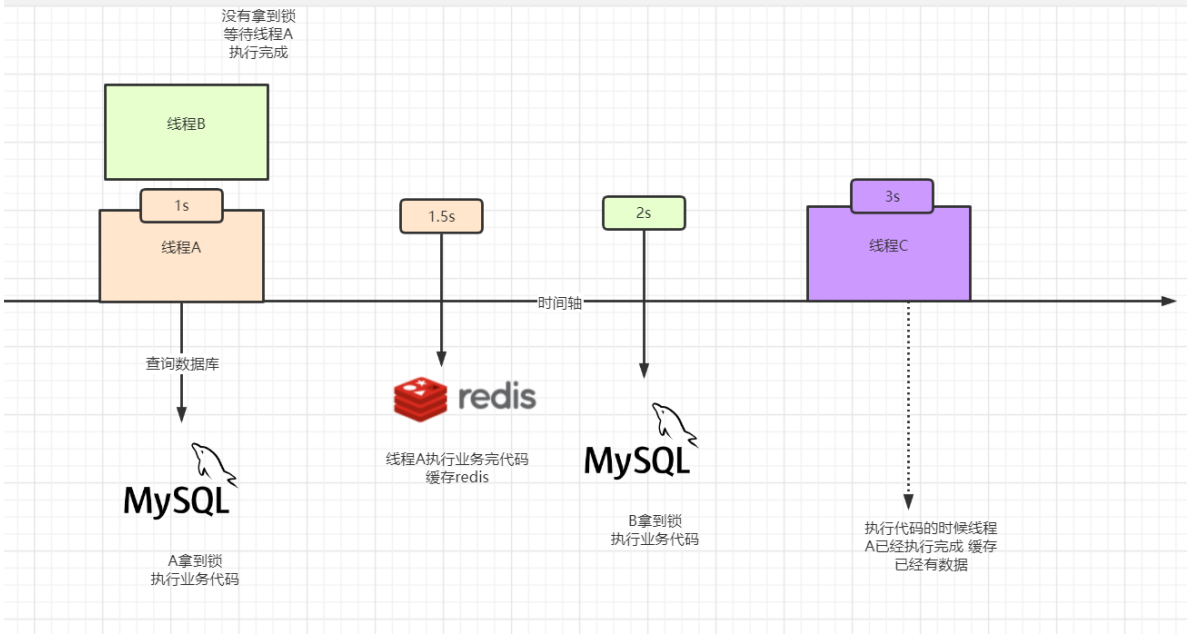
上节课问题：



两种方式：

双重检测：

```
lock.lock();
 productInfo = redisOpsUtil.get(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE +
id, PmsProductParam.class);
 if (null != productInfo) {
 return productInfo;
 }
 lock.unlock();
```

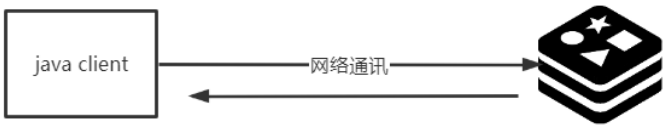
等待时间为0秒

```
 /**
 * 这里比上面多一个参数，多添加一个锁的有效时间
 *
 * @param waitTime 等待时间
 * @param leaseTime 锁有效时间
 * @param unit 时间单位 小时、分、秒、毫秒等
 */
 boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws Interru
ptedException;
```

问题：部分查询出null

解决方案：else中需要加入Tthread.sleep(10); 递归调用

一、后台缓存优化



一个网站的性能好与坏，网络IO和磁盘IO这两快的影响是比较大的。我们现在引入缓存目的是提高网站的性能，其实本质是不走磁盘走内存减少了磁盘IO来提高性能，但是我们有么有关系过，我们也增加了网络的操作。那这块我们能不能也优化了。来同学们思考下，给点思路
有同学讲到本地缓存，ok 把商品数据都缓存到本地缓存里面。来我们现在来写下这份代码。

二、引入本地缓存

```
private Map<String,PmsProductParam> cacheMap = new ConcurrentHashMap<>();
```

```
2
3  productInfo = cacheMap.get(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id);
4  //一级缓存
5  if(productInfo != null){
6   return productInfo;
7  }
8  //从二级缓存Redis里找
9  productInfo = redisOpsUtil.get(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE+id,Pm
sProductParam.class);
10  if(productInfo!=null){
11   cacheMap.put(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE+id,productInfo);
12   return productInfo;
13  }
14
15  else{
16   //缓存到一级缓存
17   cache.put(RedisKeyPrefixConst.PRODUCT_DETAIL_CACHE + id, productInfo);
18  }
19
20
```

PmsProductServic… l.java
9.02KB

三、Java本地缓存不足

可以缓存，如果有些商品访问越来越少，这个时候不需要缓存了，那请问我现在的本地缓存如何清理掉了？有同学说可以用map中的remove，如果直接用可以清楚掉吗?是可以删除，但是你怎么知道哪些数据是删除的哪些数据是不需要删除的了？这是不是得一个算法来统计下这个访问量，然后根据排序来选择删除部分数据.

