秒杀系统和商品详情页系统

一号店 基础架构组 王伟

Agenda

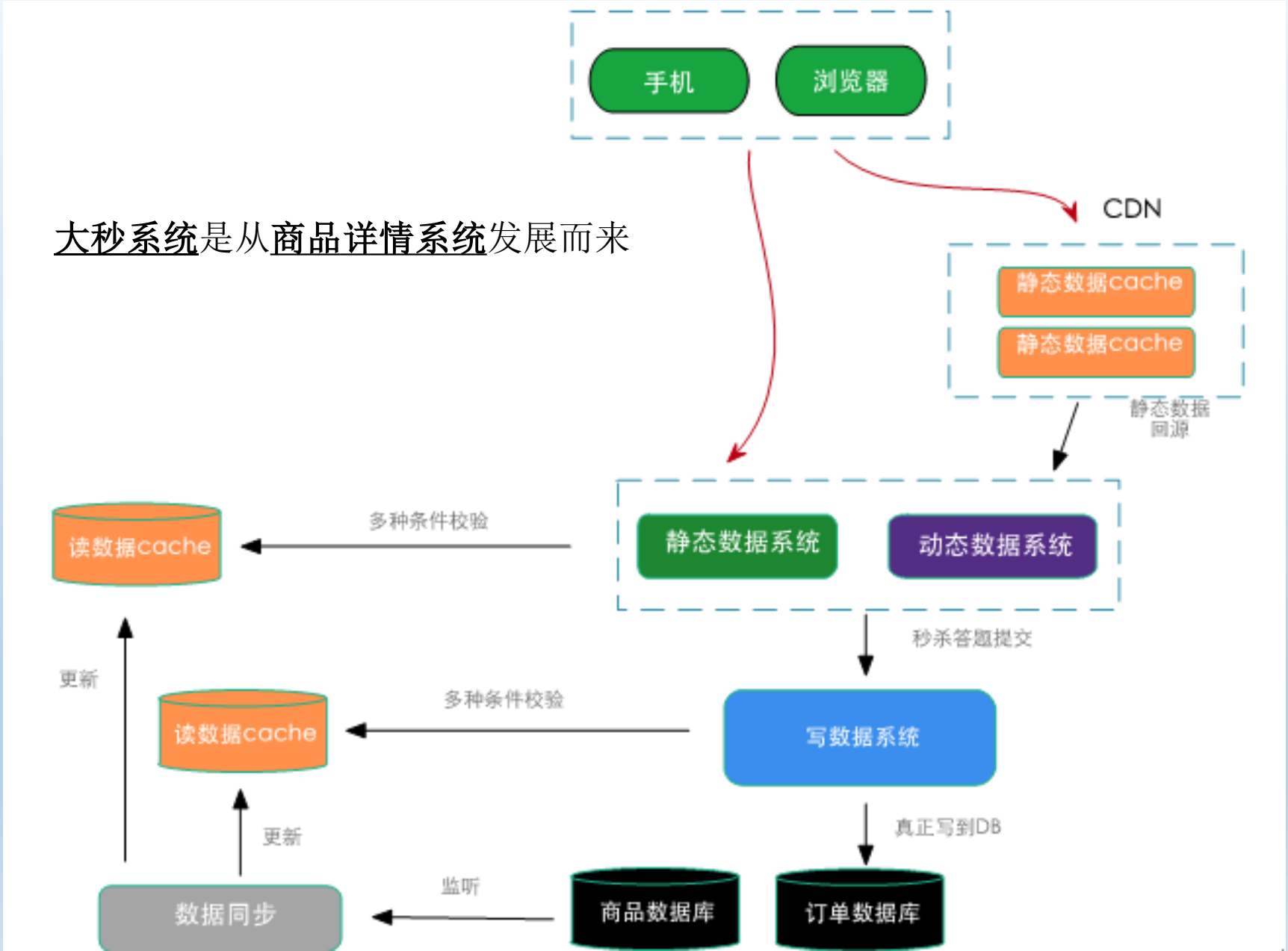
- Taobao秒杀系统（大秒）
- JD商品详情页系统+详情页统一服务系统
- VIP商品详情页系统

一些数据

- 三款小米手机各11万台
- 3分钟后成为双十一最快破亿的旗舰店
- **前端系统**双11峰值有效请求约**60w/s+的QPS**
- 而**后端cache**的集群峰值近2000w/s， **单机也近30w/s的QPS**
- 最高下单减库存**tps 1500/s**。

overview

- 大秒



大秒系统是从商品详情系统发展而来

大秒系统

- 先做数据的**动静分离**
- 将**90%的数据**缓存在**客户端浏览器**
- 将动态请求的读数据Cache在Web端
- 对**读**数据**不做强一致性**校验
- 对**写**数据进行**基于时间的合理分片**
- 对**写**请求做**限流**保护
- 对**写**数据进行**强一致**性校验

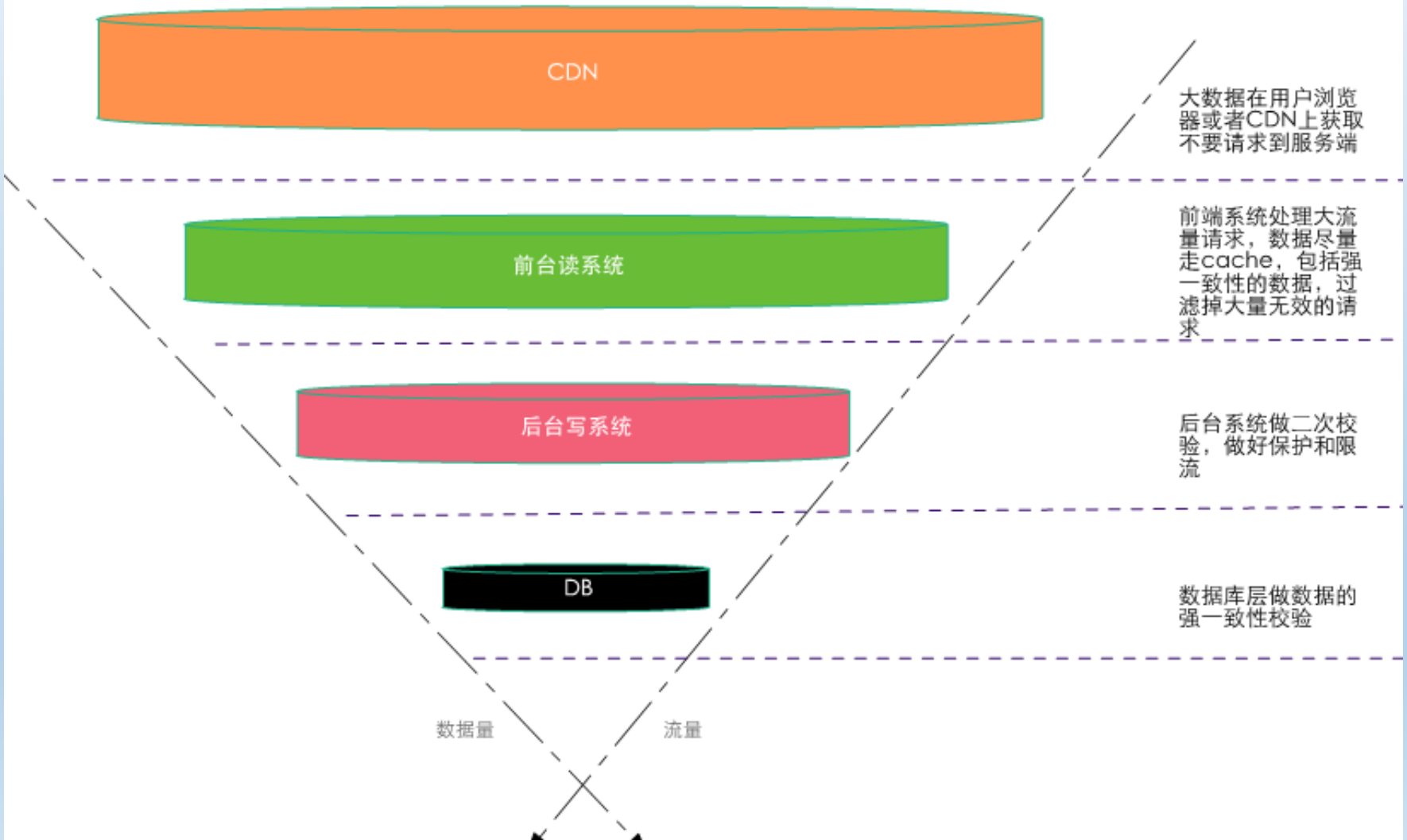
✓ 把大量静态不需要检验的数据放在**离用户最近的地方**；

✓ 在**前端读系统**中**检验**一些基本信息，如用户是否具有秒杀资格、商品状态是否正常、用户答题是否正确、秒杀 是否已经结束等；

✓ 在写数据系统中**再校验**一些如是否是非法请求，营销等价物是否充足（淘金币等），**写的数据一致性**如检查库存是否还有等；

✓ 最后在**数据库层**保证 数据**最终准确性**，如库存不能减为负数。

流量漏斗


CDN
前台读系统
后台写系统
DB
大数据在用户浏览器或者CDN上获取不要请求到服务端
前端系统处理大流量请求，数据尽量走cache，包括强一致性的数据，过滤掉大量无效的请求
后台系统做二次校验，做好保护和限流
数据库层做数据的强一致性校验
数据量
流量

基于时间分片削峰

把峰值的下单请求给拉长了，**从以前的1s之内延长到2~10s左右，**请求峰值基于时间分片了，这个时间的分片对服务端处理并发非常重要，会减轻很大压力.

