高访问量系统的静态化架构设计

许令波/君山

2013/8/14

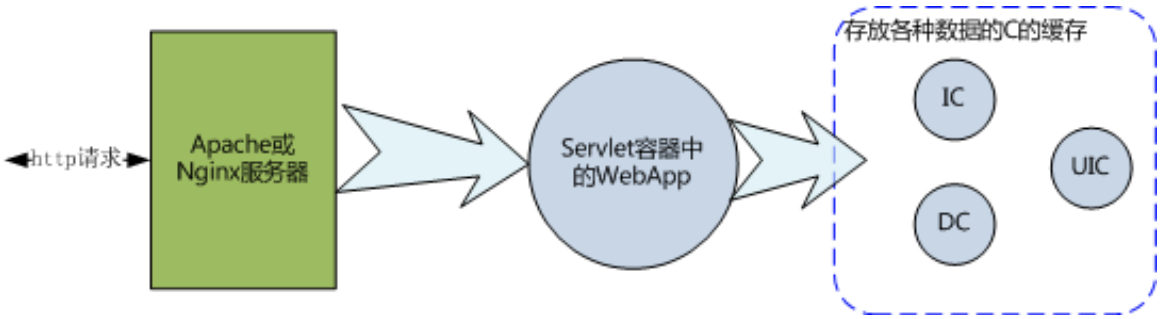
何谓高访问量系统？以 Java 系统为例，正常的用户请求要支撑 20w/s 的 QPS，对一个有复杂业务的动态系统来说是个不小的挑战。下面就以商品详情系统（Detail 系统）为例介绍下高访问量系统的静态化架构设计方案。

淘宝高访问量商品详情系统简介

根据 Alexa 全球排名淘宝目前排名 13 为，日均 PV 约有 25 亿，日均独立 IP 访问约有 1.5 亿，这其中 item.taobao.com 域名对应的 Detail 系统约占比总 PV 的 25%左右。可以说 Detail 系统是目前淘宝中单系统访问量最高的系统，当前每秒约有 20k 的请求到达我们的后端服务器。下面简单介绍一下这个高访问量系统的基本情况：

