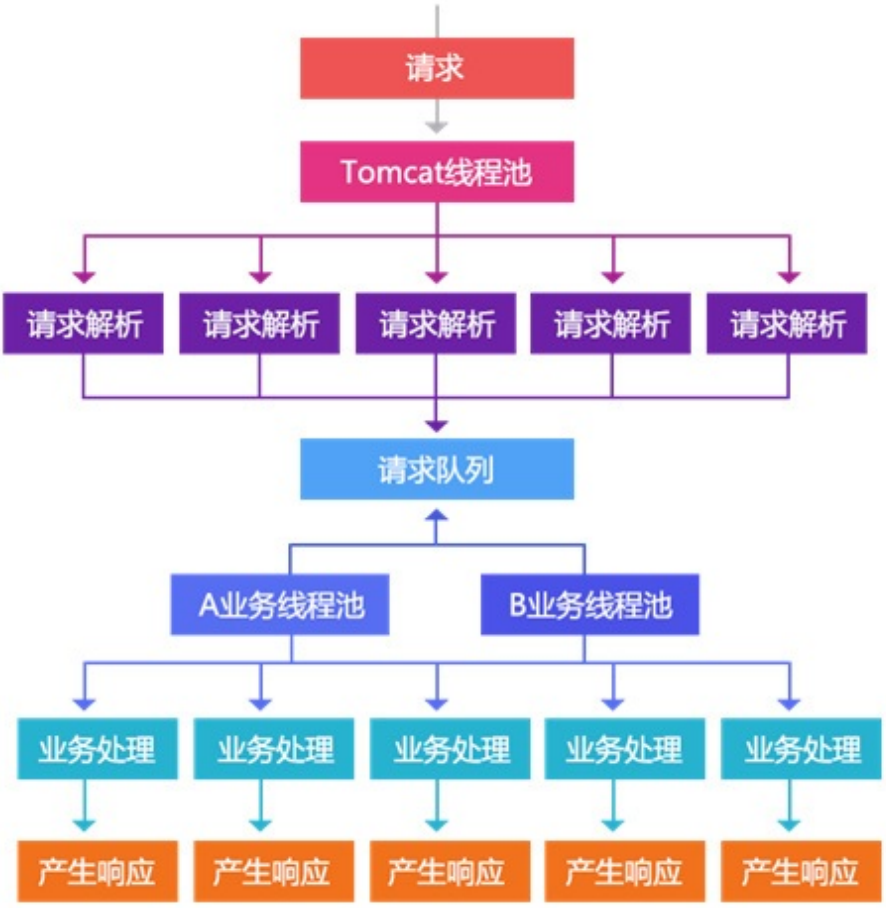
数据获取动态化，商品详情页：按维度获取数据，商品基本数据、其他数据（分类、商家信息等）；而且可以根据数据属性，按需做逻辑，比如虚拟商品需要自己定制的详情页，那么我们就可以跳转走，比如全球购的需要走jd.hk域名，那么也是没有问题的； 模板渲染实时化，支持随时变更模板需求； 重启应用秒级化，使用Nginx+Lua架构，重启速度快，重启不丢共享字典缓存数据； 需求上线速度化，因为我们使用了Nginx+Lua架构，可以快速上线和重启应用，不会产生抖动；另外Lua本身是一种脚本语言，我们也在尝试把代码如何版本化存储，直接内部驱动Lua代码更新上线而不需要重启Nginx。

弹性化

我们所有应用业务都接入了Docker容器，存储还是物理机；我们会制作一些基础镜像，把需要的软件打成镜像，这样不用每次去运维那安装部署软件了；未来可以支持自动扩容，比如按照CPU或带宽自动扩容机器，目前京东一些业务支持一分钟自动扩容。