同一商品大并发读问题

- ✓ 读的场景可以允许一定的脏数据
- ✓ 单点瓶颈问题:Tair缓存机器单台能支撑30w/s的请求，大秒这种级别的热点商品还远不够 。

  Solution： 应用层的Localcache

同一数据大并发更新问题

问题：

大量的线程竞争**InnoDB行锁**，并发大，线程越多，TPS下降，RT上升，数据库吞吐严重受到影响。

单个热点商品会影响整个数据库的性能，**0.01%商品影响99.99%的商品**。

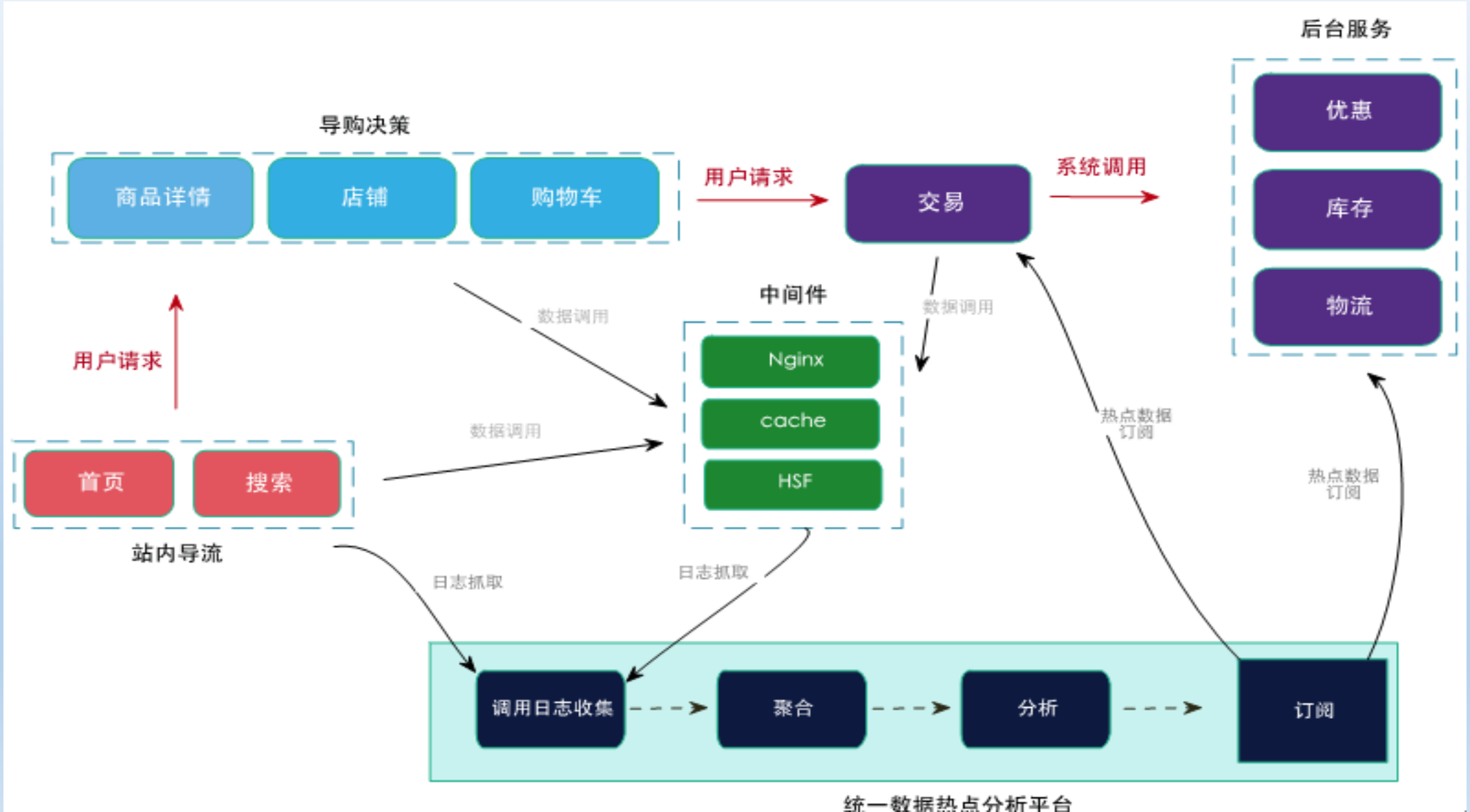
Solution： 隔离，把热点商品放到单独的**热点库**中。

解决并发锁

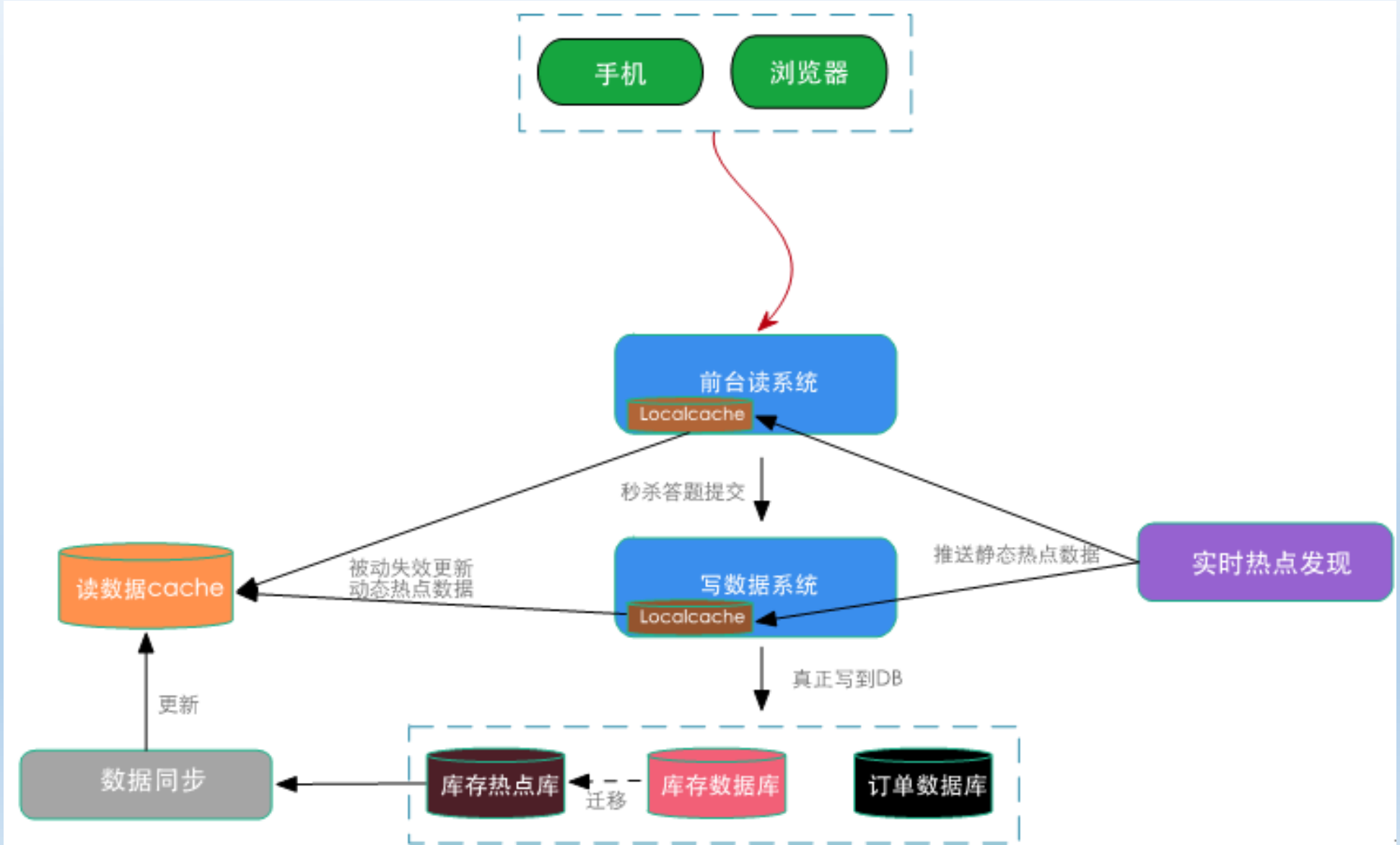
- **应用层做排队**
- **数据库层做排队**

在数据库层做全局排队是最理想的，淘宝的数据库团队开发了针对MySQL的InnoDB层上的patch，**可以做到数据库层上对单行记录做到并发排队**

热点数据实时移动


后台服务
优惠
库存
物流
导购决策
商品详情
店铺
购物车
用户请求
交易
系统调用
中间件
数据调用
数据调用
Nginx
cache
HSF
用户请求
首页
搜索
站内导流
数据调用
热点数据订阅
热点数据订阅
日志抓取
日志抓取
调用日志收集
聚合
分析
订阅
统一数据热点分析平台

热点数据实时移动


手机
浏览器
前台读系统
Localcache
秒杀答题提交
读数据cache
被动失效更新
动态热点数据
写数据系统
Localcache
推送静态热点数据
实时热点发现
真正写到DB
更新
数据同步
库存热点库
迁移
库存数据库
订单数据库

产品

- 数据同步

  Canal : 基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了mysql

  RDS（包括DTS）：

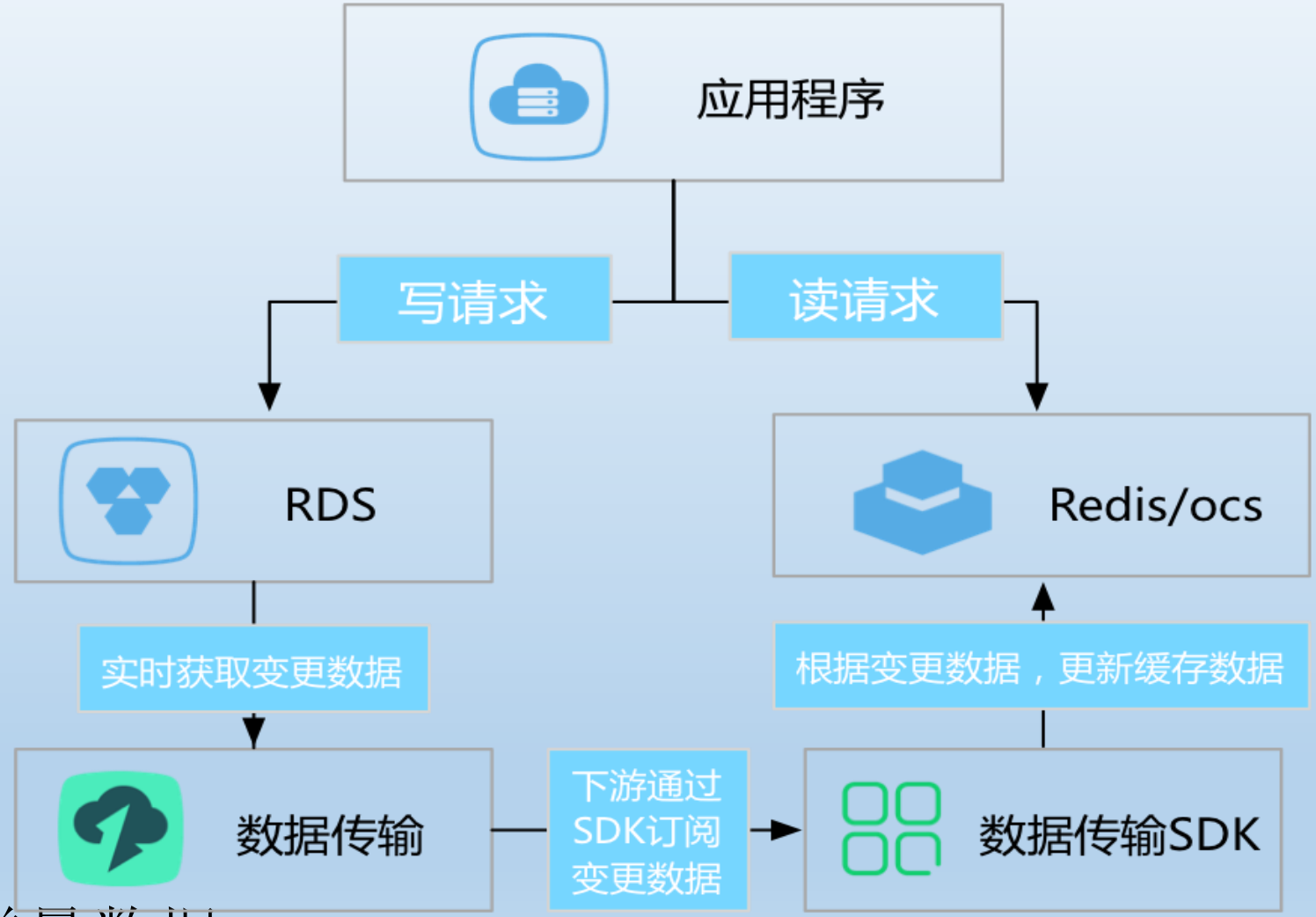
  一种稳定可靠、可弹性伸缩的在线数据库服务；

  云产品

- 缓存： OCS是分布式缓存产品；

  核心存储是淘宝的开源产品TAIR；

缓存失效(invalid)