四、引入gava缓存

```
1  <dependency>
2   <groupId>com.google.guava</groupId>
3   <artifactId>guava</artifactId>
```

```
 <version>22</version>
</dependency>

```

利用guava的缓存：

```
com.tuling.tulingmall.component.LocalCache
```

1、设置最大容量 2、初始化容量 3、缓存过期

多少层缓存？？？ 两层 本地缓存 redis缓存

布隆过滤器：

演示效果：

http://localhost:8866/pms/productInfo/261

存储：

com.tuling.tulingmall.config.BloomFilterConfig#afterPropertiesSet

拦截匹配：

com.tuling.tulingmall.intercepter.BloomFilterInterceptor

全量刷到》增量

课上问题：

分布式下，本地缓存、redis、数据库的数据怎么保持伪实时同步？（后面课程会讲到）

会不会出现多个接口在一台服务器呀？如果多个不同接口怎么处理，比如商品id查询接口订单id查询接口 ，这时候拦截器判断时 要声明多个布隆过滤器处理吗？（不会，存储是用的redis）

4核8G的机子并发在多少合适 （200-500）

项目代码的git 地址有吗（百度网盘）

用guava 本地缓存 分布式环境也可以用吗？（）

单个服务 400 /s 是高了还是低了？ 如果低了，求优化策略，多少才算 高?

tryLock里面是什么原理呀?

老师，我启动之后OrderApplication定时报

java.lang.IllegalArgumentException: protocol = http host = null的(提知识星球)

如果有一千万的商品布隆过滤器岂不是有redis大key的问题 （key 不存：value） 1212

收集10个问题 其他问题结束课时在讨论

五、终结方案

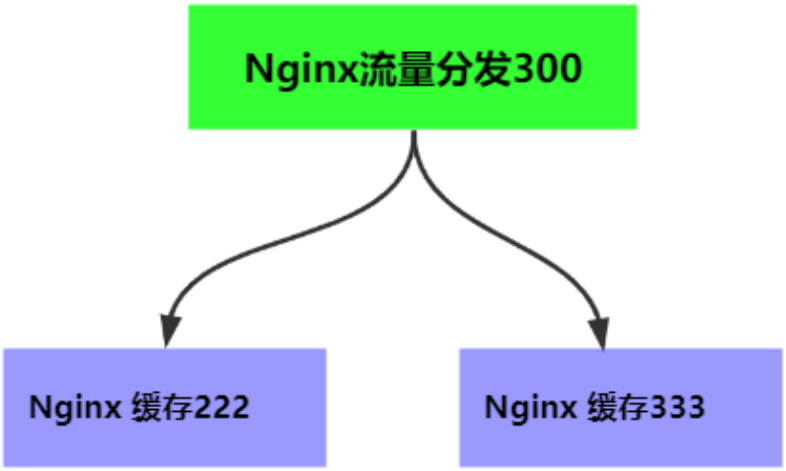
一、当前不足、主要讲前端问题

小流量架构：https://www.processon.com/view/link/5e5774dae4b0cb56daac5a80
问题：
如果后台数据有变更呢?如何及时同步到其它服务端? 页面在不通服务间如何同步
如果页面静态化了，我们搜索打开一个商品详细页，怎么知道要我需要的访问的静态页面?
万一我们模板需要修改了怎么办?
牵一发动全身。

二、静态化这块
大型网站架构：https://www.processon.com/view/link/5e57735ce4b0362765065cf3

OpenResty® 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。
OpenResty® 通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 Web 应用平台。这样，Web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10K 乃至 1000K 以上单机并发连接的高性能 Web 应用系统。
OpenResty® 的目标是让你的Web服务直接跑在 Nginx 服务内部，充分利用 Nginx 的非阻塞 I/O 模型，不仅仅对 HTTP 客户端请求,甚至于对远程后端诸如 MySQL、PostgreSQL、Memcached 以及 Redis 等都进行一致的高性能响应。
http://openresty.org/cn/

三、lua演示效果
目录：D:\ProgramData\nginx
需要用到的nginx命令：taskkill /im nginx.exe /f
演示地址：http://localhost:300/product?method=product&productId=12



实战：

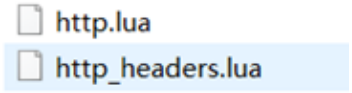
流量分发的nginx，会发送http请求到后端的应用层nginx上去，所以要先引入lua http lib包