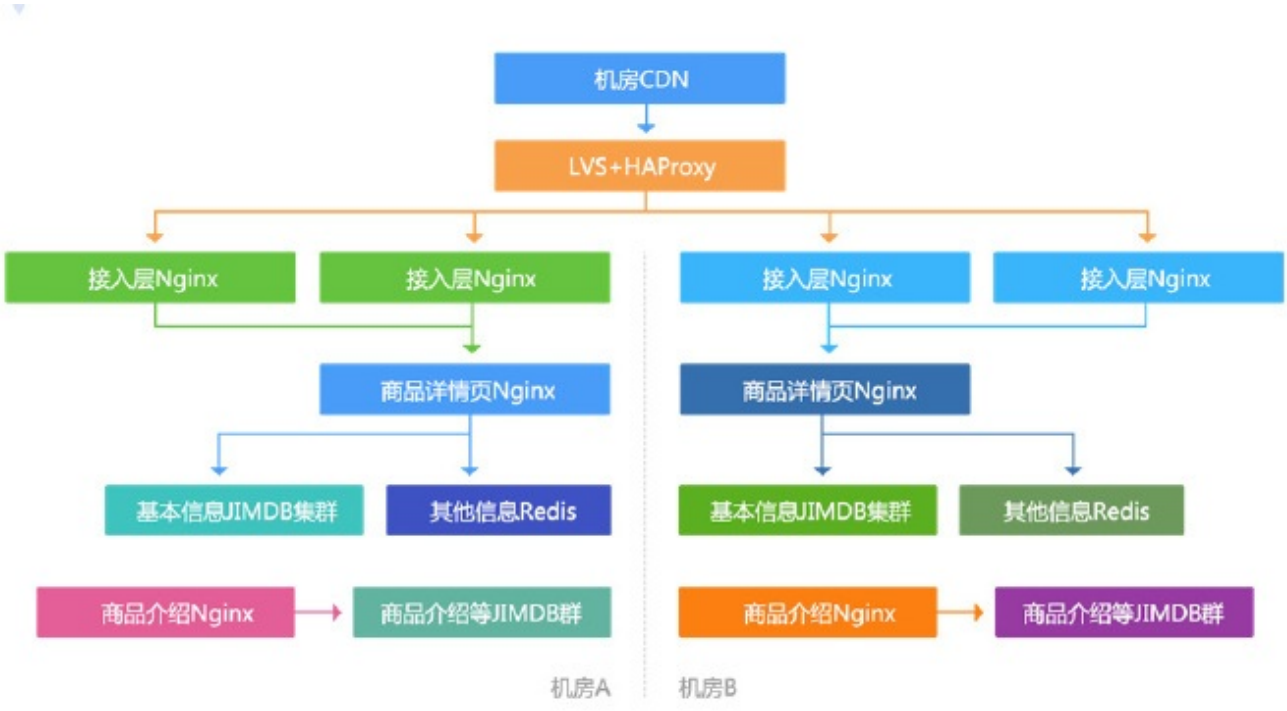
降级开关

推送服务器推送降级开关，开关集中化维护，然后通过推送机制推送到各个服务器； 可降级的多级读服务，前端数据集群—>数据异构集群—>动态服务(调用依赖系统)；这样可以保证服务质量，假设前端数据集群坏了一个 磁盘，还可以回源到数据异构集群获取数据； 开关前置化，如Nginx–àTomcat，在Nginx上做开关，请求就到不了后端，减少后端压力； 可降级的业务线程池隔离，从Servlet3开始支持异步模型，Tomcat7/Jetty8开始支持，相同的概念是Jetty6的Continuations。我们可以把处理过程分解为一个个的事件。通过这种将请求划分为事件方式我们可以进行更多的控制。如，我们可以为不同的业务再建立不同的线程池进行控制：即我们只依赖tomcat线程池进行请求的解析，对于请求的处理我们交给我们自己的线程池去完成；这样tomcat线程池就不是我们的瓶颈，造成现在无法优化的状况。通过使用这种异步化事件模型，我们可以提高整体的吞吐量，不让慢速的A业务处理影响到其他业务处理。慢的还是慢，但是不影响其他的业务。我们通过这种机制还可以把tomcat线程池的监控拿出来，出问题时可以直接清空业务线程池，另外还可以自定义任务队列来支持一些特殊的业务。