RDS实时增量数据

DTS提供了RDS MySQL增量数据订阅的功能

Cache和db一致性

- **缓存失效一致性问题**

  DB和cache双写（先写cache）

  solution：1. OCS **MVCC（多版本控制），乐观锁（类似CAS）**

  2. 增量消息

双写的好处：简单

双写的坏处：

✓ **延迟**：用户请求线程里同步完成DB、缓存双写，变更请求链路长，访问延迟大，影响用户体验

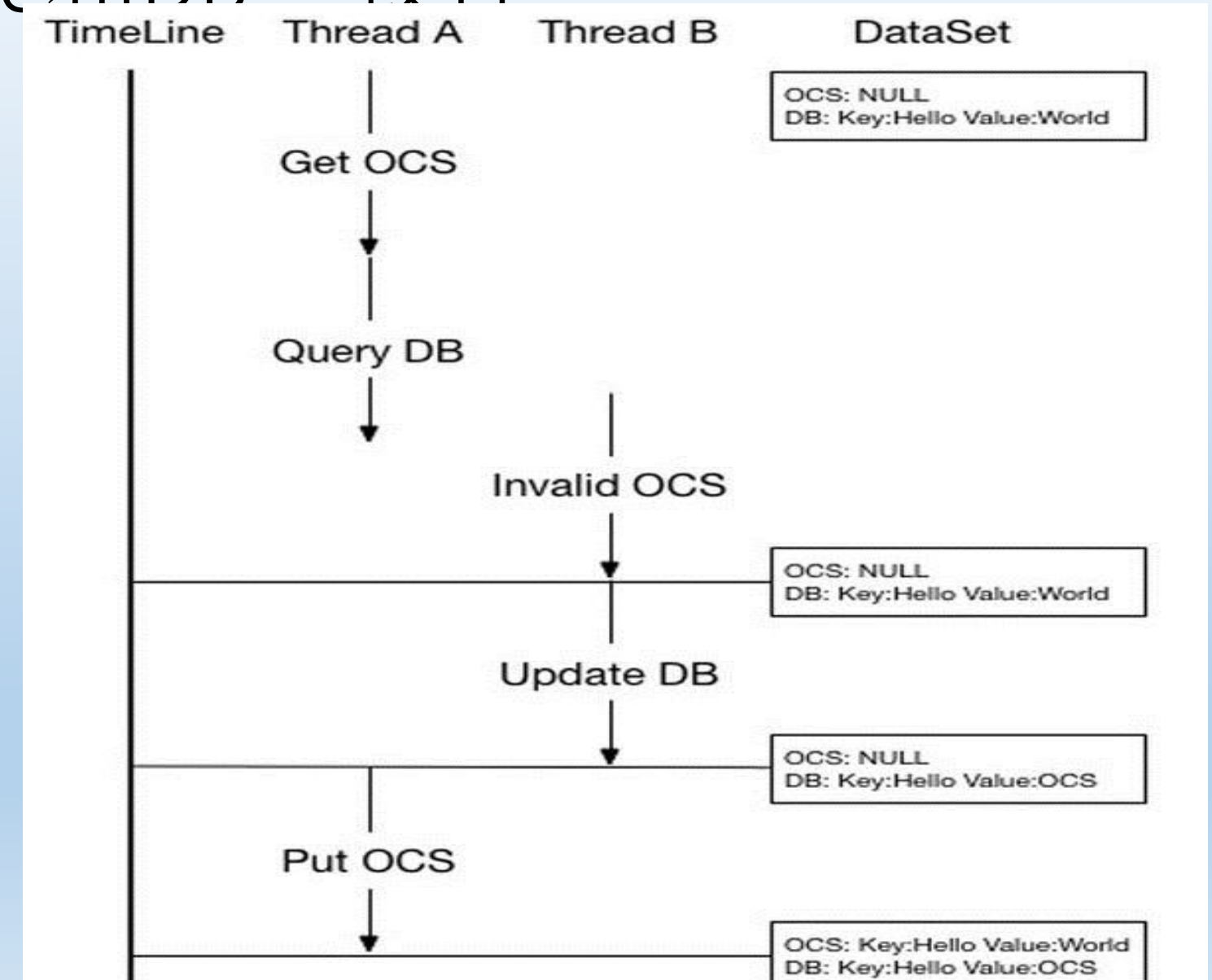
✓ **异常**：用户请求线程里同步调用缓存，对缓存存在**强依赖**，遇到缓存**超时等异常**时，**没有办法做到有效的重试**，遇到异常给用户返回系统错误、操作失败等信息，严重影响用户体验

✓ 事务在提交时失败则写缓存是不会回滚的造成**DB和缓存数据不一致**；

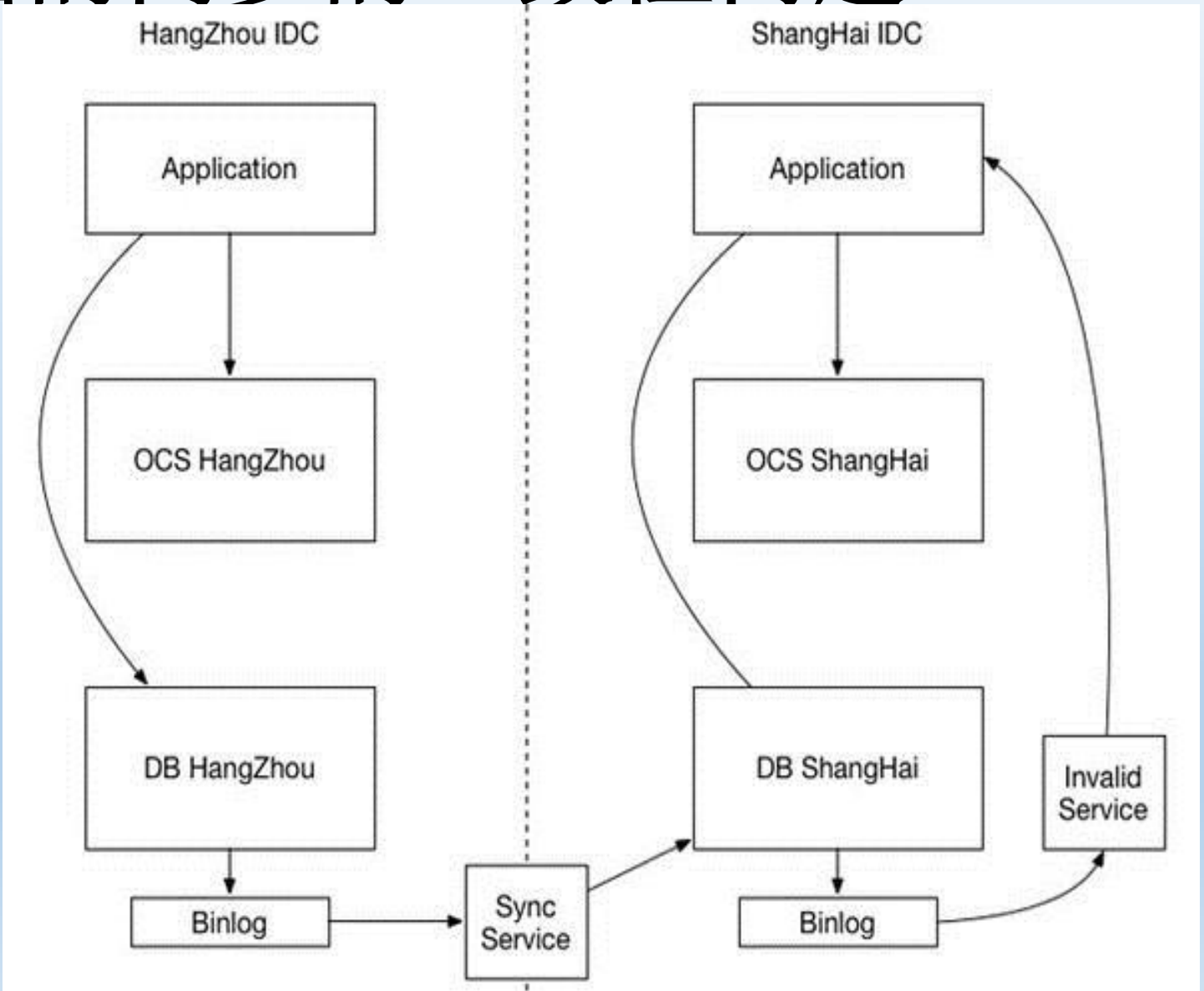
✓ 假设多个人**并发写缓存**可能出现**脏数据**的；

✓ **同步写**对**性能**有一定的影响，**异步写**存在**丢数据**的风险。

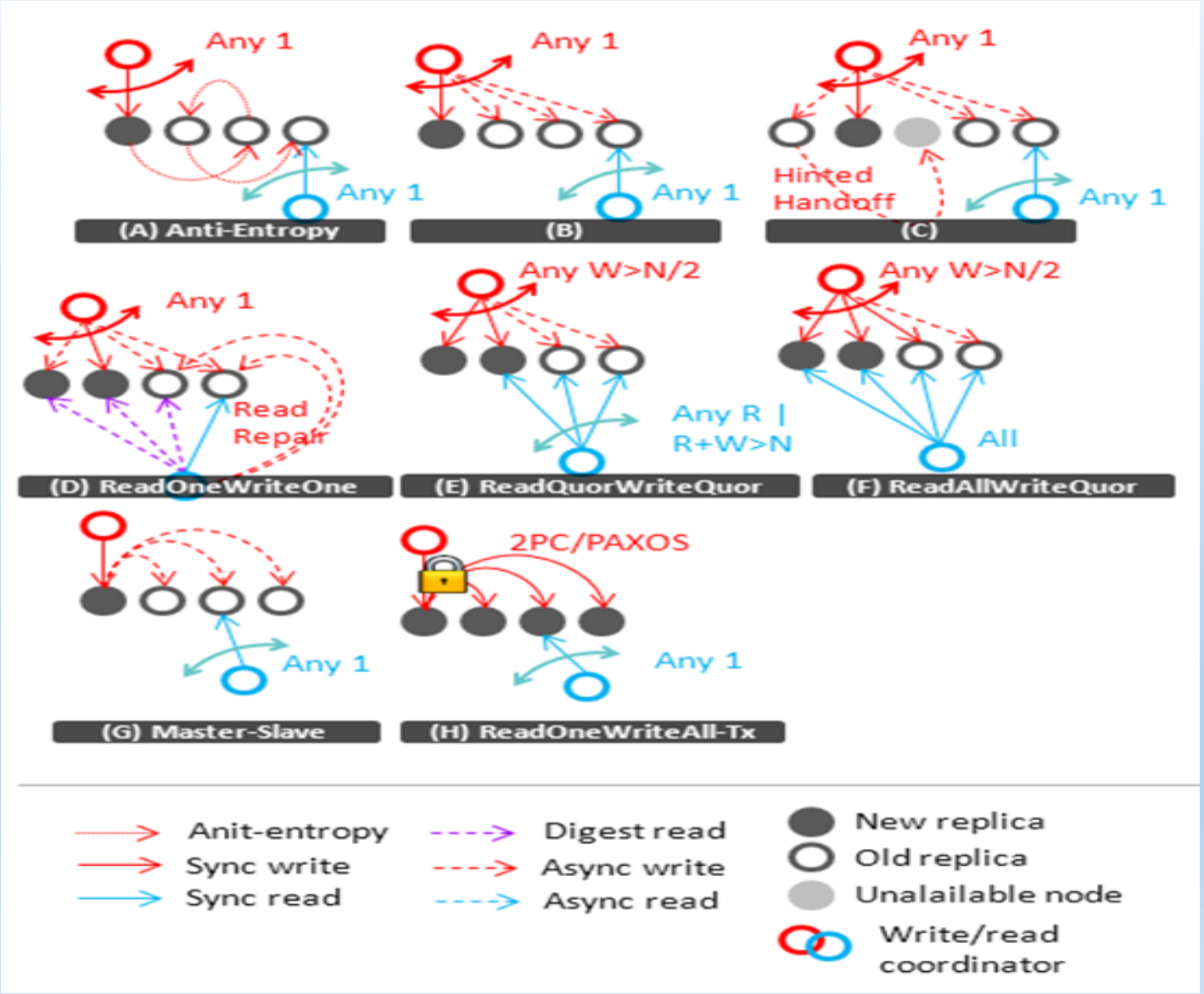
Cache和DB一致性



缓存数据的同步的一致性问题



一致性（多副本）


Any 1
Any 1
(A) Anti-Entropy
Any 1
Any 1
(B)
Any 1
Hinted
Handoff
Any 1
(C)
Any 1
Read
Repair
(D) ReadOneWriteOne
Any W>N/2
Any R |
R+W>N
(E) ReadQuorWriteQuor
Any W>N/2
All
(F) ReadAllWriteQuor
Any 1
(G) Master-Slave
2PC/PAXOS
Any 1
(H) ReadOneWriteAll-Tx
Anit-entropy
Sync write
Sync read
Digest read
Async write
Async read
New replica
Old replica
Unalailable node
Write/read
coordinator

一致性（多副本）

（A, 反熵） 一致性最弱， Cassandra， BT(Bit-Torrent)

（B，定向的反熵）

（C，B + hinted handoff） Amazon ’s Key-value Store Dynamo

（D，一次性读写，read repaired）A，B， C， D Cassandra

（E， WriteQurum ReadQurum） R+W>N, Dynamo

三副本： 2，2， 3

五副本： 3， 3， 5

(F, WriteQurum ReadAll)

(G Master-slave)

mysql (binlog-relayLog) 默认异步，V5.5 半同步， redis rdb,

大秒 **cache + db 一致性**

(H ReadOne-WriteAll) 2PC-Paxos

2PC: JTA(XA) Paxos :Google的分布式锁*系统*Chubby

一致性协议

2PC; Paxos（节点对等, 难理解）;