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

最近网络不稳定，也可以从：https://github.com/bungle/lua-resty-template 去下载这两个lua脚本

crc32_long



文件放在：openresty-1.15.8.2-win642**\lualib\resty**下（openresty-1.15.8.2-win643也要放）

操作：

在nginx.conf文http模块中加入：

导入lua相关的包

lua_package_path '../lualib/?.lua;;';

lua_package_cpath '../lualib/?.so;;';

```
lua_package_path '../lualib/?.lua;;';
lua_package_cpath '../lualib/?.so;;';
include lua.conf;
```

我们来看下lua.conf文件中

```
server {
listen 300;
location /product {
default_type 'text/html;charset=UTF-8';
lua_code_cache on;
```

```
  content_by_lua_file D:/ProgramData/nginx/dis.lua;
  }
}
```

我们监听300这个端口号，如果请求路径是product 那我们就让他包含dis.lua文件的内容并且是开启lua缓存的哦。

接下来我们来看下dis.lua脚本

dis.lua

```
local uri_args = ngx.req.get_uri_args()
local productId = uri_args["productId"]
local host = {"127.0.0.1:222","127.0.0.1:333"}
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,
 keepalive=false
})

if not resp then
 ngx.say("request error :", err)
 return
end

ngx.say(resp.body)

httpc:close()
```

这个文件大家看阅读下源码，大致意思就是截取传过来的productid 然后从我配置的host服务中hash取其中已一台服务，然后通过服务请求拿到相应的数据，然后对数据进行输出。

接下来我们再看下222和333这两个服务。我们来看下nginx.conf配置：

```
  lua_package_path '../lualib/?.lua;;';
  lua_package_cpath '../lualib/?.so;;';
  include lua.conf;
```

lua.conf 配置

```
lua_shared_dict my_cache 128m;
 server {
 listen 222;
 set $template_location "/templates";
 set $template_root "D:/ProgramData/nginx/";

 location /product {
 default_type 'text/html;charset=UTF-8';
 lua_code_cache on;
  content_by_lua_file D:/ProgramData/nginx/product.lua;
  }
}
```

请求后台：

product.lua

```
local uri_args = ngx.req.get_uri_args()
local productId = uri_args["productId"]
local productInfo = nil
if productInfo == "" or productInfo == nil then
 local http = require("resty.http")
 local httpc = http.new()
 local resp, err = httpc:request_uri("http://127.0.0.1:8866",{
 method = "GET",
 path = "/pms/productInfo/"..productId
  })
  productInfo = resp.body
end
ngx.say(productInfo);

```

此时我们可以正常去访问后台，并且能拿到后台的数据，那我们请求都会请求到后台，那就没有起到作用了。

加入缓存(升级版)：

product.lua

```
local uri_args = ngx.req.get_uri_args()
local productId = uri_args["productId"]
local cache_ngx = ngx.shared.my_cache
```

```
local productCacheKey = "product_info_"..productId
local productCache = cache_ngx:get(productCacheKey)
if productCache == "" or productCache == nil then
 local http = require("resty.http")
 local httpc = http.new()
 local resp, err = httpc:request_uri("http://127.0.0.1:8866",{
  method = "GET",
  path = "/pms/productInfo/"..productId
  })
  productCache = resp.body
  local expireTime = math.random(600,1200)
  cache_ngx:set(productCacheKey, productCache, expireTime)
end
```

加入缓存 现在我们来看下运行效果。、

访问地址：http://localhost:300/product?method=product&productId=26

json：

```
local cjson = require("cjson")
local productCacheJSON =cjson.decode(productCache)
ngx.say(productCache);
local context = {
 id = productCacheJSON.data.id,
 name = productCacheJSON.data.name,
 price = productCacheJSON.data.price,
 pic = productCacheJSON.data.pic,
 detailHtml = productCacheJSON.data.detailHtml
}
local template = require("resty.template")
template.render("product.html", context)
```

html模板：

```
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
 </head>
 <body>
 <h1>
商品id: {* id *}<br/>
商品名称: {* name *}<br/>
商品价格: {* price *}<br/>
```

```
10 商品库存: <img src={* pic *}/><br/>
11 商品描述: {* detailHtml *}<br/>
12  </h1>
13 </body>
14 </html>
```

product.lua
1KB

product.html
327B

dis.lua
635B

总结:

数据热点:

3层缓存:

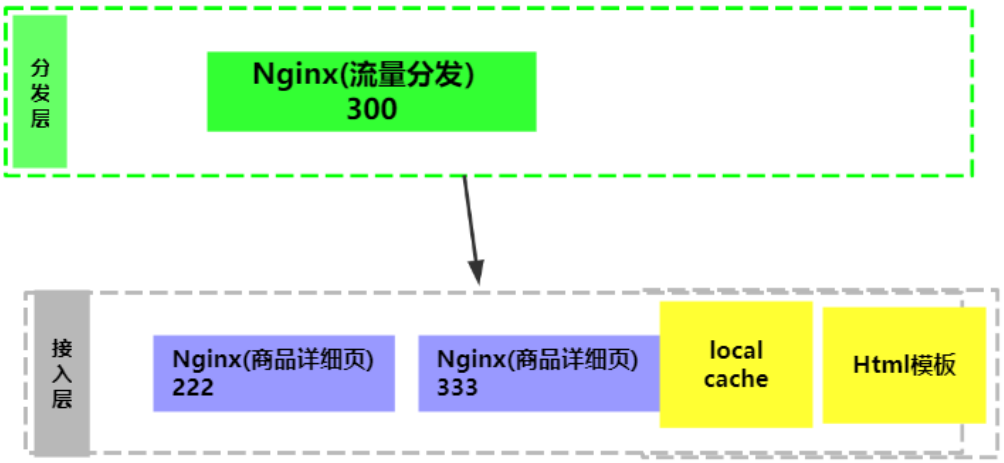
lua+nginx:数据里小、访问量相对来很高

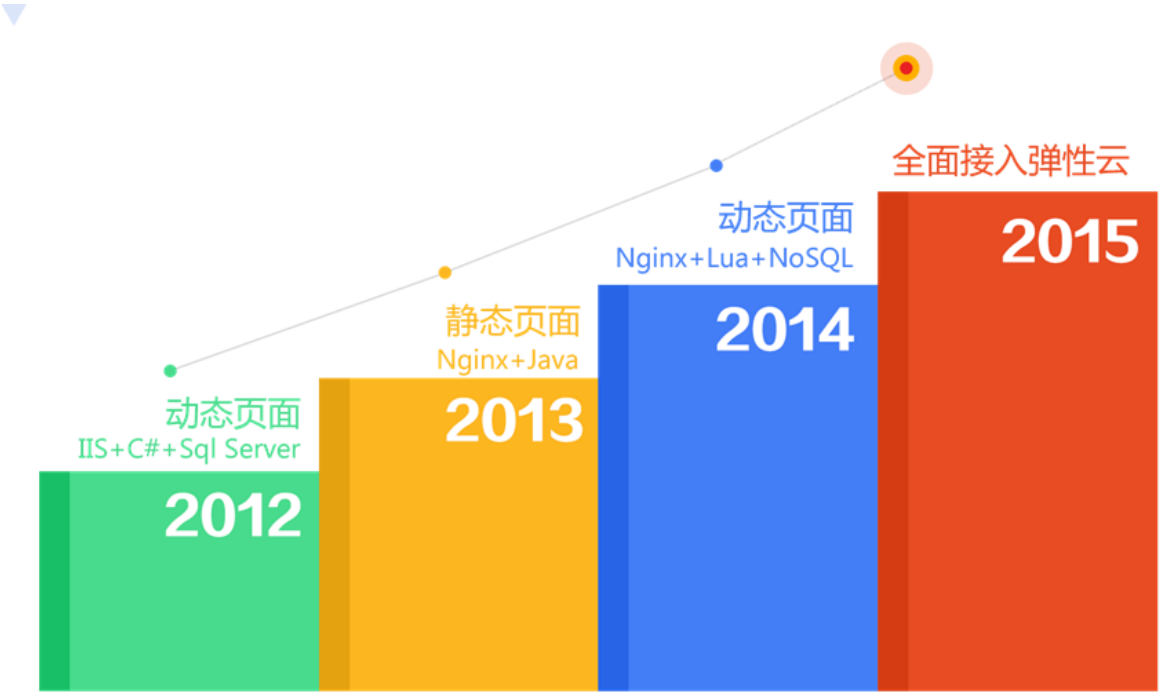
jvm本地缓存:数据里很大、访问量相对来高

redis:数据里相对来说比较大、访问量相对来不高

LRU算法:最热的数据缓存

LRU-k(链表K) 最近最热的一次数据 非常火 连续访问3次以上