页面大小 45k，压缩后 15k，峰值带宽可达 2Gbps。服务端页面平均 RT 约为 15ms。

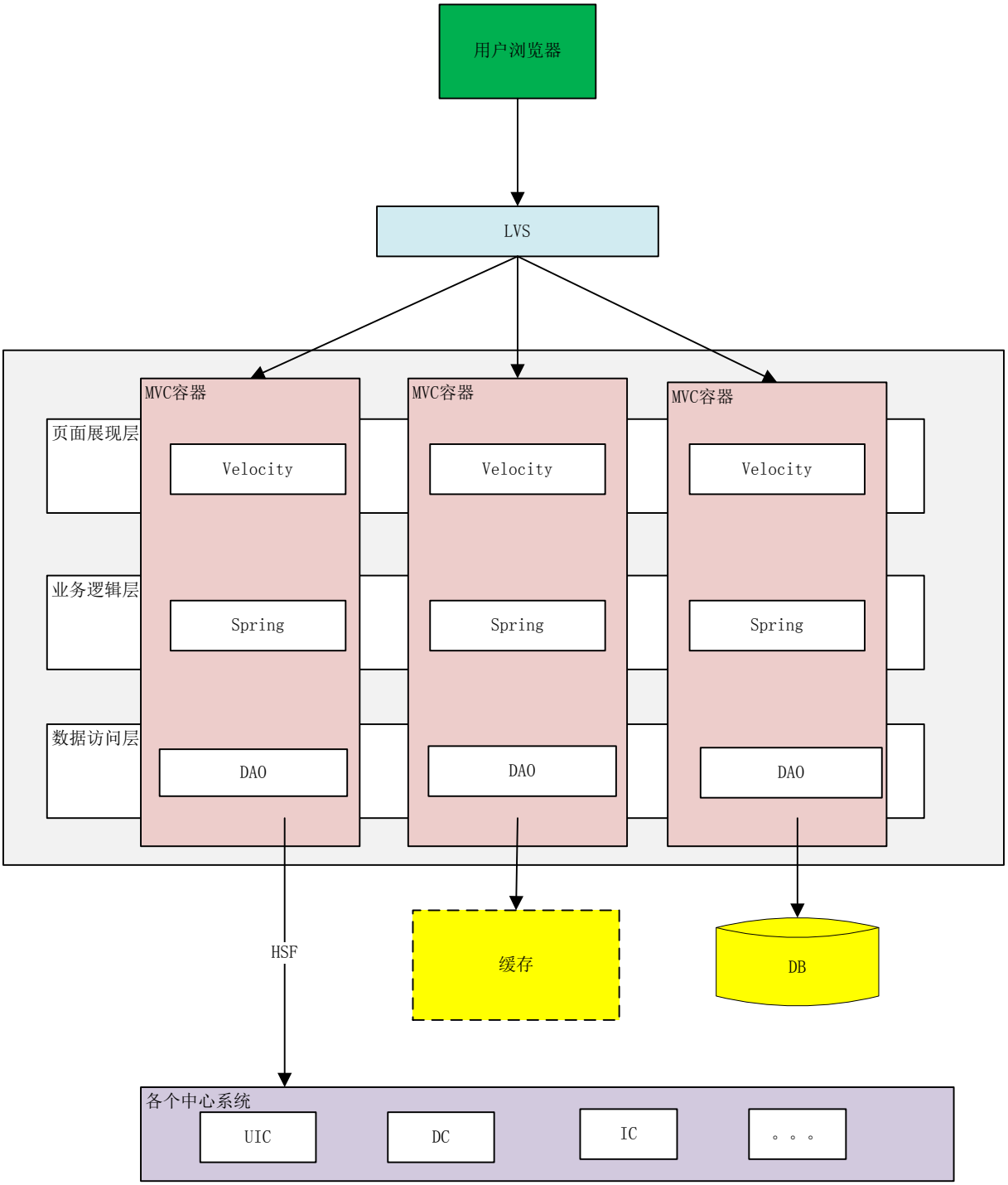
在静态化改造之前，淘宝大多的前台系统多是如下结构：



前台系统的基本结构

前面的 HTTP 请求经过负载均衡设备分配到某个域名对应的应用集群，经过 Nginx 代理到 Jboss 或者 Tomcat 容器，由他们负责具体处理用户请求。目前像这些高访问量的系统大部分需要读取的数据都已经直接走 K/V 缓存了，不会直接从 DB 获取数据。还有一部分应用逻辑会是走远程的系统调用，淘宝有一套高性能的分布式服务框架（HSF 框架）来提供系统之间的服务调用。

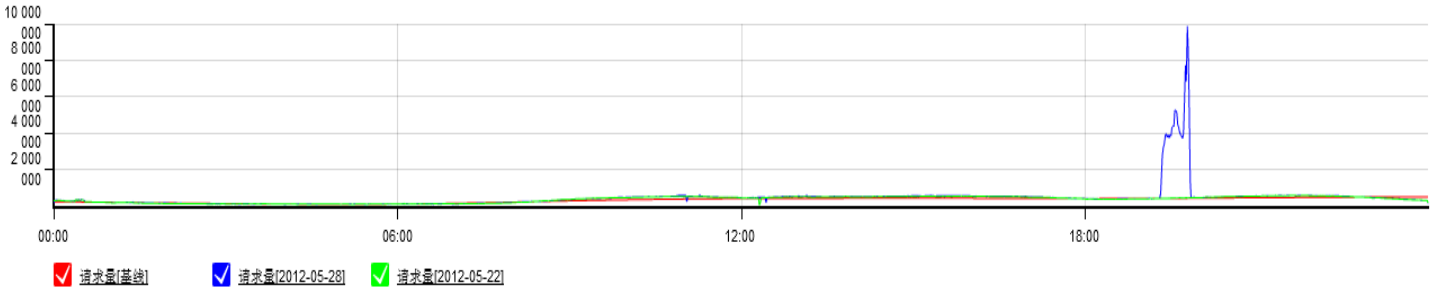
还有淘宝的前台系统都是基于 Java 的 MVC 框架开发的（WebX 框架），一个典型的系统结构图如下：


用户浏览器
LVS
MVC容器
MVC容器
MVC容器
页面展现层
业务逻辑层
数据访问层
Velocity
Velocity
Velocity
Spring
Spring
Spring
DAO
DAO
DAO
HSF
缓存
DB
各个中心系统
UIC
DC
IC
。。。

前台系统结构

系统存在哪些挑战

随着淘宝网站的发展壮大，系统也面临越来越多的的挑战，有些是业务发展带来的挑战如双 11/双 12 的大型促销活动和秒杀活动等突发流量冲击；还有一些是非正常的访问请求，如网站经常受到攻击和恶意请求。这些流量有些是可预测的有些是不可预测和防范的。如下图所示是一次访问的流量图：



非正常访问流量图

像这种流量突然暴增的情况对系统的冲击很大，有时候瞬间流量可达到 20 万的 QPS，所以如何让系统有更好的性能和稳定性是我们面临的一大挑战。

淘宝前台系统的优化历程

在介绍前台系统的静态化改造之前，我们先回顾一下我们已经做了哪些优化工作，这些工作对我们后续的优化也至关重要。淘宝从当初一个很小的系统发展成现在这么高访问量的系统，这中间也经历了很多次的系统迭代升级。

- 2009 系统拆分、静态文件合并、前端页面异步化和 JSON 化
- 2010 去 DB 依赖、引入缓存、提升单机 QPS、 关注用户体验
- 2011 优化进入深水区 Velocity、BigPipe
- 2012 静态化改造
- 2013 统一 Cache、CDN 化、网络协议

高访问量系统的静态改造

1）什么是静态化系统

在要改造成静态化系统之前首先要搞明白什么是静态化系统，他有那些属性？我们要改造要满足这些基本的属性才能有目标的去改造。

一般静态系统通常有如下几方面特征：

- 一个页面对应的 URL 通常固定。不同的 URL 表示不同的内容，让返回的请求和 URL 相关，也就是通过 URL 能唯一标识一个页面。
- 页面中不包含浏览者相关因素。页面中的不能包含与浏览者相关的因素，这里所说的不能包含不包括 JS 动态生成的部分，也就是页面中 HTML 代码不能显示的含有浏览器相关的 DOM。如不能含有用户的姓名、身份

标识以及 cookie 相关的因素等。

- 页面中不包含的时间因素。页面同样不能含有时间（这里的时间不是指客户端浏览器中获取的时间，而是服务器端输出的时间）相关的因素，不能随着时间的变化，页面中的 DOM 结构会变化。典型的如淘宝的秒杀中，到某个时间点时页面中的立即购买按钮就可以使用，这个判断的时间就是从服务器端获取的时间。
- 页面中不包含地域因素。页面中的地域因素很好理解，即北京访问的页面要和上海访问的页面相同。淘宝上也有个例子，就是宝贝的运费，不同地区的运费可能不一样。那要做成静态化，这个运费就不能直接的反映在 HTML 代码中。
- 不能包含 Cookie 等私有数据。Cookie 实际上主要是用来标识访问量信息的一个工具，如果页面中包含这些私有数据，也不可能不包含上面这些因素了。所以要满足静态化，就不能包含 Cookie 信息。