Raft（相对Paxos简单，主从，三个阶段）: etcd(类似zk)， Jgroup（Jboss cache 复制的事务处理缓存）

Vector clock 向量时钟 (Dynamo)　微信朋友圈的评论（因果一致性）;

Zab协议： 2f+1个节点，允许f个节点失败（ **Quorum** ）。 两个主流程，三个阶段。

RWN协议（ *Dynamo* ）: R 读， W写，N副本数。自定义的一致性。

Kafka ISR(in-sync replicas): ISR set是强一致的， ISR set之外的副本是落后的，非强一致的。

相对于zab的Quorum性能提高了，同步时间少。生产2个副本（ES）。

一致性

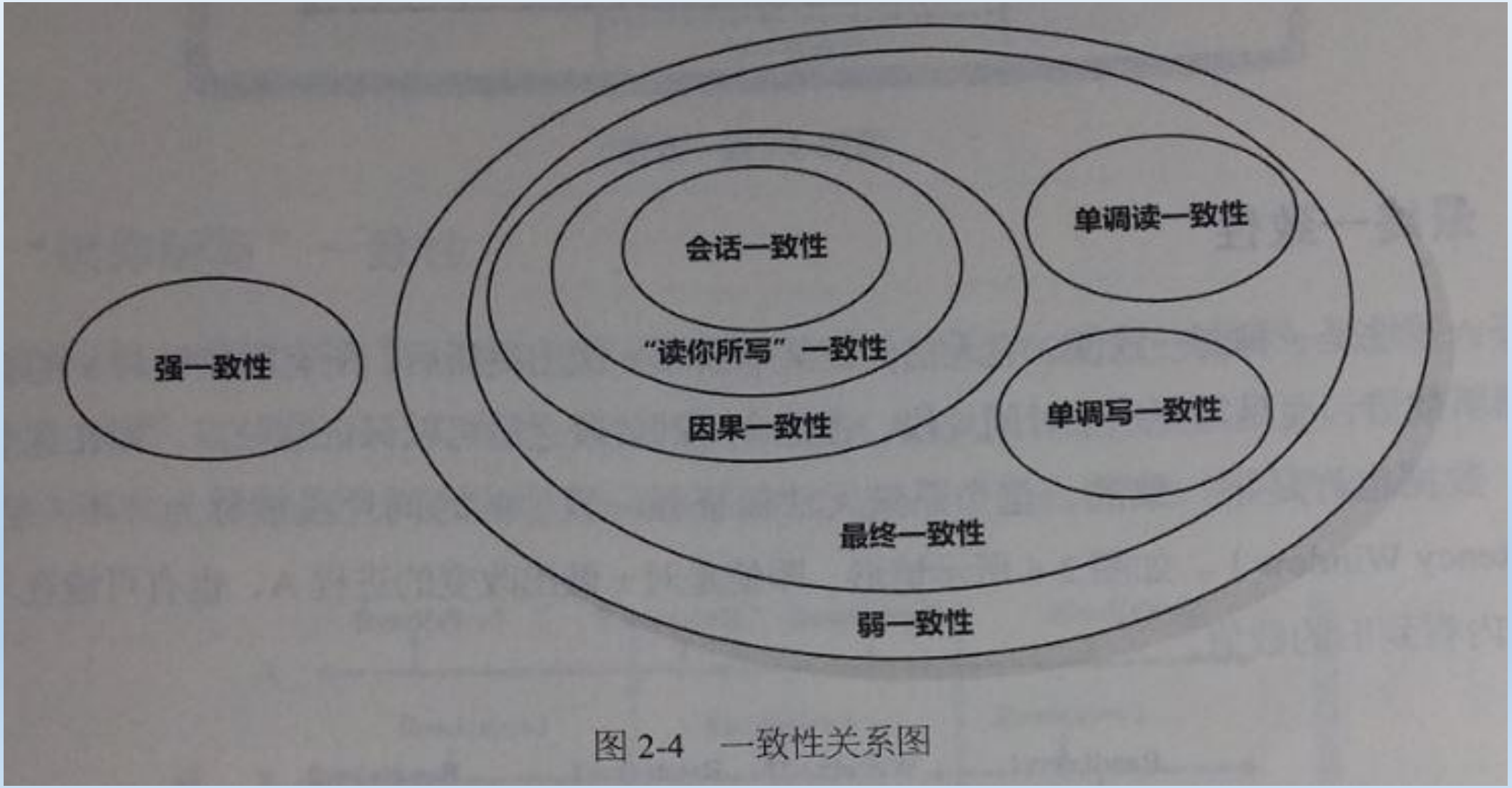


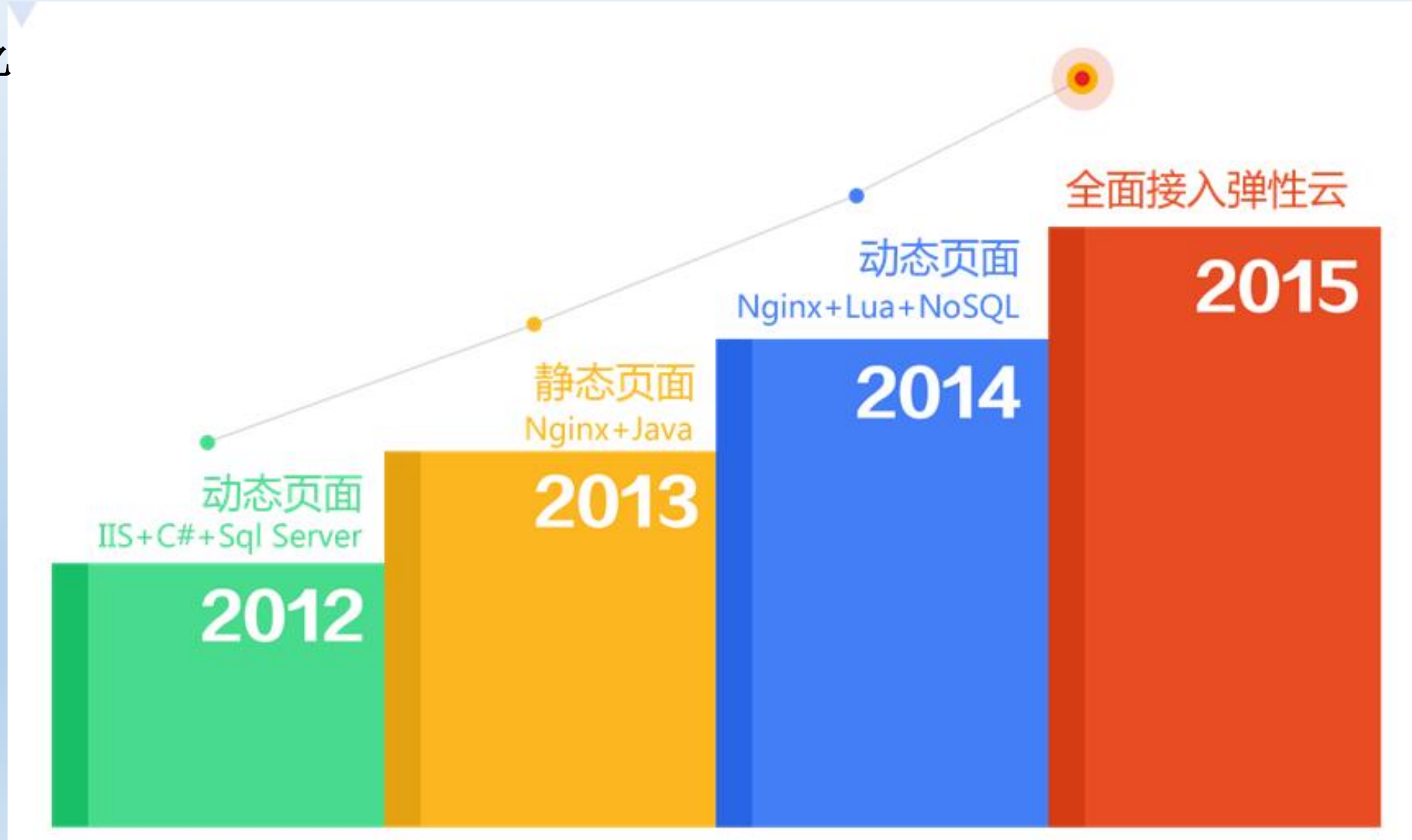
图 2-4　一致性关系图

大秒总结

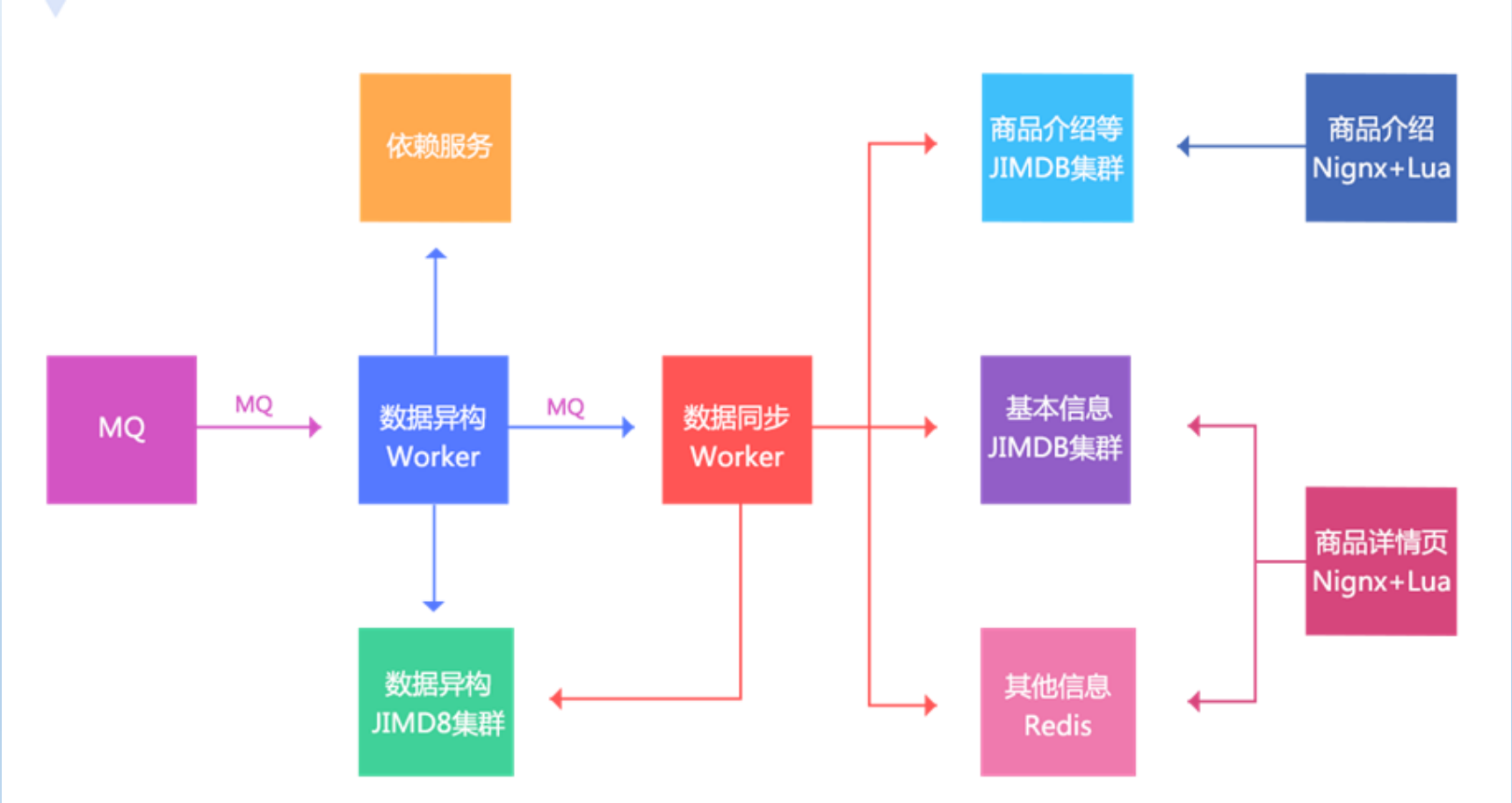
- 动静分离
- 缓存分层（两次校验）
- 读写分离（CQRS）
- 写基于时间分片削峰
- 异步化（Cache和DB）
- 数据本地性（本地cache）
- 热点数据实时移动
- 排队