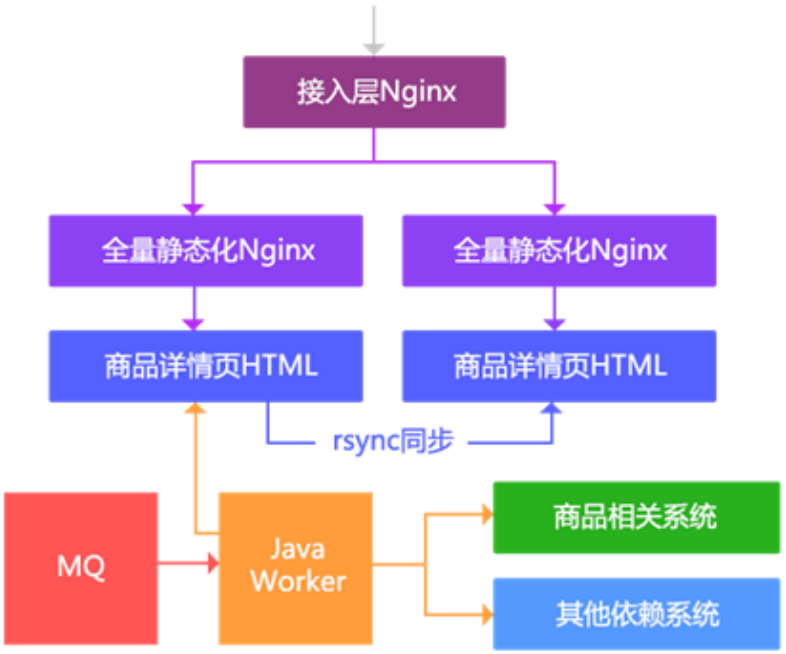


架构1.0：

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。

备注：加入了缓存，但是问题受限于缓存服务器，不稳定

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。

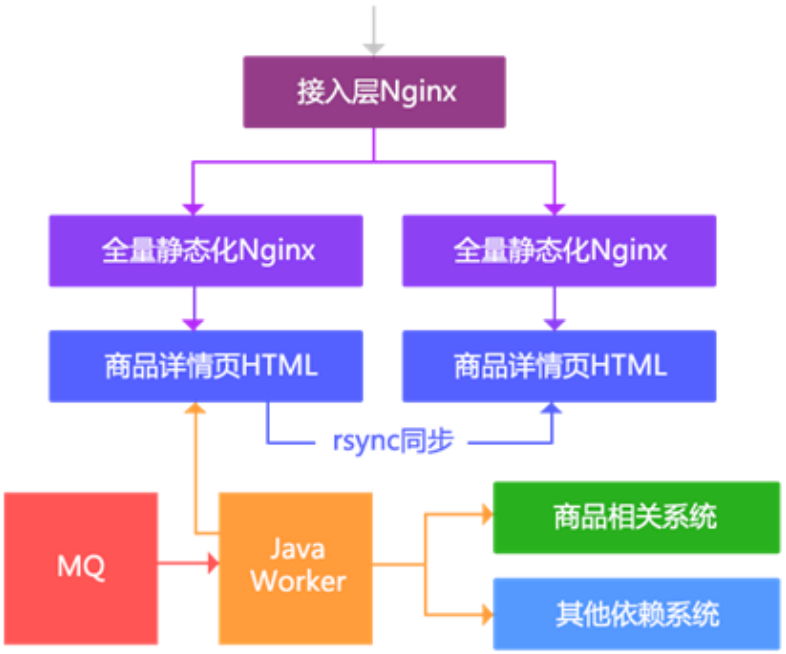