这里再强调一下，所谓静态化不仅是传统意义上完全的存在磁盘上 HTML 页面，它可能经过 Java 系统产生的页面但是它输出的页面本身不包含上面所说的那些因素，还有一点页面中不包含指的是页面的 HTML 源码不含有，这一点务必要清楚。

2）为什么要进行静态化架构设计

我们前面分析了我们的系统面临的各种挑战，这些挑战都涉及到性能优化，那性能优化为什么要进行静态化这种架构设计呢？

其实从前面的系统的优化历程来看，我们的系统经过多次优化升级，包括系统架构的升级、系统本身的模块优化、代码优化和增加各种缓存等这些优化，我们优化层次都是在 Java 系统中做改进。我们改进的思路多是尽量让应用本身怎么更快的获取数据，怎么更快的计算出结果，然后把结果返回给用户。然而我们测试了一种极端的情况就是将系统所有的数据全部缓存，然后将所有的请求结果直接返回。在这种情况下压测我们的 Java 系统，性能仍然不能满足我们的期望，而我们的目标是要再上一个数量级，达到 2k 甚至上万的 QPS，所以在 Java 系统上不可能达到这个目标。

所以我们判断 Java 系统本身已经达到了瓶颈， 因为 Java 系统本身也有其弱点如不擅长处理大量连接请求，每个连接消耗的内存较多，Servlet 容器解析 Http 协议较慢等。所以我们必须要跳出 Java 系统来做，如何跳出 Java 系统？就是让请求尽量不经过 Java 系统，在前面的 Web 服务器层就直接返回。这种模式自然就想到了静态化这种架构，所以静态化就成了我们的必然选择。

那么系统静态化为何能做的 Java 系统做不到的高性能呢？静态化有如下优点：

- 改变了缓存方式。直接缓存 HTTP 连接而不是仅仅缓存数据，Web 代理服务器根据请求 URL 直接取出对应的 HTTP 响应头和响应体直接返回，这个响应连 HTTP 协议都不用重新组装，同样 HTTP 请求头也不一定需要解析，所以做到了获取数据最快。
- 改变了缓存的地方。不是在 Java 层面做缓存而是直接在 Web 服务器层上做，所以屏蔽了 Java 层面的一些弱点，而 Web 服务器（如 Nginx、Apache、Varnish）都擅长处理大并发的静态文件请求。

3）动态系统如何改造

有了目标有了方向，接下去就是如何来改造我们的系统，我们的动态页面如何改造成适合缓存的静态页面呢？改造的方法就是前面所说的要去除那几个影响因素。

如何去掉这些影响因素？解决办法就是将这些因素单独分离出来，也就是做动静分离。下面以 Detail 系统为例介绍如何做动静分离。

动静分离

- URL 唯一化。Detail 系统天然的就可以做到 URL 统一化，如每个商品都有 ID 来标识，那么 http://item.taobao.com/item.htm?id=xxxx 就可以作为唯一的 URL 标识。
- 分离浏览者相关的因素。浏览者相关的包括是否登录以及登录身份等因素，我们可以单独拆分出来，通过动态请求来获取。
- 分离时间因素。服务端输出的时间也通过动态请求获取。
- 异步化地域因素。Detail 上与地域相关的做成异步方式获取。
- 去掉 Cookie。服务端输出的页面包含的 Cookie 可以通过代码软件来删除，如 Varnish 可以通过 unset req.http.cookie 命令去掉 Cookie。

动态内容结构化

分离出动态内容后，如何组织这些内容页非常关键，应该这些动态内容会被页面中其他模块用到，如判断该用户是否登录、用户 ID 等。这些信息 JSON 化方便前端很容易取到，如下图：

```
<script>(function() {
    g config.vdata = {
        "viewer": {//浏览者相关
            "cc": false,
            "tnik": "君山"
            ...},
        "vtgs": {
            "tg": "524288",
             ...},
        "sys": {//系统相关
            ....},
            "now": 1376561863065,
            "tkn": "VrxuqfFuSm",
            "p": 1.0}
    };})();
</script>
```

动态数据结构

对 Detail 系统来说，虽然商品信息是可以缓存的，不需要动态获取，但是也可以将商品的关键信息结构下：

```
<script> (function() {
      g config.idata = {
         item: {//商品信息
            id: 17804510436,
            status: 0,
                ...},
         seller: {//卖家信息
            status: 0},
         shop: {//店铺信息
            id: "102447199",
            xshop: true}
      }
   })(); < /script>
```

商品数据结构

如何组装动态内容

知道了分离那些内容，又知道怎么组织它们，现在的文件就是如何获取它们了，并和静态文件组装在一次了。获取动

态内容通常有两种方式：ESI（Edge Side Includes）和 CSI（Client Side Include）。

- ESI。即在 Web 代理服务器上做动态内容请求，并将请求插入到静态页面中，当用户拿到页面时已经是一个完整的页面了，如现在的 Detail 就是采用这种方式。这种方式对服务端性能有些影响，但是对用户体验较好。
- CSI。这种方式就是在发起一个异步 JS 请求单独向服务端获取动态内容。这种方式服务端性能更加，但是用户端页面有些延时提交稍差。