jd商品详情页

2015.618当天PV数亿



jd商品详情页-架构3.0



jd商品详情页-架构3.0

1、数据变更通过**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），

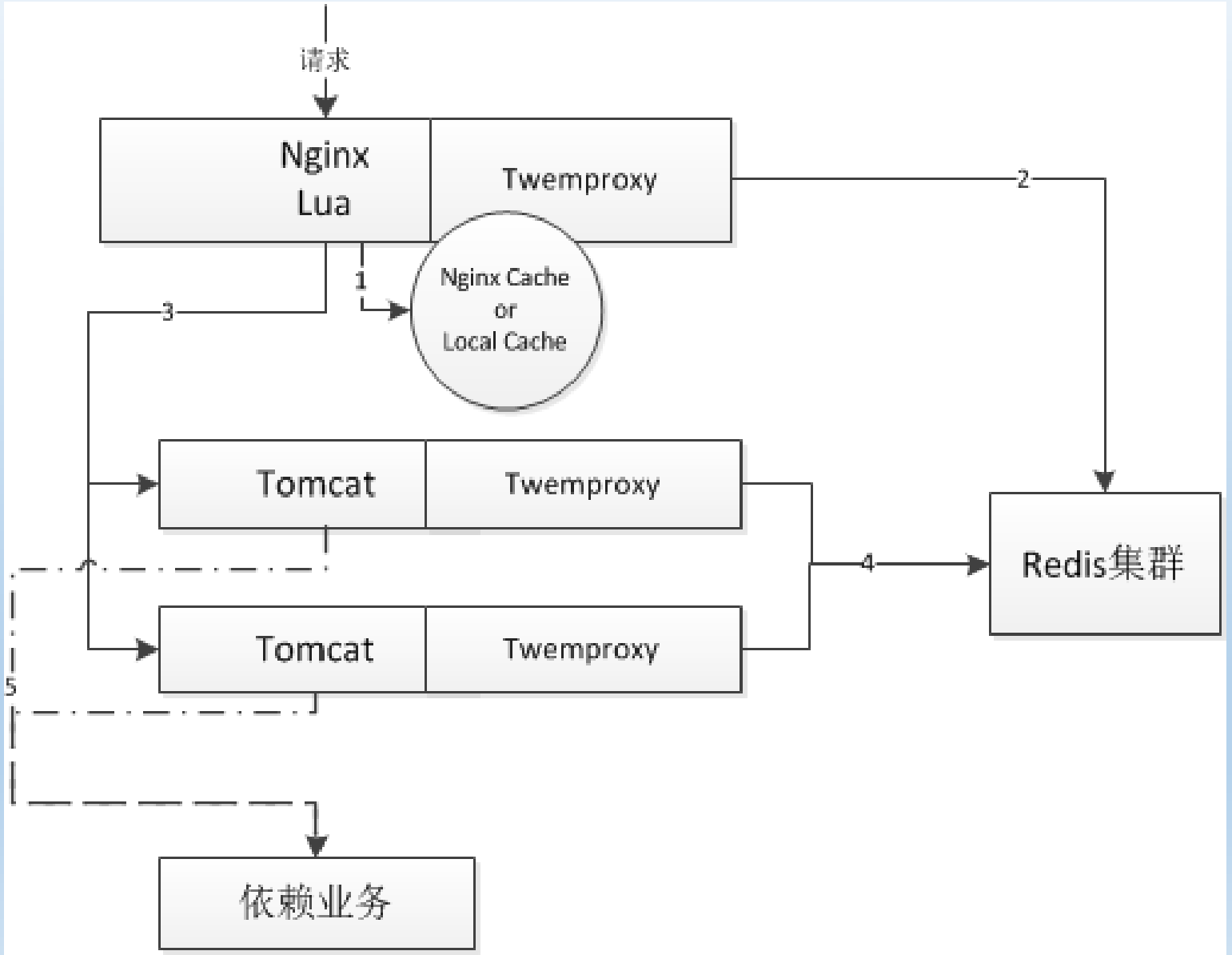
存储的数据都是**未加工的原子化数据**，如商品基本信息、商品扩展属性、商品其他一些相关信息、商品规格参数、分类、商家信息等；

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。

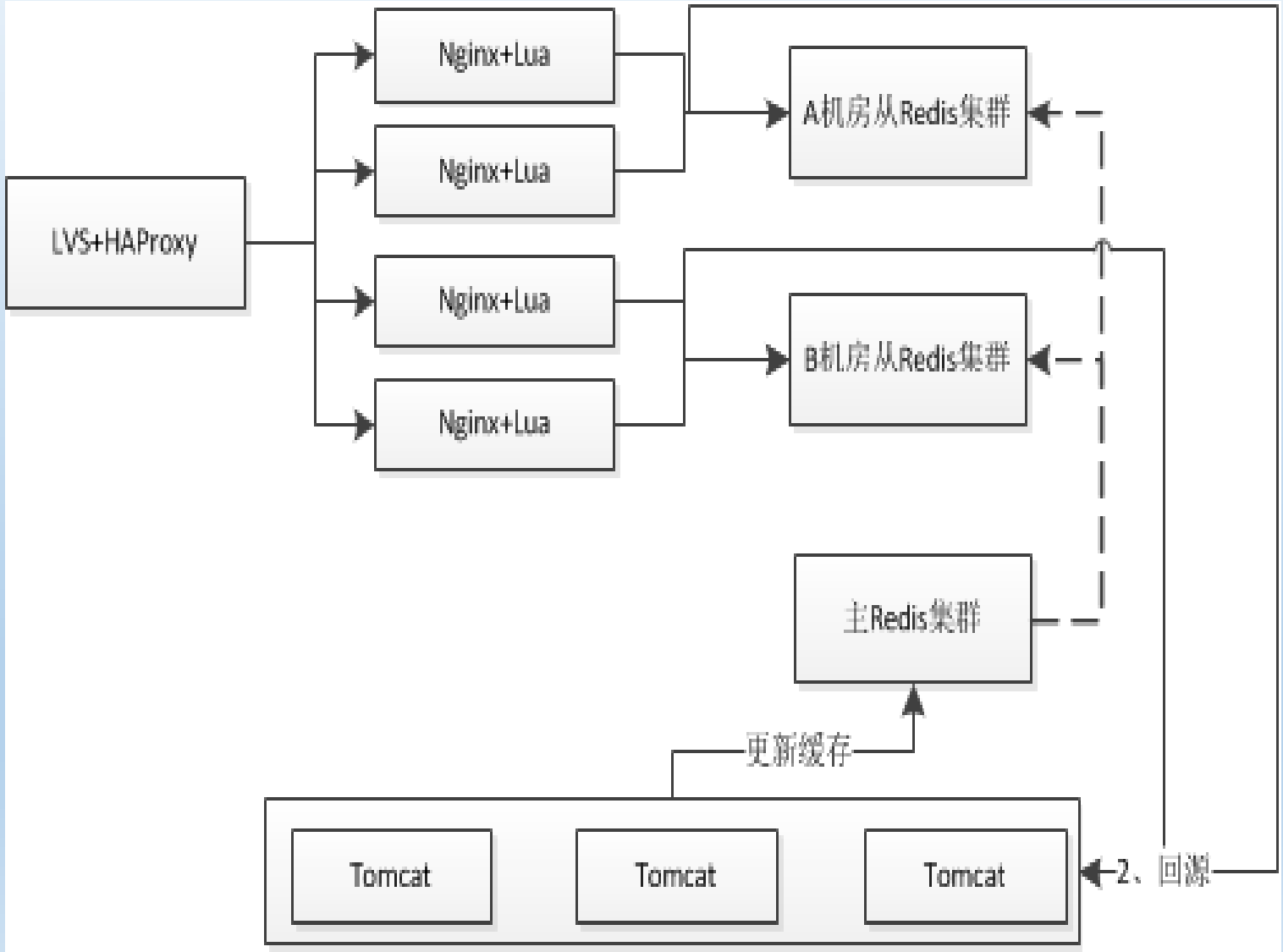
jd商品统一服务


请求
Nginx
Lua
Twemproxy
Nginx Cache
or
Local Cache
1
2
3
4
5
Tomcat
Twemproxy
Tomcat
Twemproxy
Redis集群
依赖业务

jd商品统一服务

- 1、请求首先进入Nginx，Nginx调用Lua进行一些前置逻辑处理，如果**前置逻辑不合法直接返回**；然后查询**本地缓存**，如果命中直接返回数据；
- 2、如果本地缓存不命中数据，则查询**分布式Redis集群**，如果命中数据，则直接返回；
- 3、如果分布式Redis集群不命中，则会调用**Tomcat进行回源**处理；然后把结果**异步写入Redis集群**，并返回。

jd商品统一服务


Nginx+Lua
Nginx+Lua
Nginx+Lua
Nginx+Lua
LVS+HAProxy
A机房从Redis集群
B机房从Redis集群
主Redis集群
更新缓存
Tomcat
Tomcat
Tomcat
2、回源

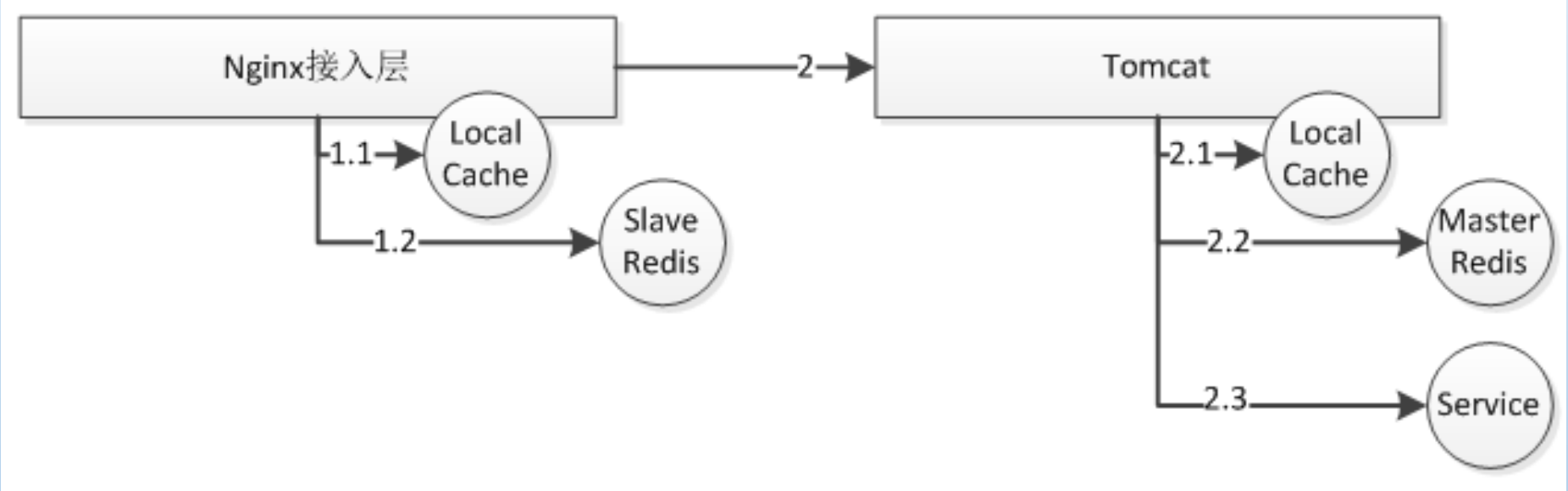
jd商品统一服务

- 数据会写一个**主集群**，然后通过**主从方式把数据复制到其他机房**，而**各个机房读自己的集群**；此处没有在各个机房做一套独立的集群来保证机房之间没有交叉访问，这样做的目的是保证数据一致性。

本地缓存

- 一致性哈希+本地缓存
- **如库存数据缓存5秒，平常命中率：本地缓存25%；分布式Redis28%；回源47%；　50%命中缓存**
- **一次普通秒杀活动命中率：本地缓存 58%；分布式Redis 15%；回源27%；　73%命中缓存**

多级缓存


Nginx接入层
2
Tomcat
1.1
Local Cache
1.2
Slave Redis
2.1
Local Cache
2.2
Master Redis
2.3
Service