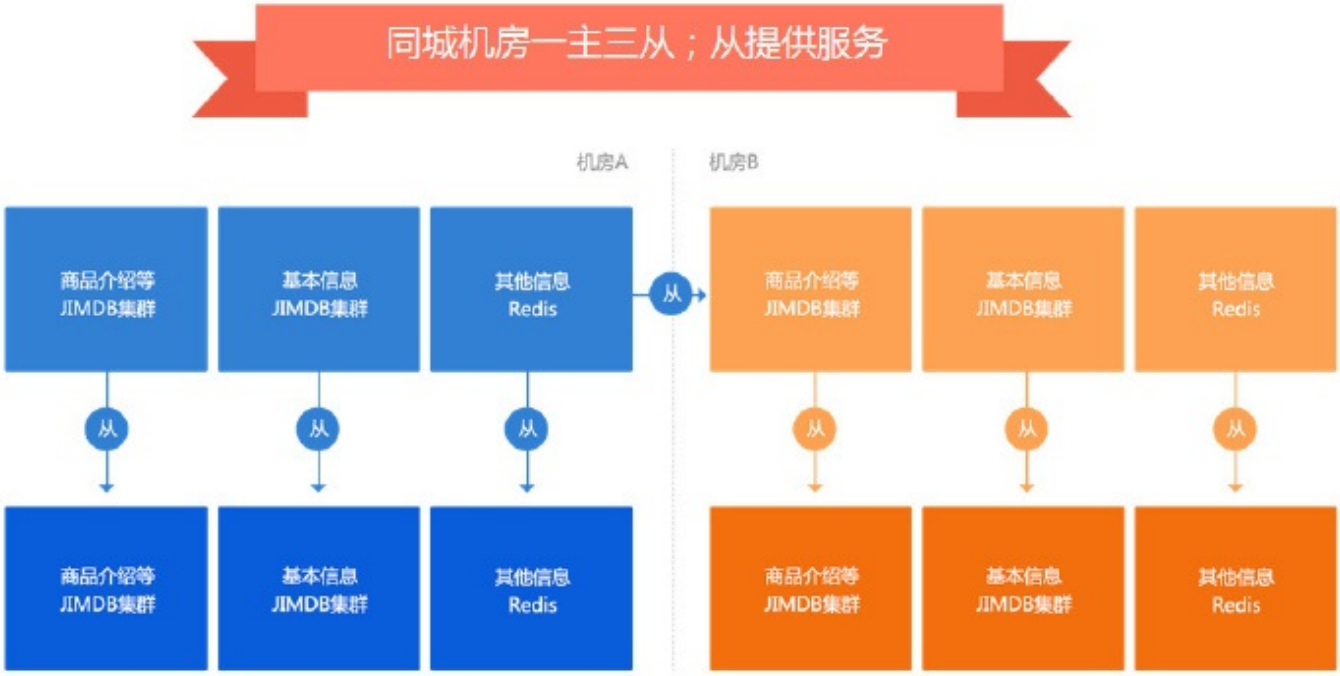


多机房多活

应用无状态，通过在配置文件中配置各自机房的数据集群来完成数据读取。



数据集群采用一主三从结构，防止当一个机房挂了，另一个机房压力大产生抖动。



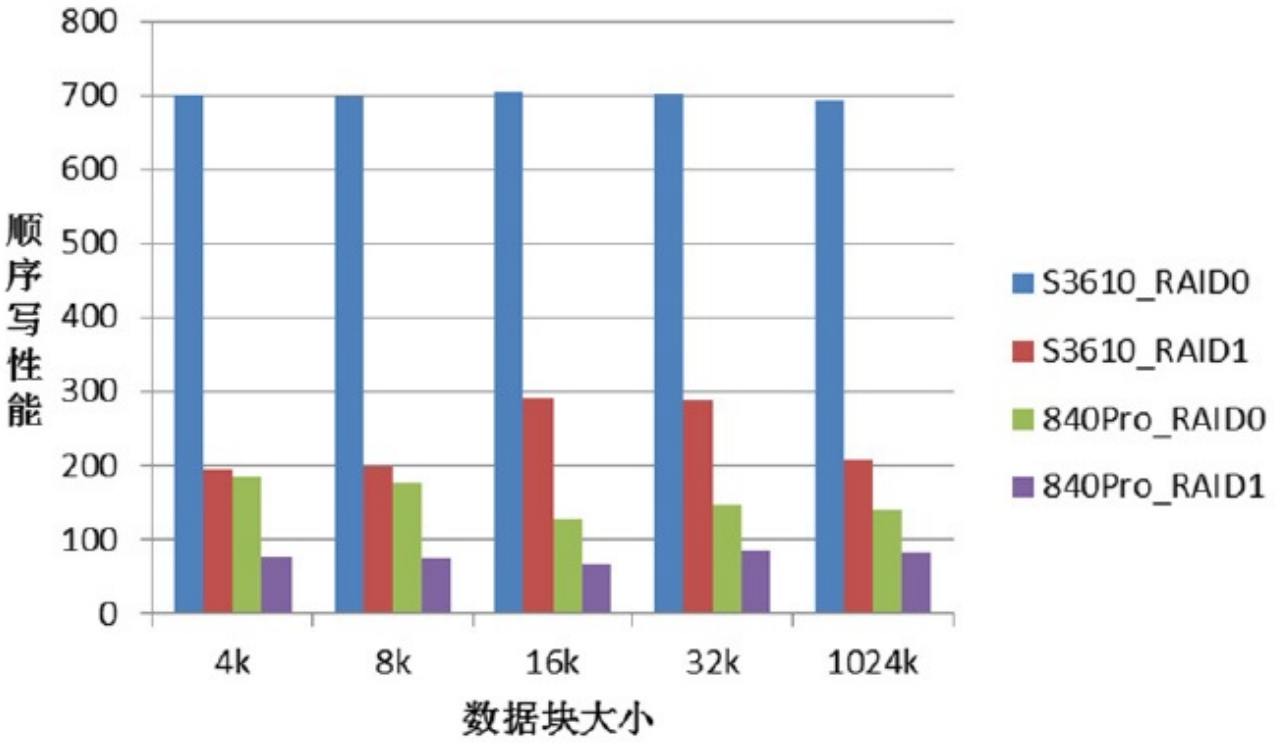
多种压测方案

线下压测，Apache ab，Apache Jmeter，这种方式是固定url压测，一般通过访问日志收集一些url进行压测，可以简单压测单机峰值吞吐量，但是不能作为最终的压测结果，因为这种压测会存在热点问题； 线上压测，可以使用Tcpcopy直接把线上流量导入到压测服务器，这种方式可以压测出机器的性能，而且可以把流量放大，也可以使用Nginx+Lua协程机制把流量分发到多台压测服务器，或者直接在页面埋点，让用户压测，此种压测方式可以不给用户返回内容。

遇到的一些坑和问题

SSD性能差

使用SSD做KV存储时发现磁盘IO非常低。配置成RAID10的性能只有3~6MB/s；配置成RAID0的性能有~130MB/s，系统中没有发现CPU，MEM，中断等瓶颈。一台服务器从RAID1改成RAID0后，性能只有~60MB/s。这说明我们用的SSD盘性能不稳定。 根据以上现象，初步怀疑以下几点：SSD盘，线上系统用的三星840Pro是消费级硬盘。RAID卡设置，Write back和Write through策略。后来测试验证，有影响，但不是关键。RAID卡类型，线上系统用的是LSI 2008，比较陈旧。