4）几种静态化方案的设计及如何选择

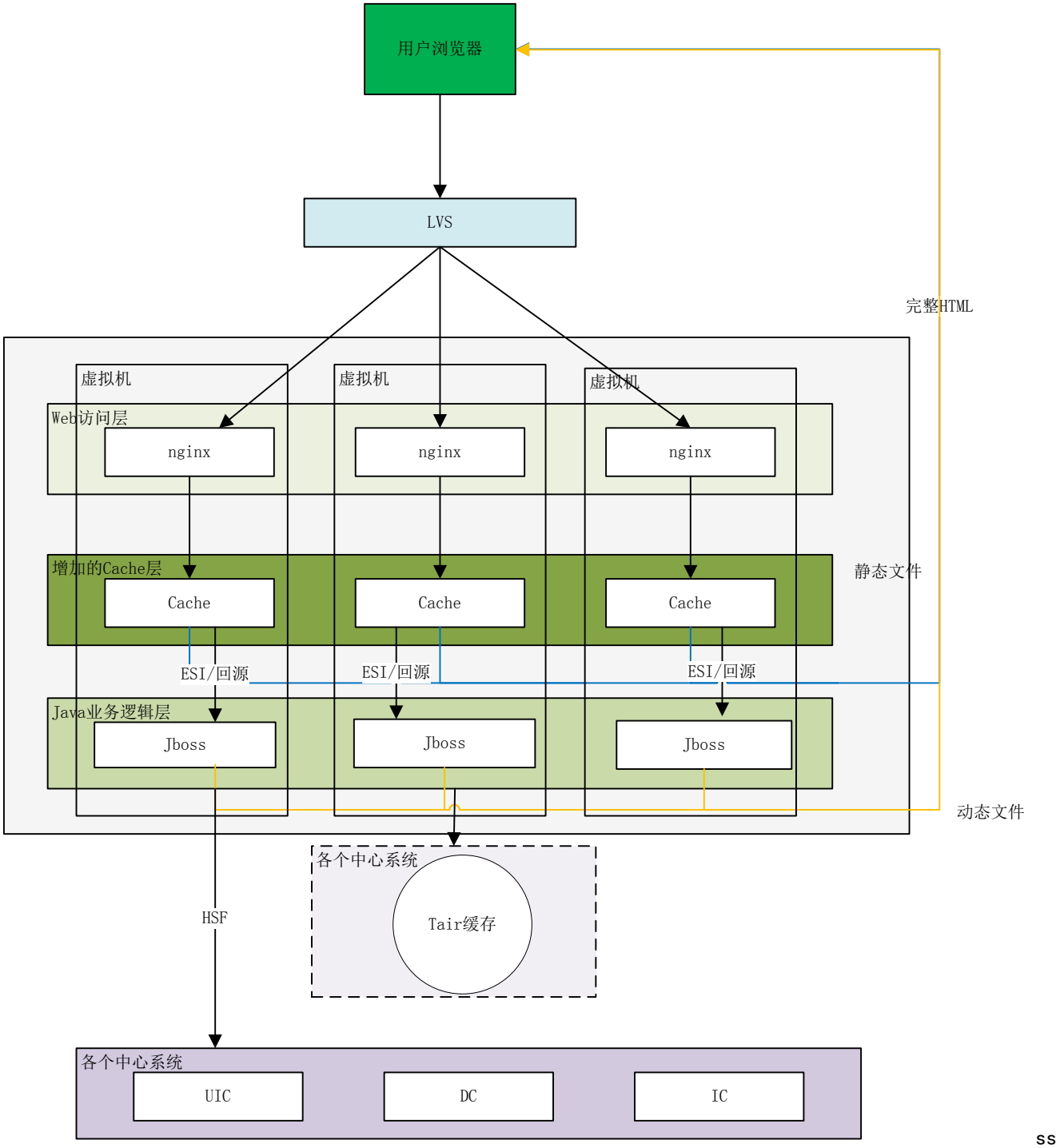
下面详细分析一下静态化架构如何设计，首先要考虑下方案的应该遵循的几个原则，这其中要设计到如下几个问题该如何回答：

- 是否一致性 Hash 分组？做缓存一定是和命中率紧密相关的，而命中率就和数据的集中度相关，而要让数据集中一致性 Hash 就是一个必然选择。但是一致性 Hash 有一个天然的缺陷就是导致热点问题，当热点特别集中时可能会导致网络瓶颈。
- 是否使用 ESI？ESI 和 CSI 的利弊前面已经分析过，ESI 对性能有影响但是对客户端友好，对前端编程也方便。
- 是否使用物理机？物理机可以提供更大的内存，更好的 CPU 资源，但是使用物理机也有一些缺点，如会导致应用集群的相对集中，进而导致网络风险增加。另外对 Java 系统而言内存增加并不能带来那么大的好处。
- 谁来压缩?在哪里压缩问题也是一个比较纠结的问题，增加一层 Cache，必然增加了数据的传输，那么谁来压缩就会影响到 Cache 的容量和网络数据的传输量。
- 网卡选择？网卡选择其实是个成本问题，避免网络瓶颈可以选择万兆网卡和交换机，但是必然带来成本增加。

根据这几个方面的考虑，分别产出如下几个方案：

方案 1. 采用 Nginx+Cache+Java 结构虚拟机单机部署

这种部署结构图如下：



ss

Nginx+Cache+Java 结构虚拟机单机部署

这个方式是最简单的静态化方案，只需要在当前的架构上加一层 Cache 层就行了，网络结构和业务逻辑都不用变化，只需要将系统做静态化改造就完成。但是它也有各自优缺点：