备注：该方案的主要缺点：

1、假设只有分类、模板变更了，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；

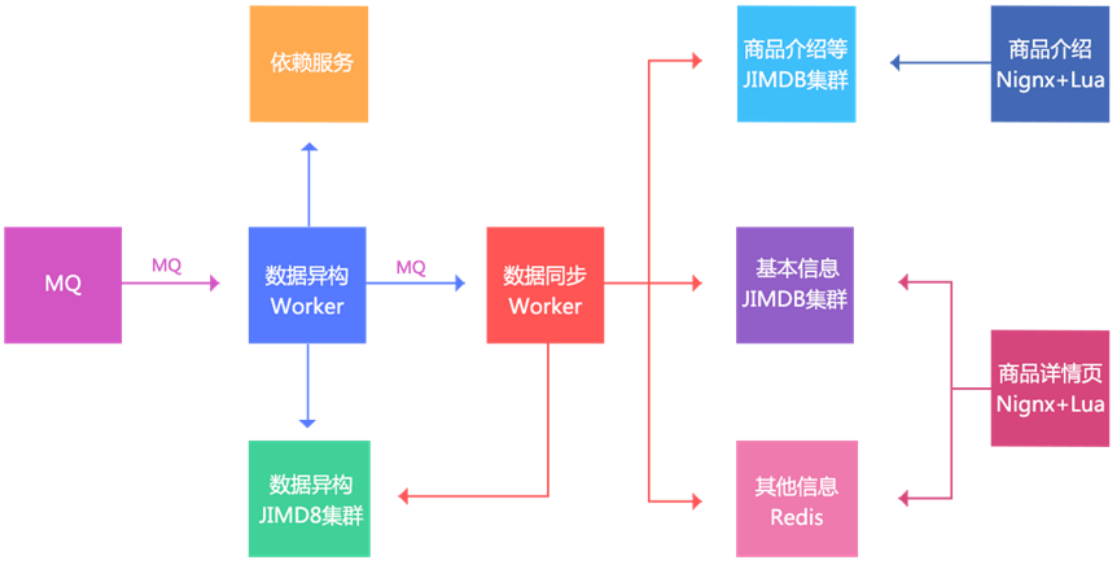
2、支持各种垂直化页面改版；

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：