多级缓存

1.1、首先在接入层，会使用**Nginx本地缓存**，这种前端缓存主要目的是**抗热点**；根据场景来设置缓存时间；

使用HttpLuaModule模块的shared dict做本地缓存

1.2、如果Nginx本地缓存不命中，接着会**读取各个机房的分布式从Redis缓存集群**，

该缓存主要是保存大量离散数据，**抗大规模离散请求**，比如使用一致性哈希来构建Redis集群，**即使其中的某台机器出问题，也不会出现雪崩的情况；**

减少访问回源率

2.1、**如果从Redis集群不命中，Nginx会回源到Tomcat**；

Tomcat首先读取本地堆缓存，这个主要用来支持在一个请求中多次读取一个数据或者该数据相关的数据；

而其他情况命中率是非常低的，或者缓存一些规模比较小但用的非常频繁的数据，如分类，品牌数据；

堆缓存时间我们设置为Redis缓存时间的一半。 防止相关缓存失效/崩溃之后的冲击。

2.2、如果Java堆缓存不命中，会读取主Redis集群，**正常情况该缓存命中率非常低，不到5%；**

读取该缓存的目的是防止前端缓存失效之后的大量请求的涌入，导致我们后端服务压力太大而雪崩；

我们默认开启了该缓存，虽然增加了几毫秒的响应时间，但是加厚了我们的防护盾，使服务更稳当可靠。

此处可以做下改善，比如我们设置一个阀值，超过这个**阀值**我们才读取主Redis集群，比如**Guava就有RateLimiter API**来实现。

可降级的多级读服务，比如只读本地缓存，只读分布式缓存，或者只读一个默认的降级数据

Nginx接入层

- **数据校验/过滤逻辑前置**

一种是用户无关的接口，另一种则是**用户相关的接口**；两种类型的域名c.3.cn/c0.3.cn/c1.3.cn和cd.jd.com；**请求cd.jd.com会带着用户cookie信息到服务端, 校验cookie信息**；

- **缓存前置:** 热点数据的**削峰**
- **业务逻辑前置**

在接入层直接实现了一些业务逻辑，当在高峰时出问题，可以在这一层做一些**逻辑升级**.

后端java重启慢，性能抖动。改完代码推送到服务器，**重启只需要秒级**，而且不存在抖动的问题。这些逻辑都是在**Lua**中完成。

- **降级开关前置**

防止降级后流量还无谓的打到后端应用。**总开关**是对整个服务降级，比如库存服务默认有货；而**原子开关**时整个服务中的其中一个小服务降级，比如库存服务中需要调用商家运费服务，如果只是商家运费服务出问题了，此时可以只降级商家运费服务

- **限流**

对于大多数请求按照IP请求数限流，对于登陆用户按照用户限流

- AB测试，灰度发布/流量切换，监控服务质量

限流

分类：

接入层限流

应用级限流

算法

简单：计数器

复杂：令牌桶算法(突发请求)，漏桶算法(平滑负载)

应用级限流

- **限制总并发数(资源) :**如数据库连接池、线程池， 信号量Semaphore
- **限制瞬时并发数 :**

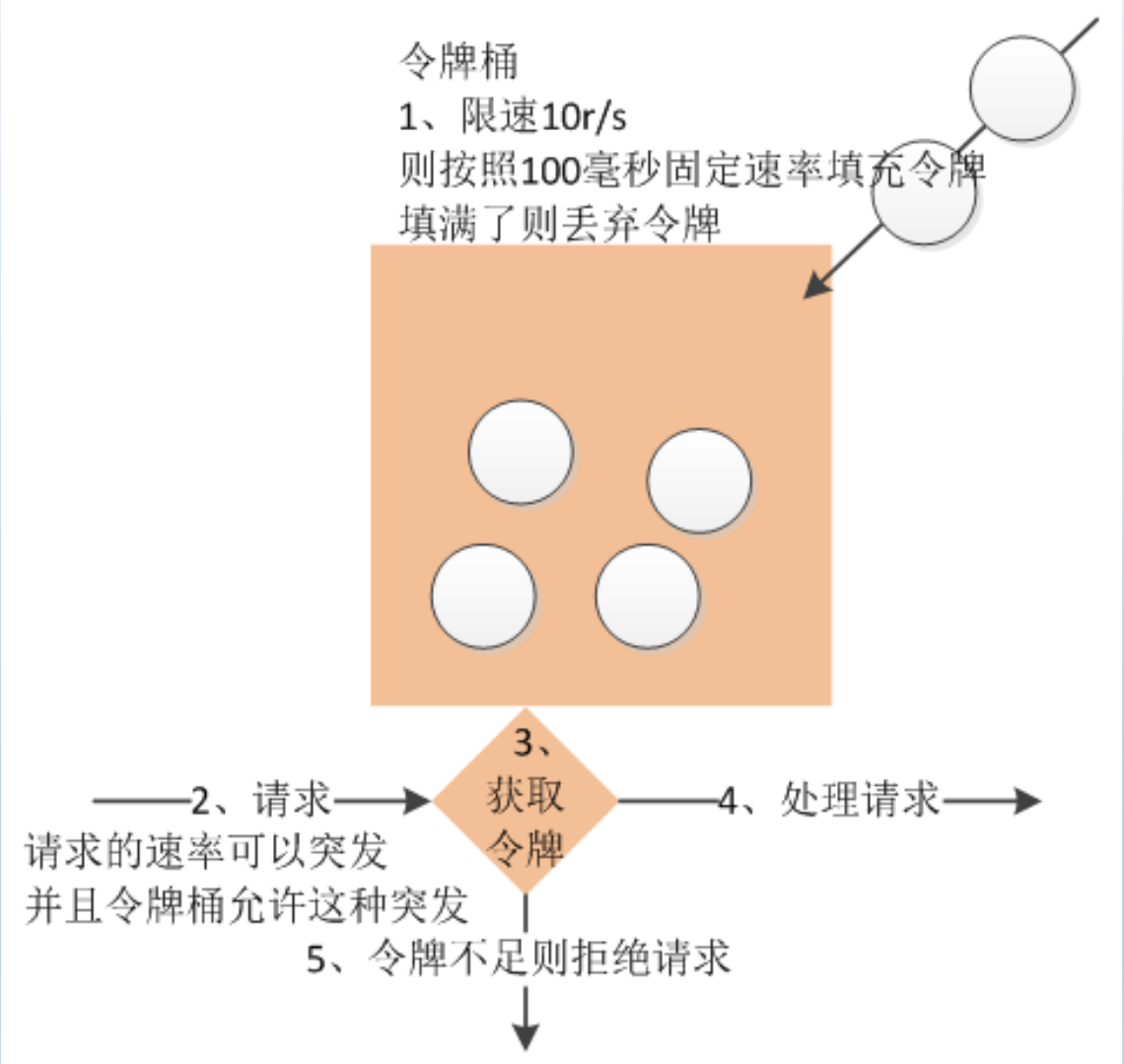
  如nginx的limit_conn模块，用来限制瞬时并发连接数

  如Tomcat Connector acceptCount， maxConnections， maxThreads
- **限制时间窗口内的平均速率:**

  如Guava的RateLimiter、nginx的limit_req模块，限制每秒的平均速率

限流算法

令牌桶算法


令牌桶
1、限速10r/s
则按照100毫秒固定速率填充令牌
填满了则丢弃令牌
2、请求
3、
获取
令牌
4、处理请求
请求的速率可以突发
并且令牌桶允许这种突发
5、令牌不足则拒绝请求

令牌桶算法

Guava RateLimiter　　平滑突发限流(SmoothBursty)　突发请求

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(10));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1));
```

将得到类似如下的输出：

```
1.997428

0.192273

0.200616
```

限流算法

- 漏桶算法

漏桶算法

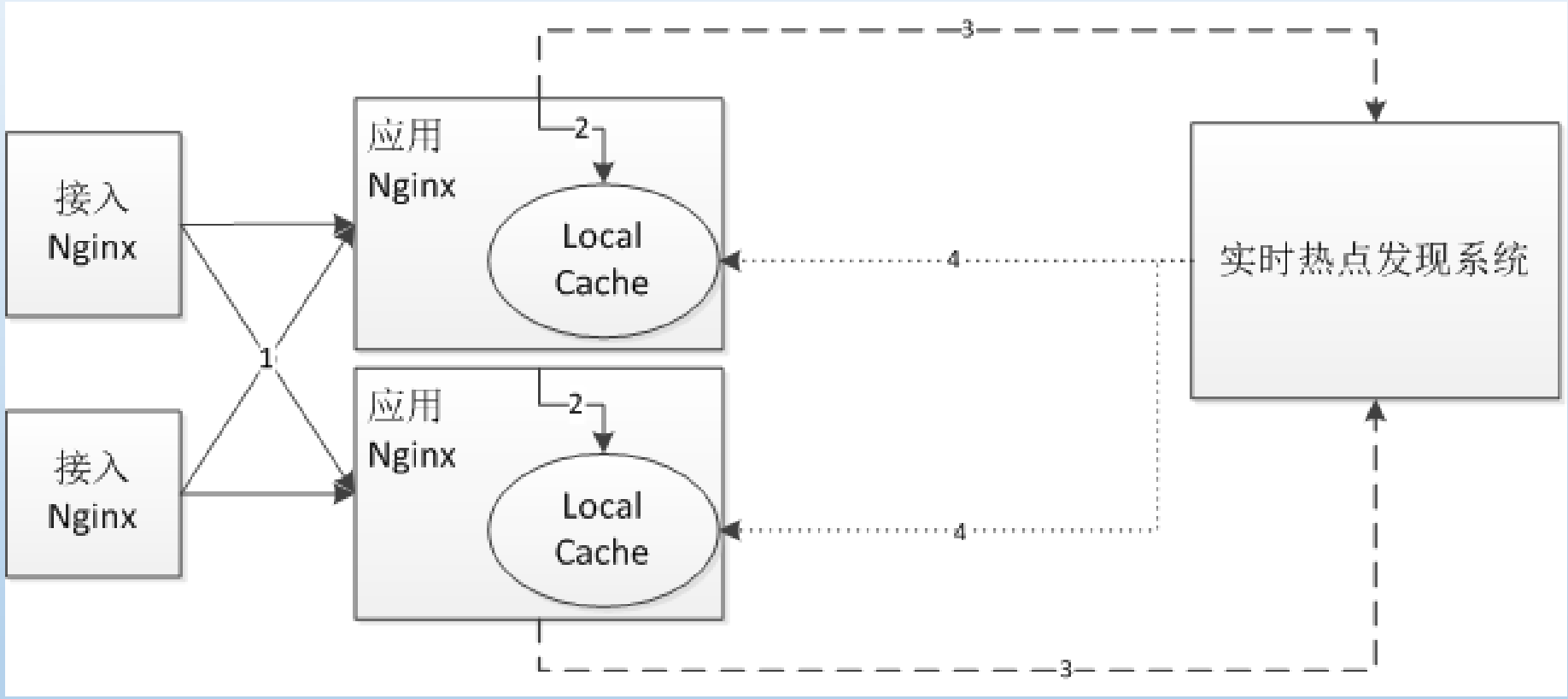
Guava RateLimiter　平滑预热限流(SmoothWarmingUp)　平滑负载

```
 RateLimiter limiter = RateLimiter.create(5, 1000, TimeUnit.MILLISECONDS);
for(int i = 1; i < 5;i++) {
   System.out.println(limiter.acquire());
}
```

将得到类似如下的输出：