优点：

- 没有网络瓶颈，不需要改造网络
- 机器增加，也没有网卡瓶颈
- 机器数增多故障风险减少

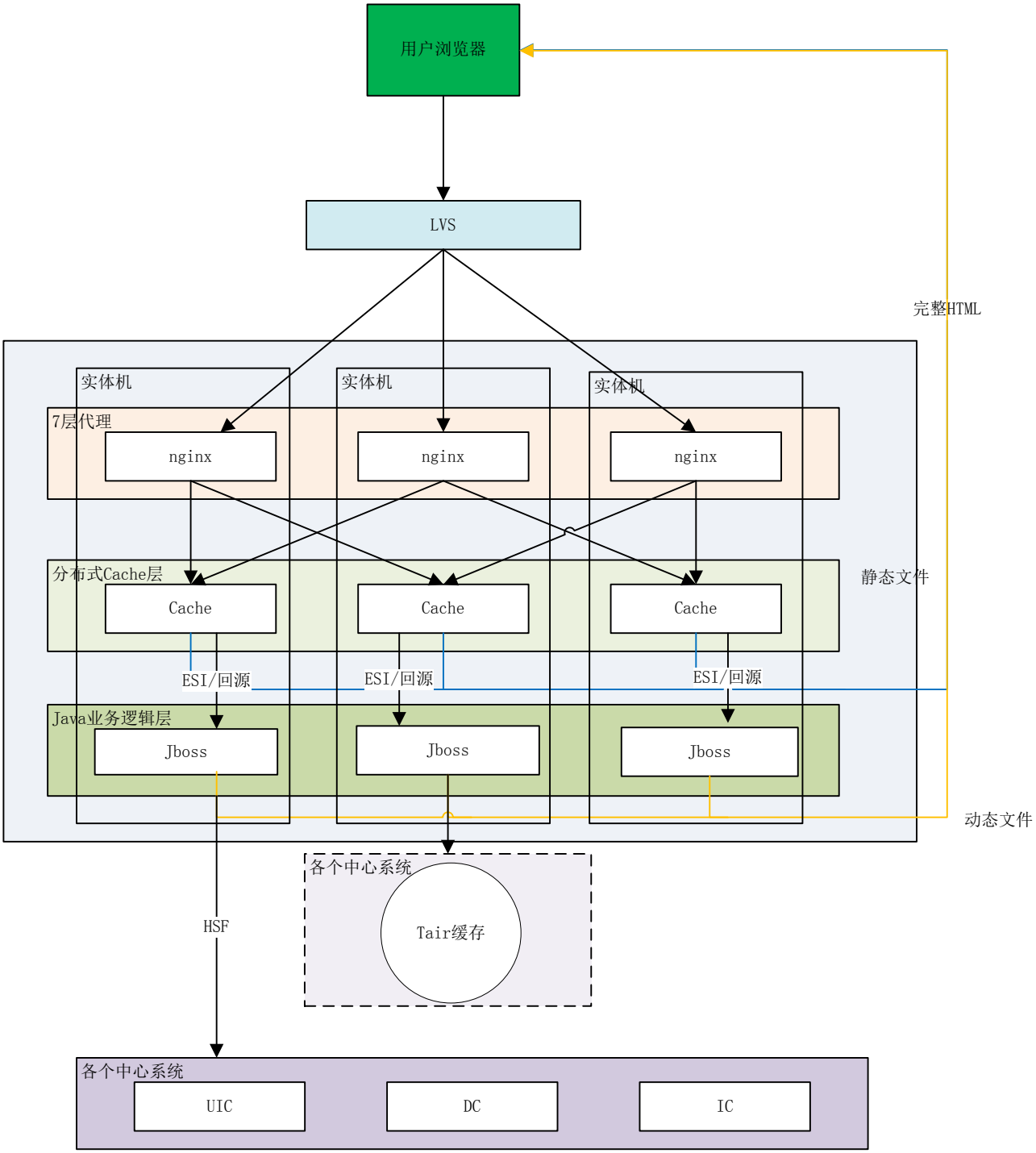
缺点：

- 机器增加，缓存命中率下降
- 缓存分散，失效难度增加 ·
- Cache 和 Jboss 都会争抢内存

该方案虽然比较简单，但也是能够解决热点商品的访问问题，如做大促时，商品数比较少，在有限内存中仍然能够命中这些商品；另外针对一些恶意攻击也十分有效，这个时候的命中率能达到 90%以上。但是对系统的整体性能不能有很多的提升。

方案 2. 采用 Nginx+Cache+Java 结构实体机单机部署

该方案部署结构图如下：



Nginx+Cache+Java 结构实体机单机部署

这种方案是在前面的基础上将虚拟机改成实体机增大 Cache 的内存，并且采用了一致性 Hash 分组的方式来提升命中率，这里采用分组将 Cache 分成若干组，这样可以达到命中率和访问热点的平衡。它的优点如下：