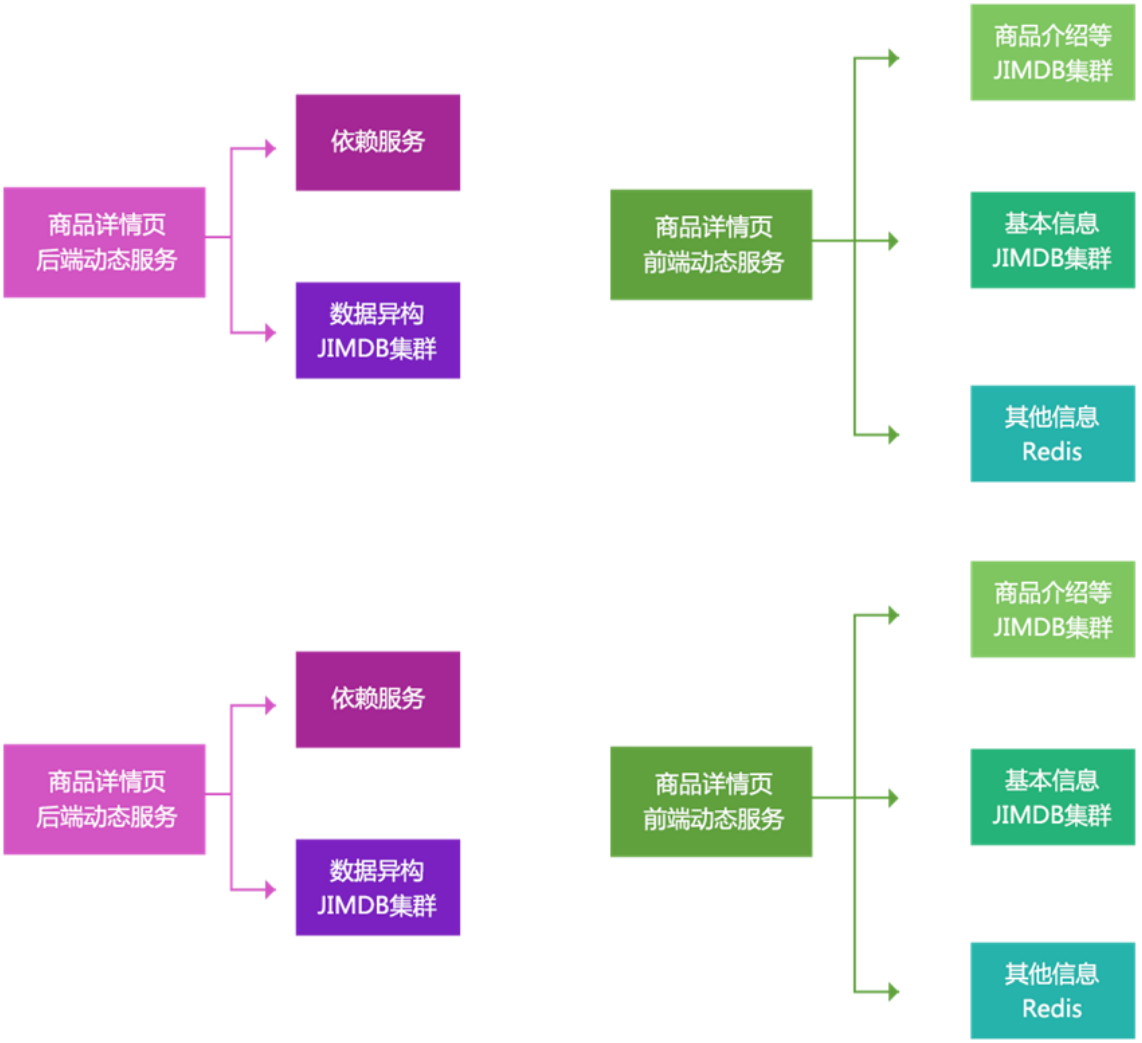
1、数据变更还是通过MQ通知；

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）；

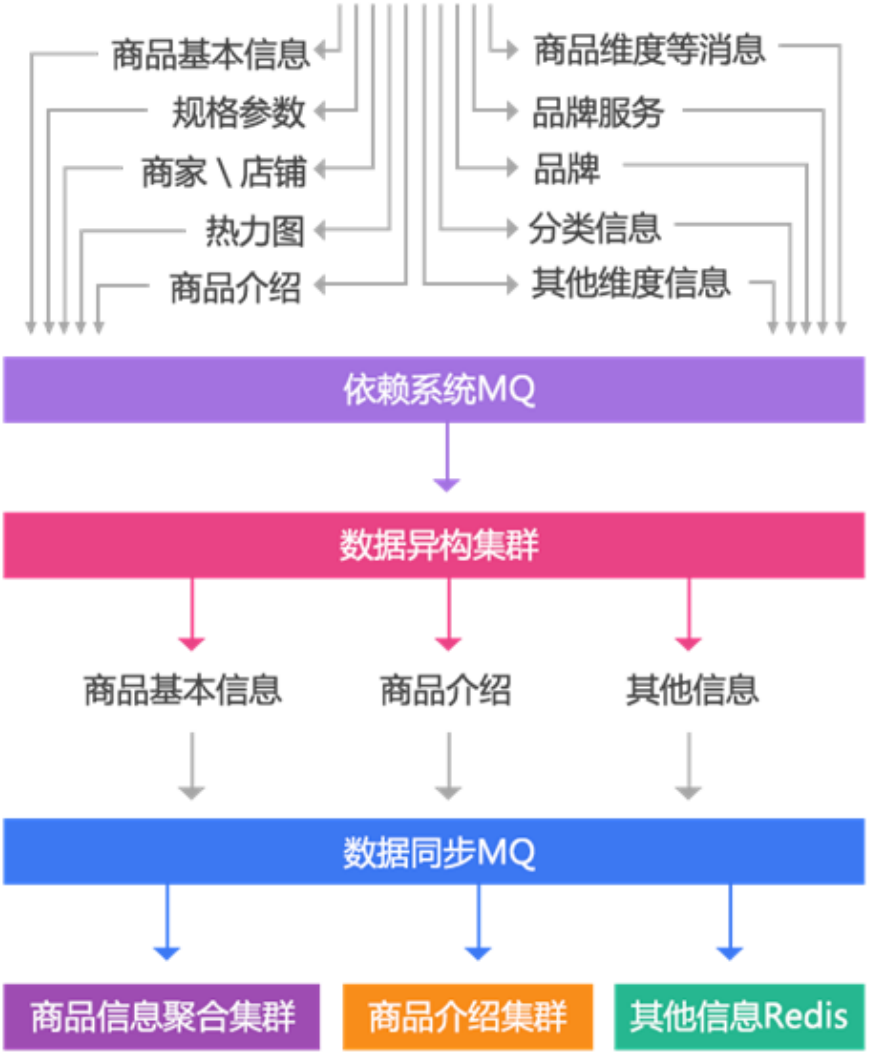
4、前端展示分为两个：商品详情页和商品介绍，使用Nginx+Lua技术获取数据并渲染模板输出。