```
0.0
0.51767
0.357814
0.219992
0.199984
```

实时热点发现系统



- 应用Nginx会将请求上报给实时热点发现系统，如使用UDP直接上报请求、或者将请求写到本地kafka、或者**使用flume订阅本地nginx日志**；上报给实时热点发现系统后，它将进行**统计热点（可以考虑storm实时计算）**.
- 根据设置的阀值将**热点数据推送到应用Nginx本地缓存**
- **负载低使用一致性hash，热点请求降级一直性hash为轮询**

服务隔离

- 应用内线程池隔离

  Servlet3异步化，并为不同的请求按照重要级别分配线程池，这些线程池是**相互隔离**的

- 部署/分组隔离

  不同的消费方提供不同的分组

- 拆应用隔离

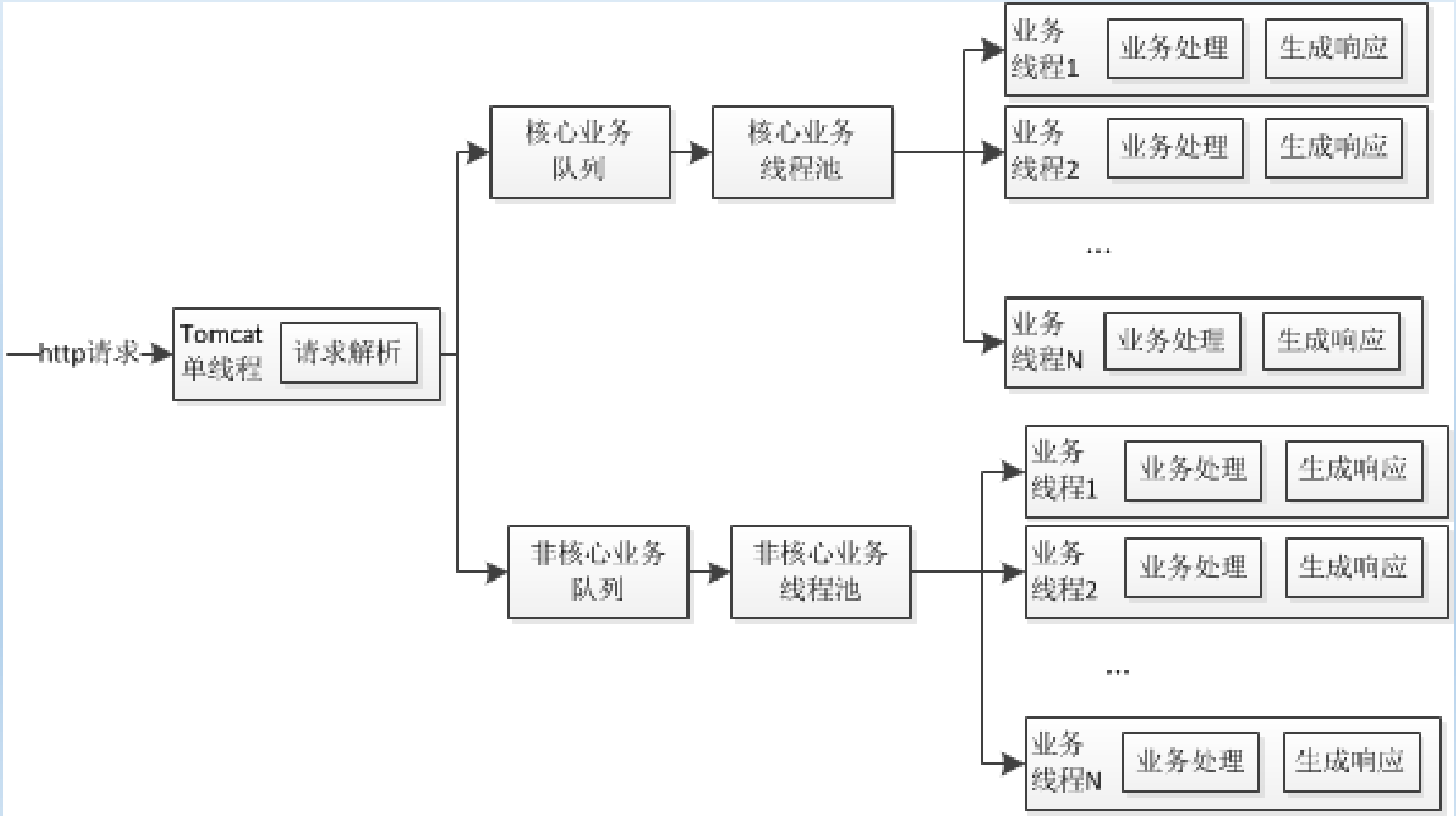
  如果一个服务调用量巨大，那我们便可以把这个服务单独拆出去

异步化 好处

- 更高的**并发能力**；
- 请求解析和业务处理线程池**分离**；
- 根据业务重要性对业务分级，并**分级线程池**；
- 对业务线程池进行监控、运维、降级等处理。

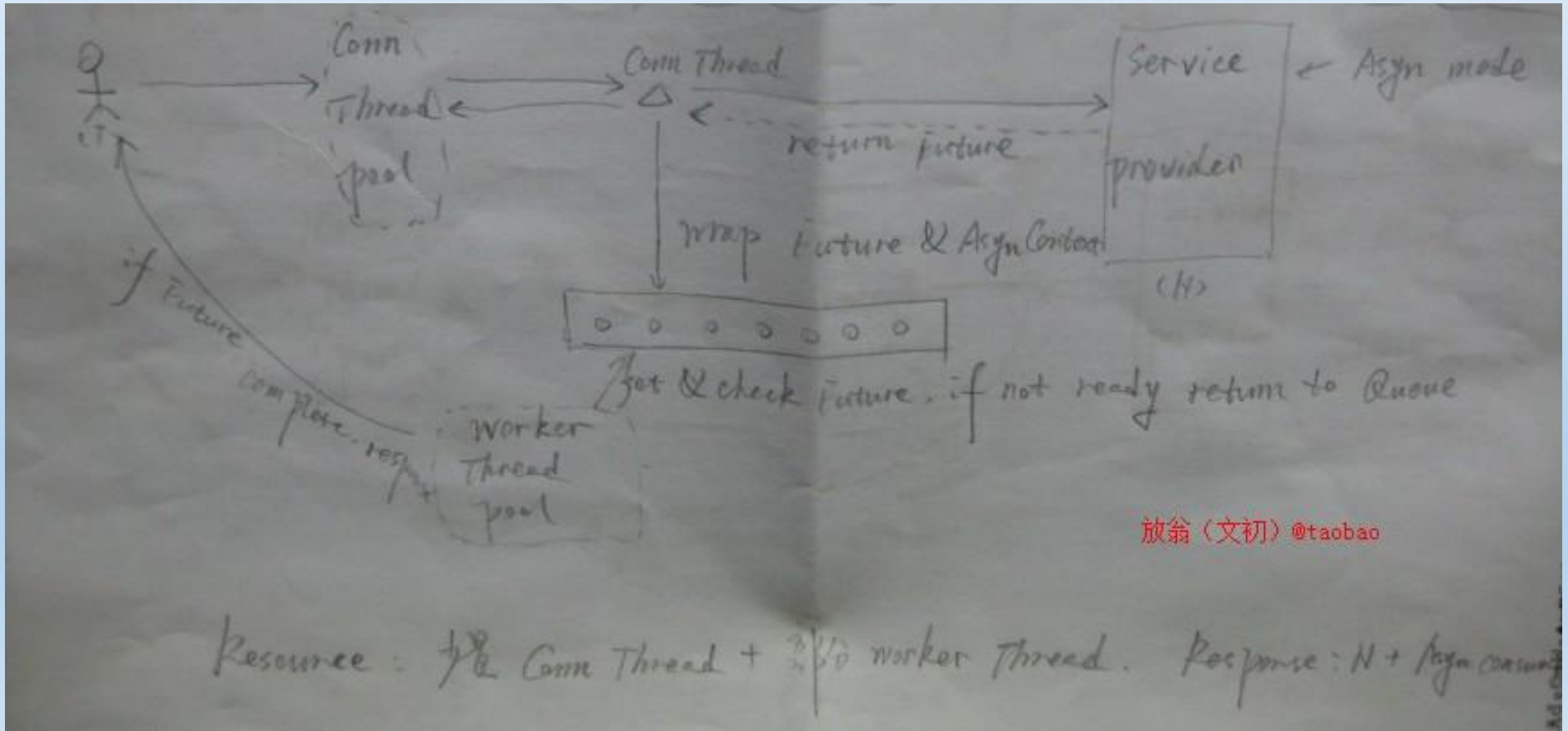
商品详情页系统

Servlet3异步化实践



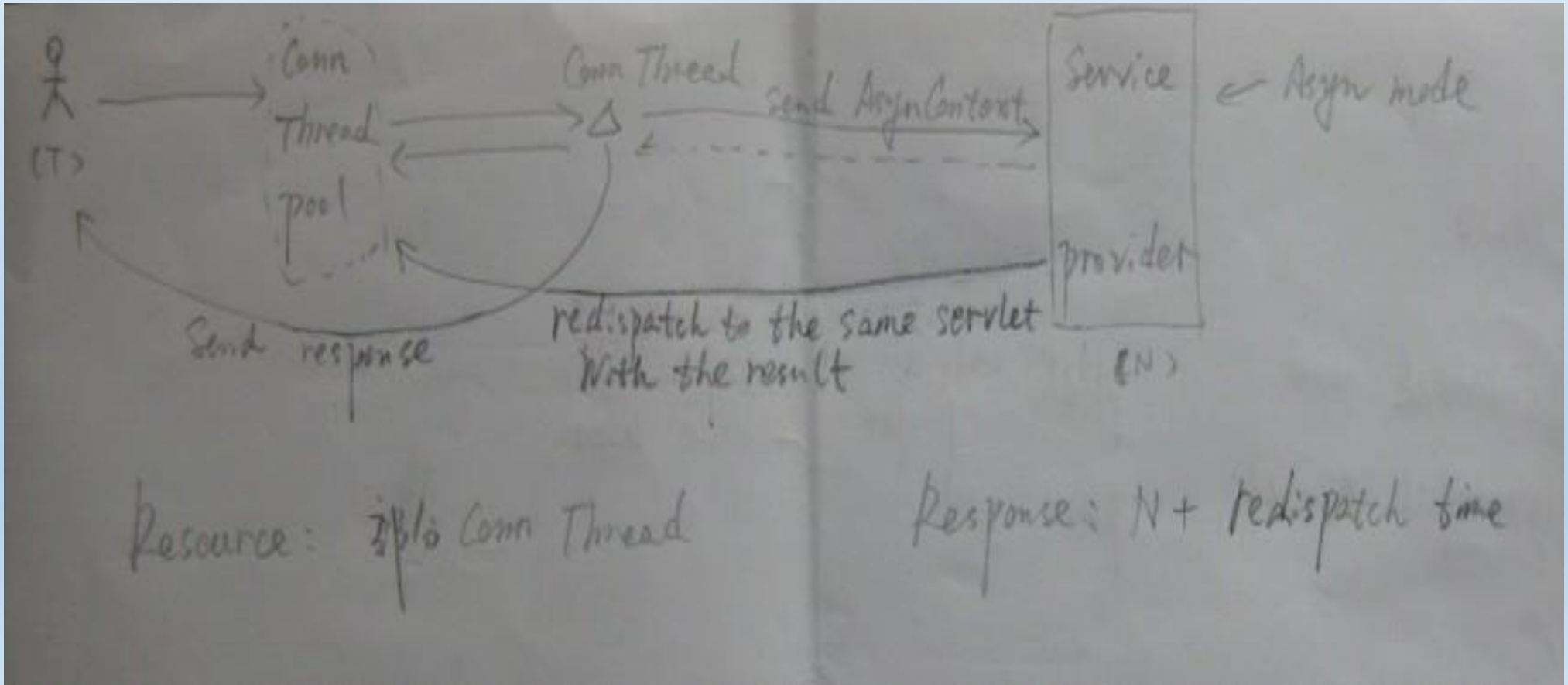
异步化 原型

异步化Web请求处理，后端服务提供者为非阻塞模式。（Pull & Complete mode） 实现：netty 异步调用



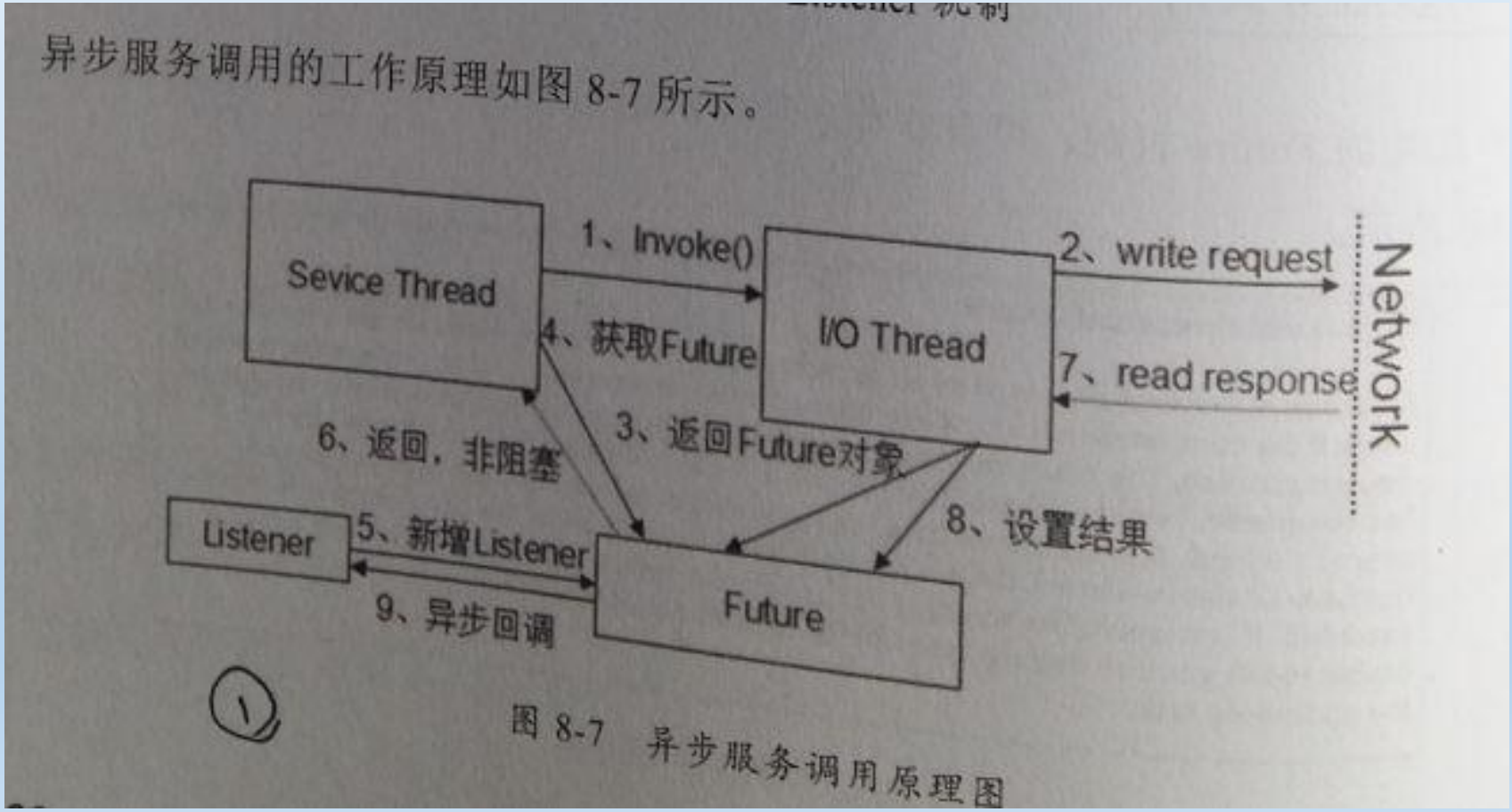
异步化 原型

- 异步化Web请求处理，后端服务提供者为非阻塞模式。（Push & resume mode） 实现： jetty continuations



异步化 Netty

Future listener 机制. Google Guava并发库 ListenableFuture：完成后触发回调的Future



Servlet 3.0 异步化