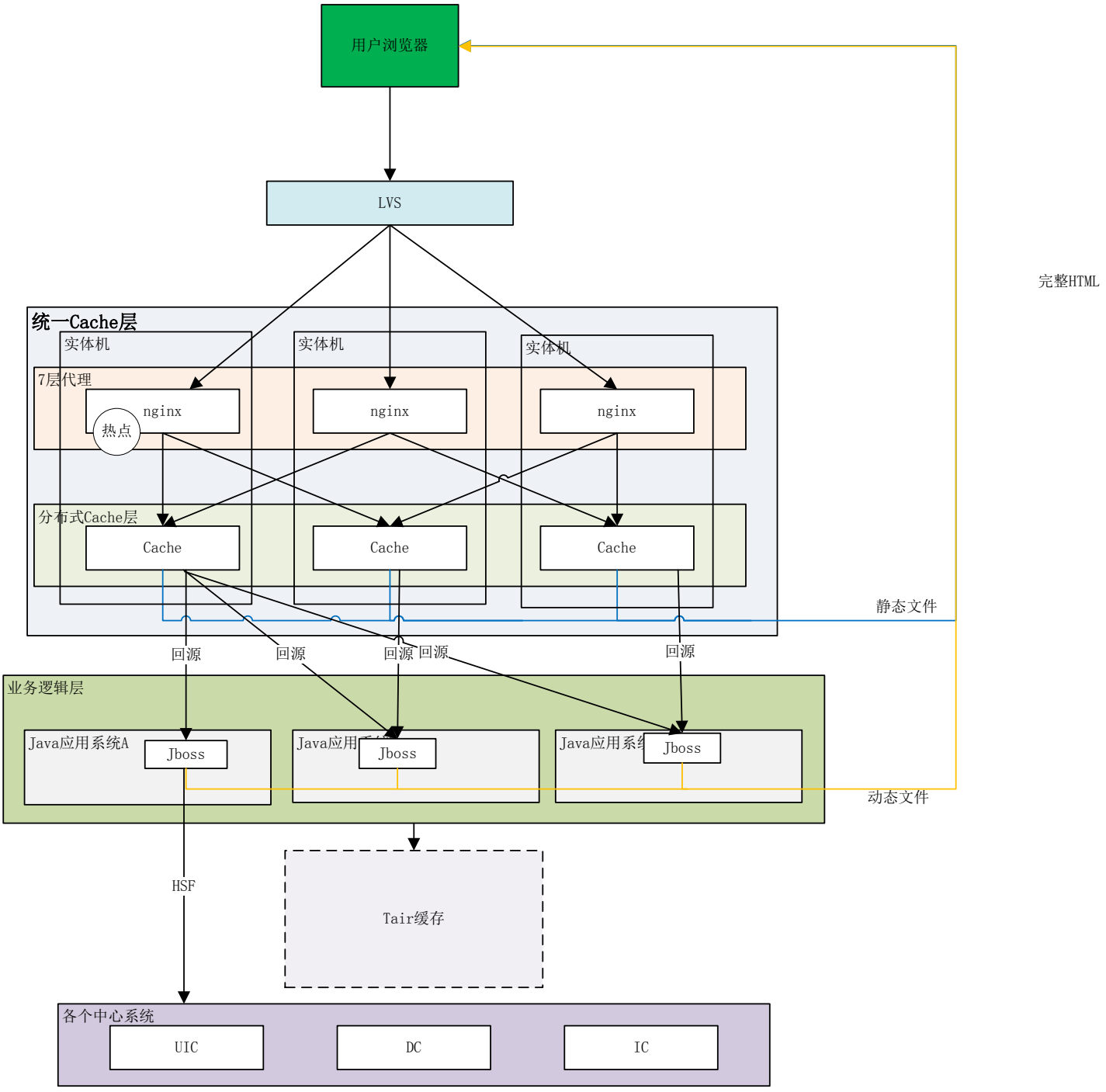
- 既没有网络瓶颈也能使用大内存

- 减少 Varnish 机器，提升命中率
- 提升命中率，能减少 Gzip 压缩
- 减少 Cache 失效压力

该方案是个比较理想的方案了，在正常请求下也能达到 50%左右的命中率，对一些基数数据比较小的系统如天猫 Detail 命中率能达到 80%左右，这样的命中率也就较理想了。

统一 Cache 层

统一 Cache 层是个更理想的推广方案，该方案的结构图如下：



统一 Cache

将 Cache 层单独拿出来统一管理可以减少运维成本，同时也方便其他静态化系统接入。他还有如下一些优点：

- 可以减少多个应用接入使用 Cache 的成本，接入的应用只有维护自己的 Java 系统就好，不用单独维护 Cache，只要关心如何使用。更好的让更多流量型系统接入使用
- 统一 Cache 易于维护，如后面加强监控、配置的自动化，统一起来维护升级比较方便