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。

架构原则：



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；

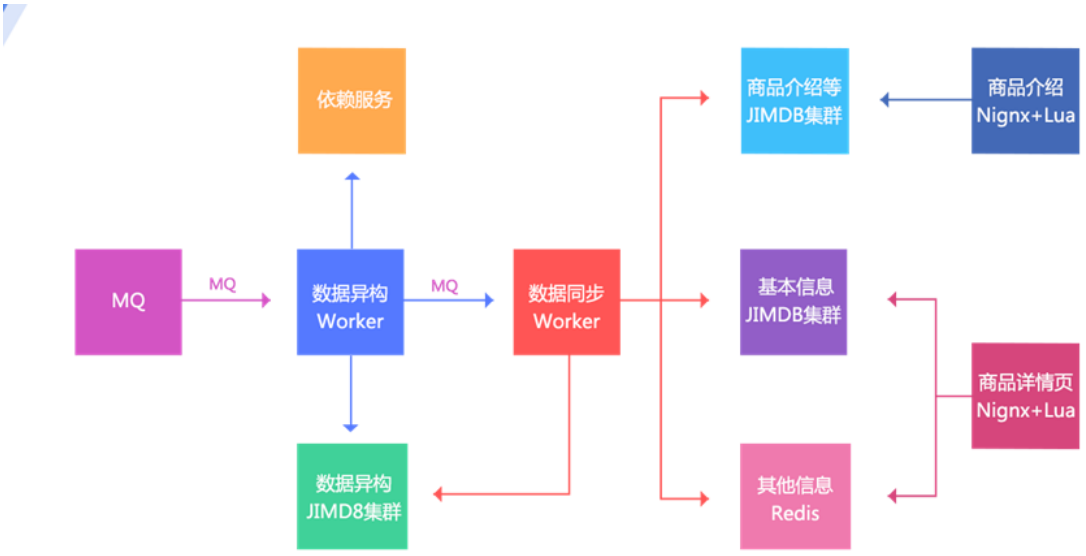
2、商品介绍信息，商品维度商家模板、商品介绍等；

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。



多级缓存优化：

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。

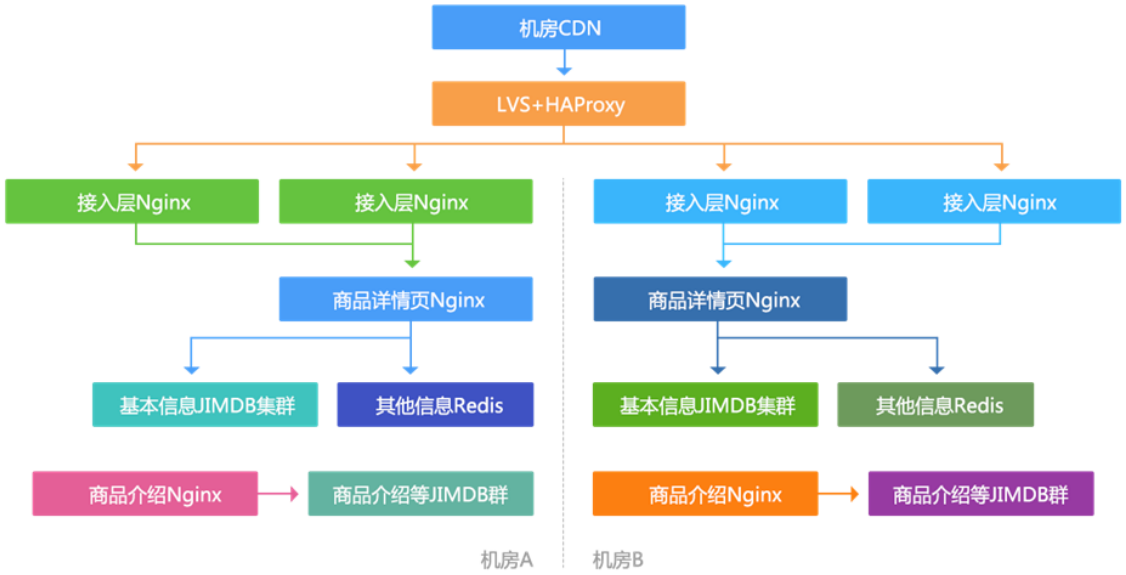
降级开关

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

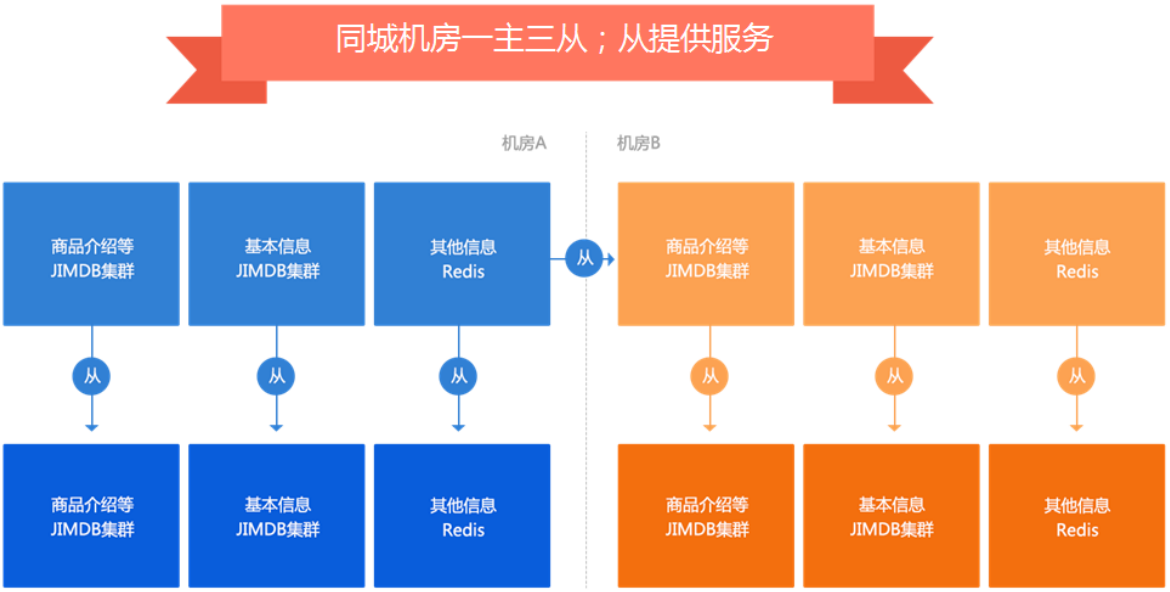
业务处理影响到其他业务处理。慢的还是慢，但是不影响其他的业务。我们通过这种机制还可以把tomcat线程池的监控拿出来，出问题时可以直接清空业务线程池，另外还可以自定义任务队列来支持一些特殊的业务。

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。

文档：04秒杀系统-商品详细页多级缓存实战三....
链接：http://note.youdao.com/noteshare?id=63d31b96ca60b59374650cec2e6f4b3e&sub=6417557878F74BF9BD3493909C17BDB6