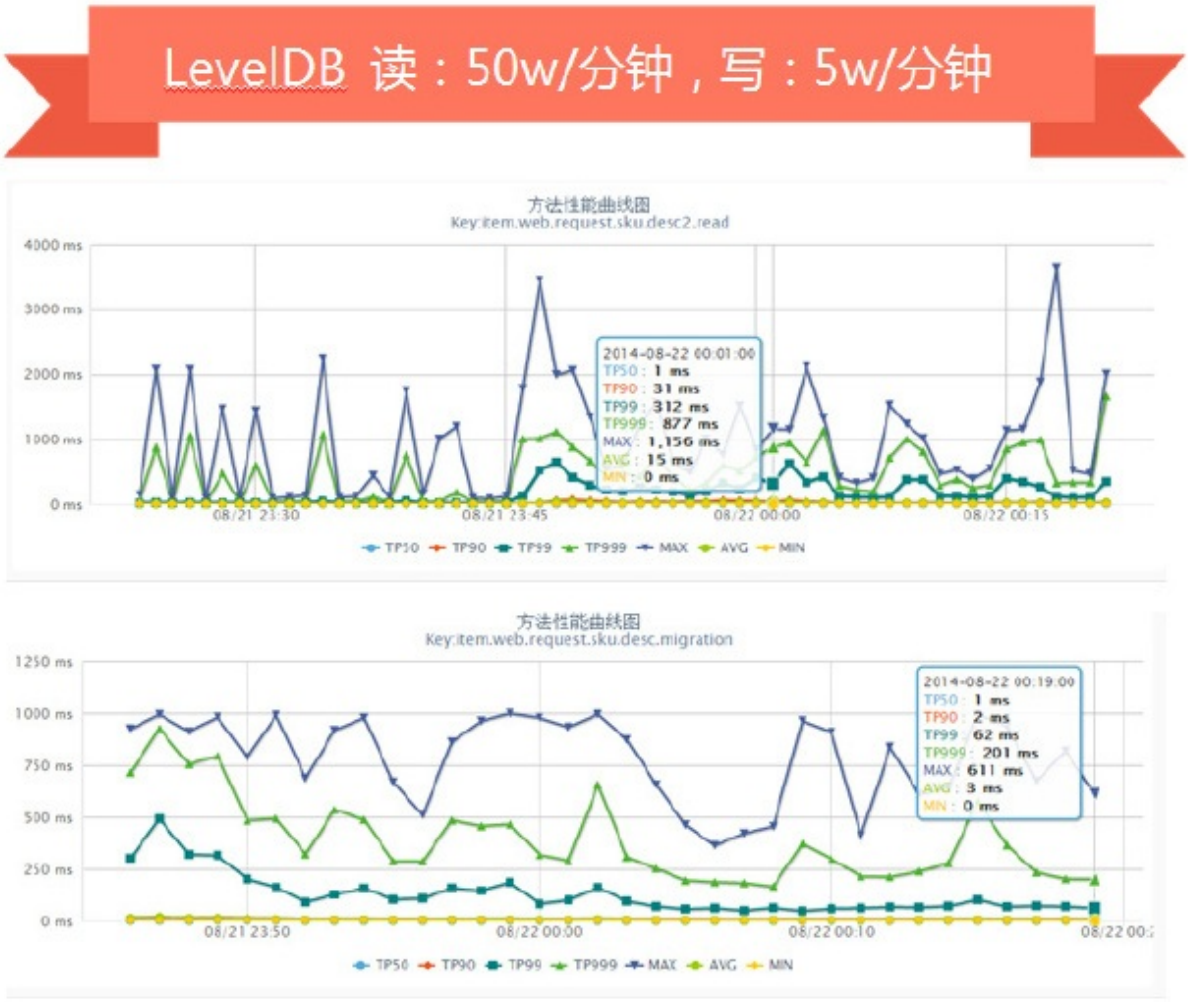


本实验使用dd顺序写操作简单测试，严格测试需要用FIO等工具。

键值存储选型压测

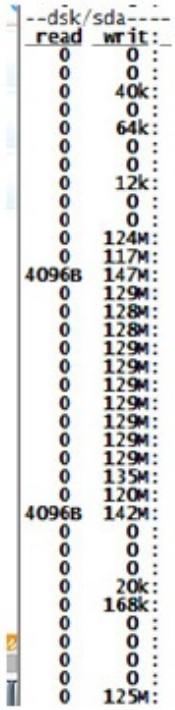
我们对于存储选型时尝试过LevelDB、RocksDB、BeansDB、LMDB、Riak等，最终根据我们的需求选择了LMDB。 机器：2台 配置：32核CPU、32GB内存、SSD（(512GB)三星840Pro –> (600GB)Intel 3500 /Intel S3610） 数据：1.7亿数据（800多G数据）、大小5~30KB左右 KV存储引擎：LevelDB、RocksDB、LMDB，每台启动2个实例 压测工具：tcpcopy直接线上导流 压测用例：随机写+随机读