- 可以共享内存，最大化利用内存，不同系统之间的内存可以动态切换，有效应对攻击这种情况
- 更有助于安全防护

5）失效问题如何解决

搞清楚了怎么缓存，缓存什么后，接下去就是该如何失效了？失效采用主动失效加被动失效结合的方式：

被动失效

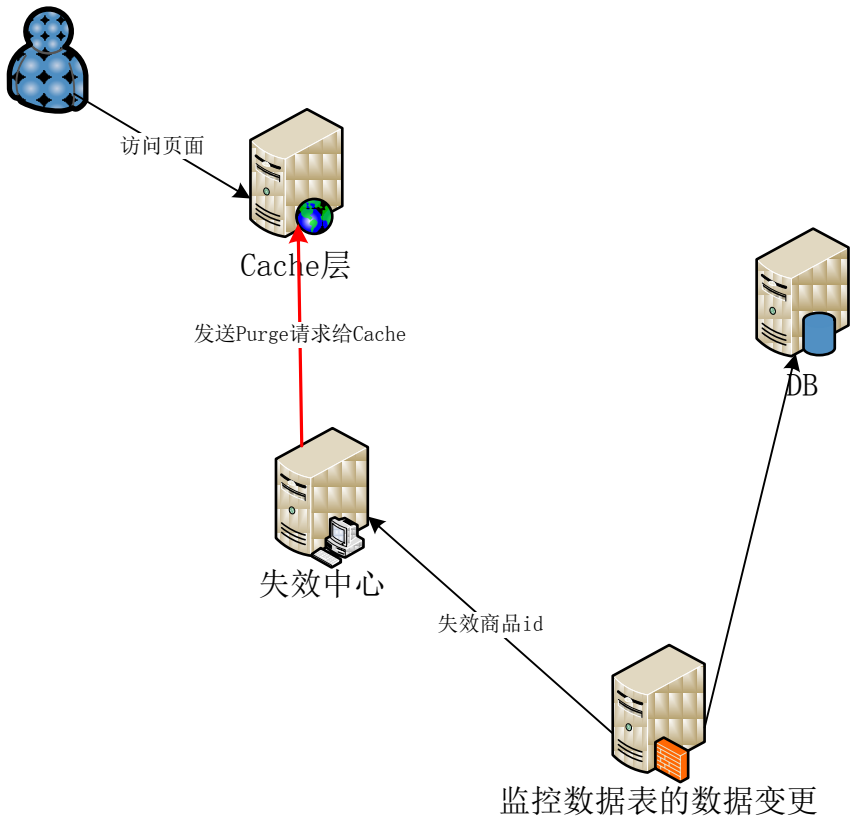
被动失效主要处理如模板变更和一些对时效性不是太敏感的数据的失效，采用设置 Cache 时间长度这种自动失效的方式。同时也要开发一个后台管理界面用于手工失效某些 Cache。

主动失效

主动失效有如下几种：

- Cache 失效中心监控数据库表变化发送 Purge 失效请求
- 装修时间戳比较失效装修内容
- Java 系统发布清空 Cache
- Vm 模板发布清空 Cache

其中失效中心承担了主要的失效功能，这个失效中心的逻辑图如下：



失效中心逻辑图

失效中心通过监控关键数据对应表的变更来发送失效请求给 Cache，从而清除 Cache 数据。

6）服务端静态化方案的演进： CDN 化

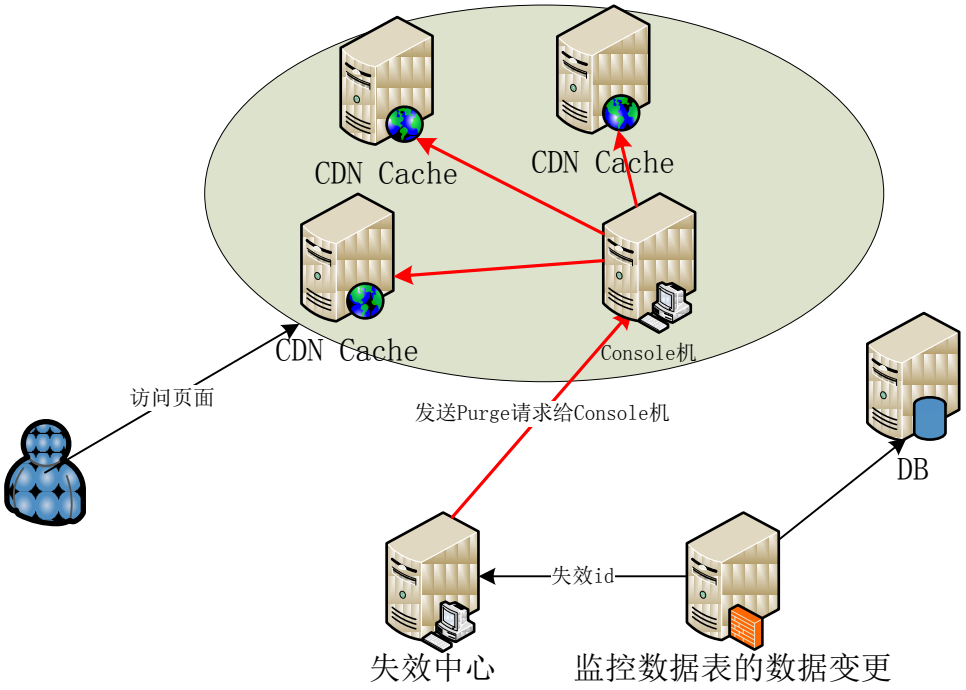
在将动态系统静态化后，自然会想到一个更进一步的方案就是讲 Cache 前移到 CDN 上，因为 CDN 离用户最近，效果会更好。但是要想这么做，还有下面几个问题需要解决：

- 失效问题。由于 CDN 分布全国要在秒级时间内失效这么广泛的 Cache，对 CDN 的失效系统要求很高。
- 命中率问题。Cache 最重要的一个指标就是要保证高命中率，不然 Cache 就是失去了意义，同样如果将数据全部放到全国的 CDN 上 Cache 分散是必然的，Cache 分散导致访问的请求命中到同一个请求的 Cache 就降低，那么命中率就成为一个问题。
- 发布更新问题。作为一个业务系统每周都有日常业务需要发布，所以发布系统的快速简单也是一个不可回避的问题，有问题快速回滚和问题排查的简便性也是要考虑的方面。