```
1.  public void submitFuture(final HttpServletRequest req, final Callable<
    k) {
2.      final String uri = req.getRequestURI();
3.      final Map<String, String[]> params = req.getParameterMap();
4.      final AsyncContext asyncContext = req.startAsync();  //开启异步上下文
5.      asyncContext.getRequest().setAttribute("uri", uri);
6.      asyncContext.getRequest().setAttribute("params", params);
7.      asyncContext.setTimeout(asyncTimeoutInSeconds * 1000);
8.      if(asyncListener != null) {
9.          asyncContext.addListener(asyncListener);
10.     }
11.     executor.submit(new CanceledCallable(asyncContext) { //提交任务给业务线程池
12.         @Override
13.         public Object call() throws Exception {
14.             Object o = task.call();  //业务处理调用
15.             if(o == null) {
16.                 callBack(asyncContext, o, uri, params);  //业务完成后，响应处理
17.             }
18.             if(o instanceof CompletableFuture) {
19.                 CompletableFuture<Object> future = (CompletableFuture<Object>)o;
20.                 future.thenAccept(resultObject -> callBack(asyncContext, resultObjec
    t, uri, params))
21.                 .exceptionally(e -> {
22.                     callBack(asyncContext, "", uri, params);
23.                     return null;
24.                 });
25.             } else if(o instanceof String) {
26.                 callBack(asyncContext, o, uri, params);
27.             }
28.             return null;
29.         }
30.     });
```

Servlet 3.0 异步化

```
1.  private void callBack(AsyncContext asyncContext, Object result, String uri, Map<Strin
    g, String[]> params) {
2.      HttpServletResponse resp = (HttpServletResponse) asyncContext.getResponse();
3.      try {
4.          if(result instanceof String) {
5.              write(resp, (String)result);
6.          } else {
7.              write(resp, JSONUtils.toJSON(result));
8.          }
9.      } catch (Throwable e) {
10.         resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR); //程序内部错误
11.         try {
12.             LOG.error("get info error, uri : {},  params : {}", uri, JSONUtils.toJSO
    N(params), e);
13.         } catch (Exception ex) {
14.         }
15.     } finally {
16.         asyncContext.complete();
17.     }
18. }
```

Servlet 3.0 异步化

```
        asyncListener = new AsyncListener() {
            @Override
            public void onComplete(AsyncEvent event) throws IOException {
            }

            @Override
            public void onTimeout(AsyncEvent event) throws IOException {
                AsyncContext asyncContext = event.getAsyncContext();
                try {
                    String uri = (String) asyncContext.getRequest().getAttribute("ur
i");
                    Map params = (Map) asyncContext.getRequest().getAttribute("param
s");
                    LOG.error("async request timeout, uri : {}, params : {}", uri, JSO
NUtils.toJSON(params));
                } catch (Exception e) {}
                try {
                    HttpServletResponse resp = (HttpServletResponse) asyncContext.getR
esponse();
                    resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
                } finally {
                    asyncContext.complete();
                }
            }

            @Override
            public void onError(AsyncEvent event) throws IOException {
                AsyncContext asyncContext = event.getAsyncContext();
                try {
                    String uri = (String) asyncContext.getRequest().getAttribute("ur
i");
                    Map params = (Map) asyncContext.getRequest().getAttribute("param
s");
```

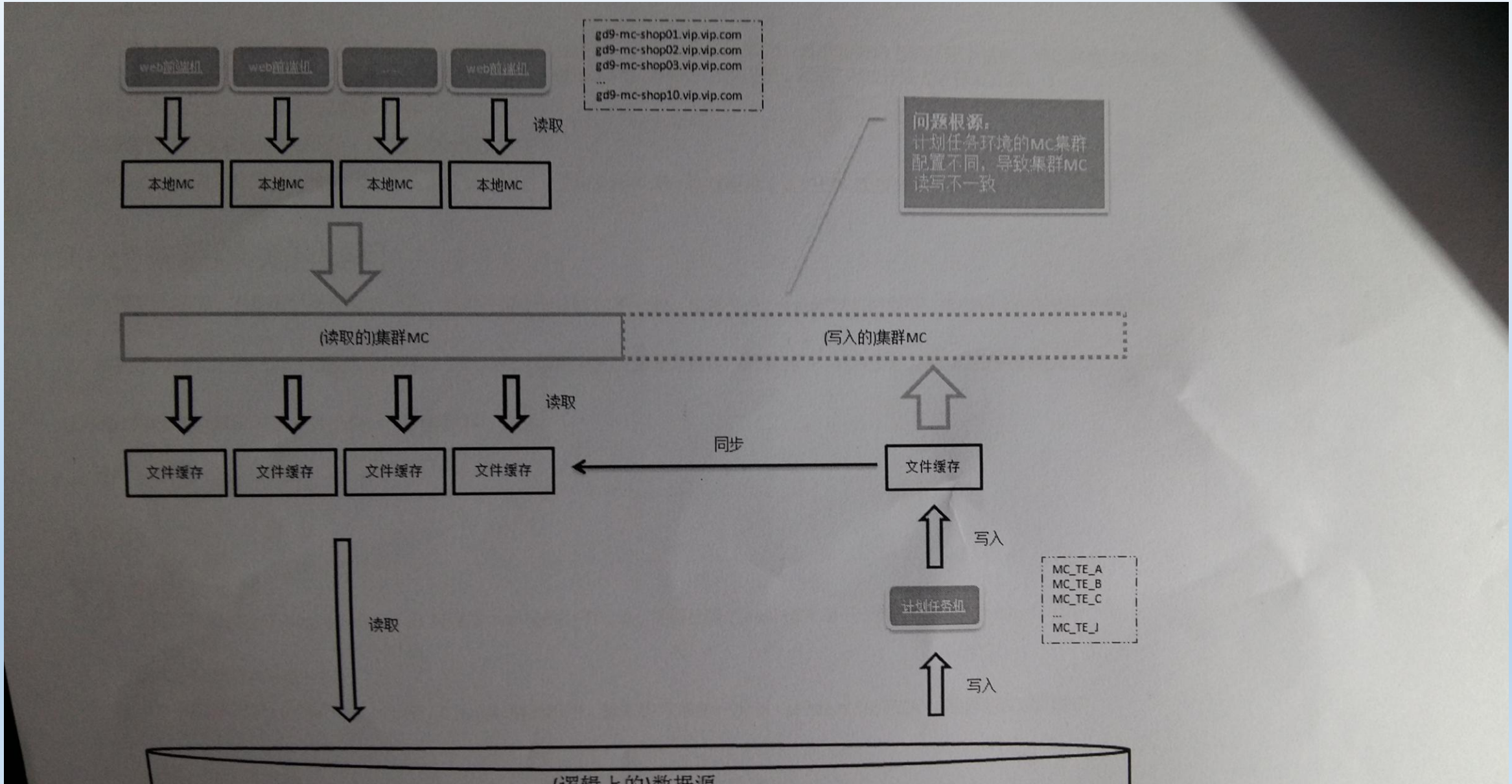
Servlet 3.0 异步化

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    res.setContentType("text/html;charset=UTF-8");
    res.setHeader("Cache-Control", "private");
    res.setHeader("Pragma", "no-cache");
    req.setCharacterEncoding("UTF-8");
    PrintWriter writer = res.getWriter();
    writer.println("Comet is a programming technique that enables " +
            "web servers to send data to the client without having any need for the client to request it.");
    writer.flush();

    final AsyncContext ac = req.startAsync();
    ac.setTimeout(60 * 1000);
    ac.addListener(new AsyncListener() {
        public void onComplete(AsyncEvent event) throws IOException {
            ASYNC_CONTEXT_QUEUE.remove(ac);
        }

        public void onTimeout(AsyncEvent event) throws IOException {
            ASYNC_CONTEXT_QUEUE.remove(ac);
        }

        public void onError(AsyncEvent event) throws IOException {
            ASYNC_CONTEXT_QUEUE.remove(ac);
        }

        public void onStartAsync(AsyncEvent event) throws IOException {
        }
    });
    ASYNC_CONTEXT_QUEUE.add(ac);
```

Jd商品详情页总结

- 限流
- 服务异步化
- 多级缓存
- 降级（读缓存降级）
- 前置(开关,业务逻辑,缓存)
- 数据本地性（本地cache）
- 热点系统

VIP商品详情页系统


gd9-mc-shop01.vip.vip.com
gd9-mc-shop02.vip.vip.com
gd9-mc-shop03.vip.vip.com
...
gd9-mc-shop10.vip.vip.com
问题根源：
计划任务环境的MC集群配置不同，导致集群MC读写不一致
本地MC
本地MC
本地MC
本地MC
读取
(读取的)集群MC
(写入的)集群MC
读取
同步
文件缓存
文件缓存
文件缓存
文件缓存
文件缓存
写入
计划任务组
MC_TE_A
MC_TE_B
MC_TE_C
...
MC_TE_J
写入
读取

- Q & A