LevelDB压测时，随机读+随机写会产生抖动（我们的数据出自自己的监控平台，分钟级采样）。

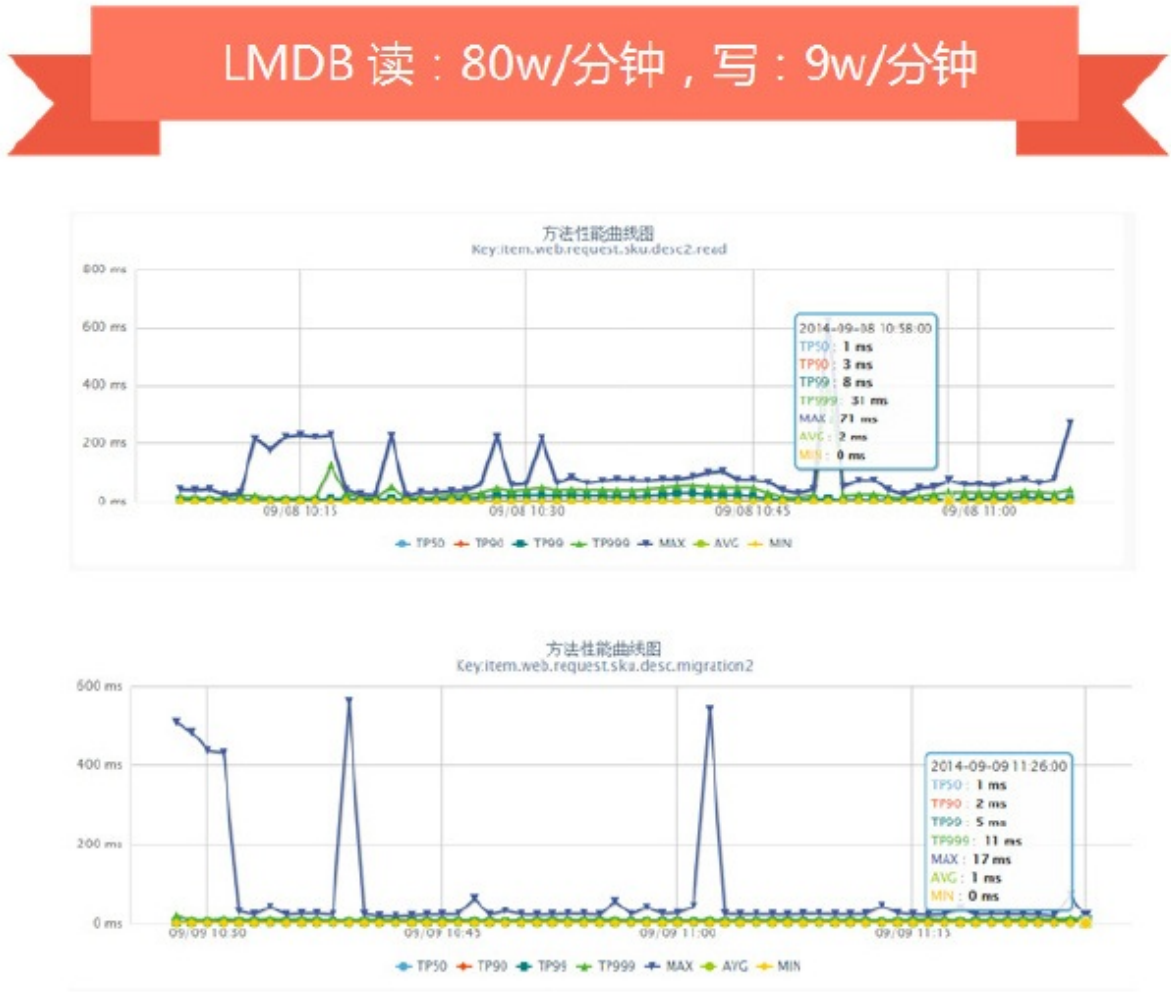


RocksDB是改造自LevelDB，对SSD做了优化，我们压测时单独写或读，性能非常好，但是读写混合时就会因为归并产生抖动。

RocksDB 读：80w/分钟，写：2.3w/分钟



LMDB引擎没有大的抖动，基本满足我们的需求。



我们目前一些线上服务器使用的是LMDB，其他一些正在尝试公司自主研发的CycleDB引擎。

数据量大时**JIMDB**同步不动

Jimdb数据同步时要dump数据，SSD盘容量用了50%以上，dump到同一块磁盘容量不足。解决方案： 1、一台物理机挂2块SSD(512GB)，单挂raid0；启动8个jimdb实例；这样每实例差不多125GB左右；目前是挂4块，raid0；新机房计划8块raid10； 2、目前是千兆网卡同步，同步峰值在100MB/s左右； 3、dump和sync数据时是顺序读写，因此挂一块SAS盘专门来同步数据； 4、使用文件锁保证一台物理机多个实例同时只有一个dump； 5、后续计划改造为直接内存转发而不做dump。