要克服这些问题才有可能将 Cache 层前移到 CDN 上，那么如何克服这些问题呢？

解决失效问题

有了服务端静态化比较成熟的失效方案，针对 CDN 可以采用类似的方式来设计级联的失效结构，采用主动发 Purge 请求给 Cache 软件失效的方式，如下图：



级联失效方式

这种失效有失效中心将失效请求发送给每个 CDN 节点上的 Console 机，然后后 Console 机来发送 Purge 请求给每台 Cache 机器。

解决命中率问题

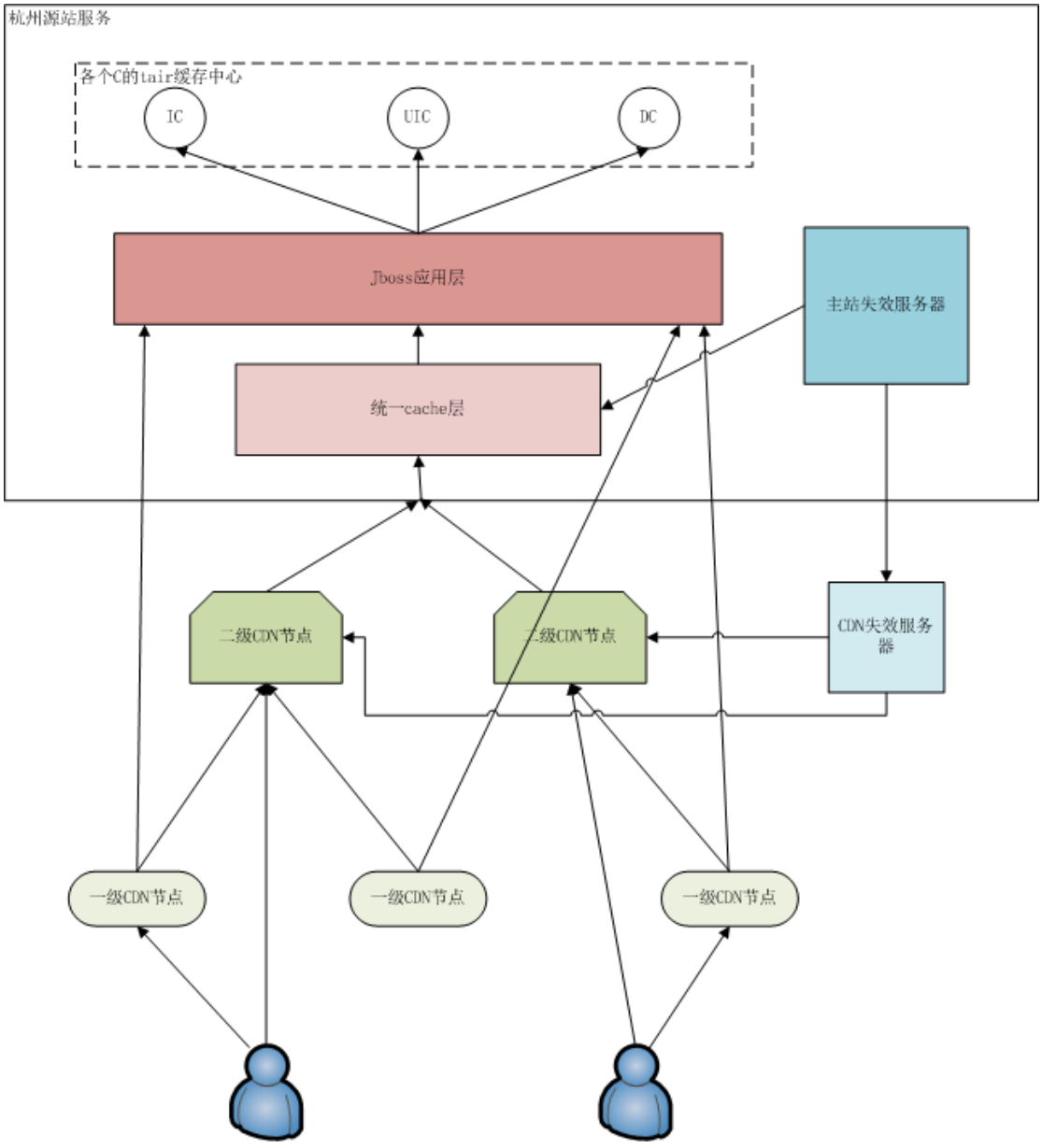
从前面的分析来看，将 Detail 放到全国的所有 CDN 节点上现阶段是不太可能实现的。那么是否可以选择若干个节点来

实施可以尝试？

这样的节点需要满足这么几个条件：

- 靠近访问量比较集中的地区
- 离杭州主站相对较远
- 节点到主站网络比较好、稳定
- 节点容量比较大不会占用其他 CDN 太多的资源
- 节点不要太多

基于上面几个因素选择 CDN 的二级 Cache 比较合适，部署方式如下图所示：



CDN 化部署方案

使用 CDN 的二级 Cache 作为缓存，可以达到和当前服务端静态化 Cache 类似的命中率，因为节点数不多，Cache 不是

很分散，访问量也比较集中，这样也解决了命中率问题，同时也提供给用户最好的访问体验，是在当前环境下比较理想的一直 CDN 化方案。

总结

本文主要介绍了淘宝高访问量系统的静态化架构设计主要以 Detail 系统为例来介绍了其中面临的各种问题和解决办法，这里虽然主要以 Detail 系统为例，其实对所有的浏览型系统都很适用，主要是按照这个架构思路多可以达到理想性能结果。

另外不仅介绍了服务端静态化的架构思路，也同时分享了这种架构的升级版本，当然这种升级需要有一定的硬件基础 CDN 基础设施的支持。对大部分互联网的读系统来说，方案 2 的架构设计已经能够满足要求。

作者简介:

许令波，2009 年毕业加入淘宝，关注性能优化领域，参与了淘宝高访问量 Web 系统主要的优化项目，著有《深入分析 Java Web 技术内幕》一书。 @淘宝君山、http://xulingbo.net,可以联系到我。