切换主从

之前存储架构是一主二从（主机房一主一从，备机房一从）切换到备机房时，只有一个主服务，读写压力大时有抖动，因此我们改造为之前架构图中的一主三从。

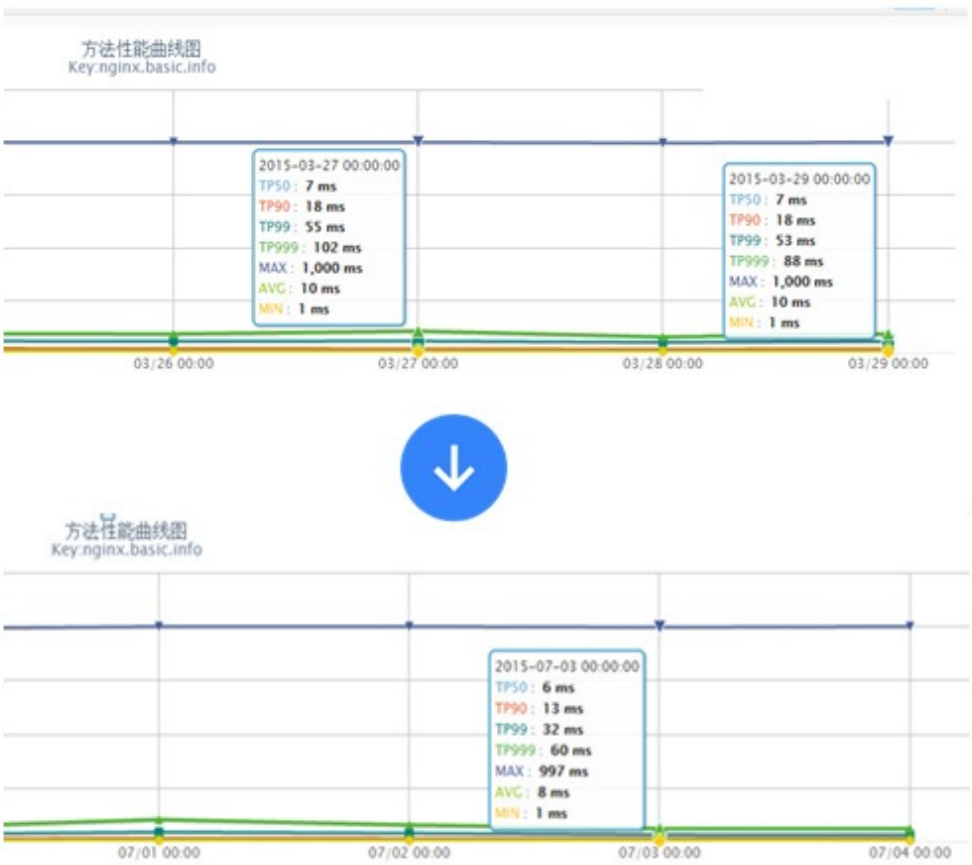
分片配置

之前的架构是分片逻辑分散到多个子系统的配置文件中，切换时需要操作很多系统；解决方案： 1、引入Twemproxy中间件，我们使用本地部署的Twemproxy来维护分片逻辑； 2、使用自动部署系统推送配置和重启应用，重启之前暂停mq消费保证数据一致性； 3、用unix domain socket减少连接数和端口占用不释放启动不了服务的问题。

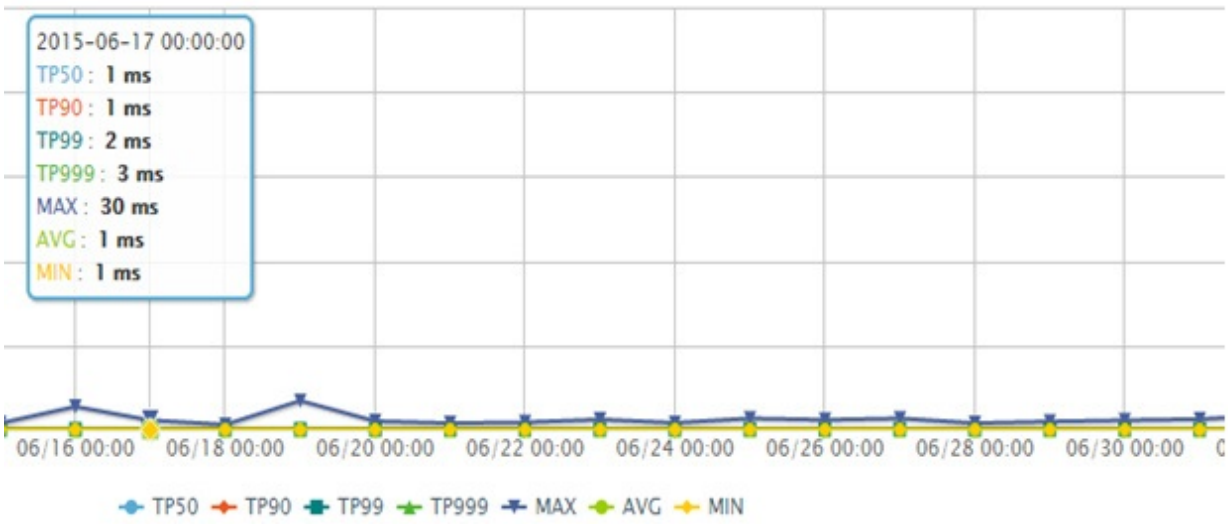
模板元数据存储**HTML**

起初不确定Lua做逻辑和渲染模板性能如何，就尽量减少for、if/else之类的逻辑；通过java worker组装html片段存储到jimdb，html片段会存储诸多问题，假设未来变了也是需要全量刷出的，因此存储的内容最好就是元数据。因此通过线上不断压测，最终jimdb只存储元数据，lua做逻辑和渲染；逻辑代码在3000行以上；模板代码1500行以上，其中大量for、if/else，目前渲染性能可以接受。

线上真实流量，整体性能从TP99 53ms降到32ms。



绑定8 CPU测试的，渲染模板的性能可以接受。



库存接口访问量**600w**/分钟

商品详情页库存接口2014年被恶意刷，每分钟超过600w访问量，tomcat机器只能定时重启；因为是详情页展示的数据，缓存几秒钟是可以接受的，因此开启nginx proxy cache来解决该问题，开启后降到正常水平；我们目前正在使用Nginx+Lua架构改造服务，数据过滤、URL重写等在Nginx层完成，通过URL重写+一致性哈希负载均衡，不怕随机URL，一些服务提升了10%+的缓存命中率。

微信接口调用量暴增

通过访问日志发现某IP频繁抓取；而且按照商品编号遍历，但是会有一些不存在的编号；解决方案： 1、读取KV存储的部分不限流； 2、回源到服务接口的进行请求限流，保证服务质量。

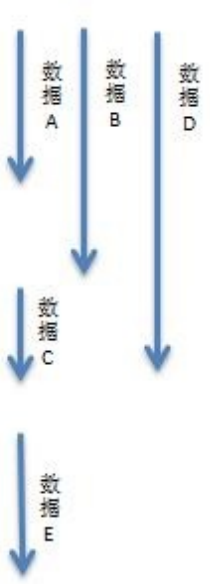
开启**Nginx Proxy Cache**性能不升反降

开启Nginx Proxy Cache后，性能下降，而且过一段内存使用率到达98%；解决方案： 1、对于内存占用率高的问题是内核问题，内核使用LRU机制，本身不是问题，不过可以通过修改内核参数 sysctl -w vm.extra_free_kbytes=6436787 sysctl -w vm.vfs_cache_pressure=10000 2、使用Proxy Cache在机械盘上性能差可以通过tmpfs缓存或nginx共享字典缓存元数据，或者使用SSD，我们目前使用内存文件系统。

配送至读服务因依赖太多，响应时间偏慢

配送至服务每天有数十亿调用量，响应时间偏慢。解决方案： 1、串行获取变并发获取，这样一些服务可以并发调用，在我们某个系统中能提升一倍多的性能，从原来TP99差不多1s降到500ms以下； 2、预取依赖数据回传，这种机制还一个好处，比如我们依赖三个下游服务，而这三个服务都需要商品数据，那么我们可以在当前服务中取数据，然后回传给他们，这样可以减少下游系统的商品服务调用量，如果没有传，那么下游服务再自己查一下。

假设一个读服务是需要如下数据： 1、数据A 10ms 2、数据B 15ms 3、数据C 20ms 4、数据D 5ms 5、数据E 10ms 那么如果串行获取那么需要：60ms； 而如果数据C依赖数据A和数据B、数据D谁也不依赖、数据E依赖数据C；那么我们可以这样子来获取数据：



那么如果并发化获取那么需要：30ms；能提升一倍的性能。

假设数据E还依赖数据F(5ms)，而数据F是在数据E服务中获取的，此时就可以考虑在此服务中在取数据A/B/D时预取数据F，那么整体性能就变为了：25ms。

通过这种优化我们服务提升了差不多10ms性能。

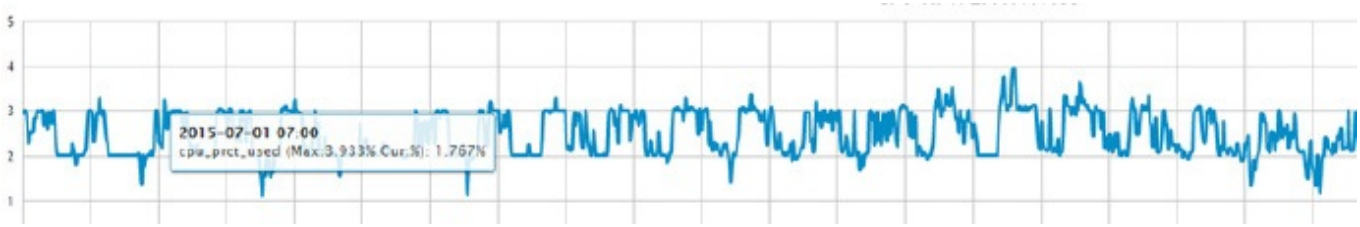
如下服务是在抖动时的性能，老服务TP99 211ms，新服务118ms，此处我们主要就是并发调用+超时时间限制，超时直接降级。

网络抖动时，返回**502**错误

Twemproxy配置的timeout时间太长，之前设置为5s，而且没有分别针对连接、读、写设置超时。后来我们减少超时时间，内网设置在150ms以内，当超时时访问动态服务。

机器流量太大

2014年双11期间，服务器网卡流量到了400Mbps，CPU 30%左右。原因是我们所有压缩都在接入层完成，因此接入层不再传入相关请求头到应用，随着流量的增大，接入层压力过大，因此我们把压缩下方到各个业务应用，添加了相应的请求头，Nginx GZIP压缩级别在2~4吞吐量最高；应用服务器流量降了差不多5倍；目前正常情况CPU在4%以下。



总结

数据闭环 数据维度化 拆分系统 Worker无状态化+任务化 异步化+并发化 多级缓存化 动态化 弹性化 降级开关 多机房多活 多种压测方案 Nginx接入层线上灰度引流 接入层转发时只保留有用请求头 使用不需要cookie的无状态域名（如c.3.cn），减少入口带宽 Nginx Proxy Cache只缓存有效数据，如托底数据不缓存 使用非阻塞锁应对local cache失效时突发请求到后端应用(lua-resty-lock/proxy_cache_lock) 使用Twemproxy减少Redis连接数 使用unix domain socket套接字减少本机TCP连接数 设置合理的超时时间（连接、读、写） 使用长连接减少内部服务的连接数 去数据库依赖（协调部门迁移数据库是很痛苦的，目前内部使用机房域名而不是ip），服务化 客户端同域连接限制，进行域名分区：c0.3.cn c1.3.cn，如果未来支持HTTP/2.0的话，就不再适用了。