1 背景与挑战

1.1 背景介绍

1.1.1 课程概述

- 了解双11 的历程
- 学习当前主流的电商系统架构体系
- 了解大促对电商系统的一些挑战
- 面对大促活动，站在架构师角度思考，可能有哪些问题，如何应对

1.1.2 双11历程

（*最早接触双11的年份？*）

起于2009年，刚开始的双十一还并不出名，电商开展促销月都是以各自的店庆月作为基础的。国美在线是4月份，京东6月份，易购8月份，而淘宝商城选择了双十一作为促销月。促销的初衷是光棍节(11月11日)大家没事干，就该买点啥东西去当礼物送人。于是乎，双11就这样诞生了。

- 2009年 销售额0.52亿，27家品牌参与;
- 2010年 销售额9.36亿，711家品牌参与;
- 2011年 销售额33.6亿，2200家品牌参与;
- 2012年 销售额132亿，10000家品牌参与;
- 2013年 销售额352亿，20000家品牌参与;
- 2014年 销售额571亿，27000家品牌参与;
- 2015年 销售额912亿，40000家品牌参与;
- 2016年 销售额1207亿，98000家品牌参与;
- 2017年 销售额1682亿，140000家品牌参与;
- 2018年 销售额2135亿，180000家品牌参与;
- 截止到2019年11日23时59分59秒 销售额2684亿。

了解双11背景下电商公司的应对措施，有助于提升高访问量背景下的系统架构知识。

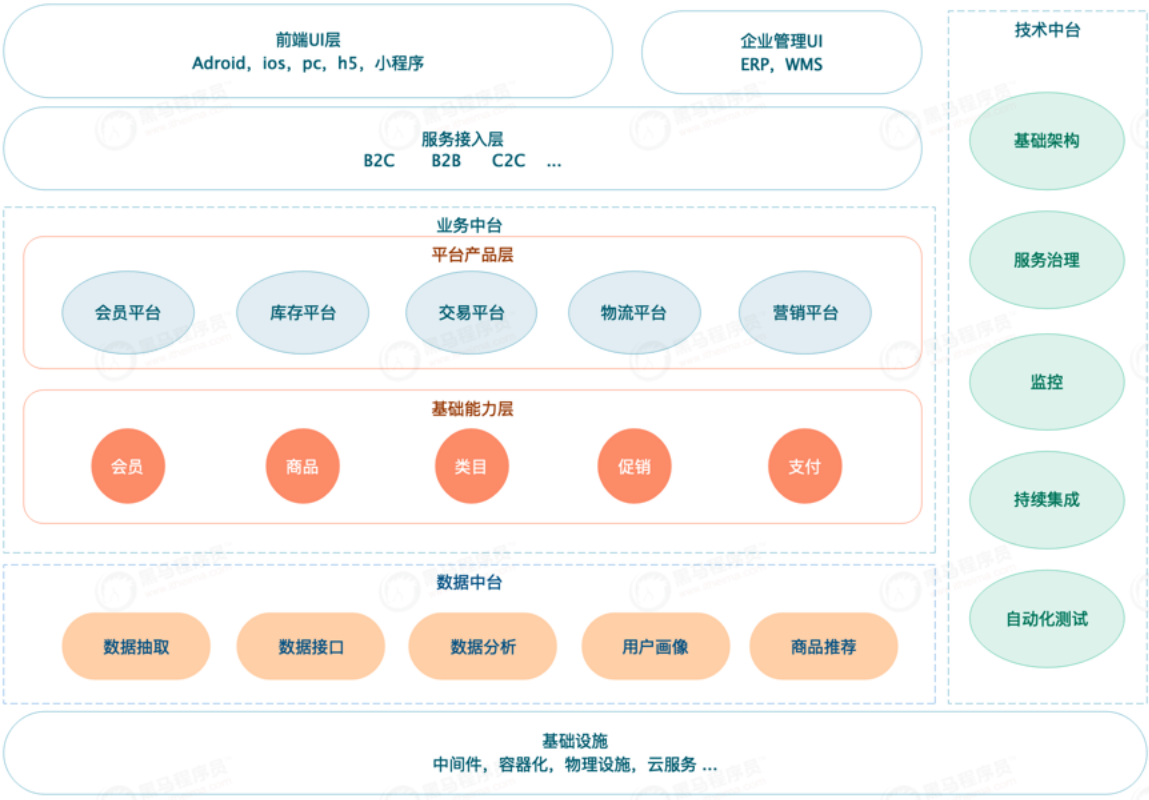
1.2 电商整体架构

1.2.1 概述

从组织架构到技术架构，当前各大电商系统基本趋于中台化。中台在2015由阿里提出，其实是一种企业架构而不是单纯的技术层面，目前几乎各大电商都进行着中台化的建设。

中台没有什么神秘的，说到底，中台就是对 ”共享“ 理念系统化的归纳和总结。

- 重复功能建设和维护带来的重复投资
- 烟囱式建设造成系统壁垒，数据孤岛
- 业务沉淀促进可持续发展
- 大中台小前台快速响应市场的需要



1.2.2 上层业务

即大中台，小前台的前台，电商中直面用户的B2B,B2C等各个业务线。

1.2.3 业务中台

业务中台基于公共服务的沉淀，需要收敛一些基础的业务服务，如商品、订单、会员、库存、财务、结算等等。

1.2.4 数据中台

数据中台不是一个平台，也不是一个系统。数据仓库、数据平台和数据中台是有区别的。简单的举例：数据平台可以理解为数据库，数据仓库类比为报表，而数据中台更贴近上层业务，带着业务属性。

1.2.5 技术中台

与业务无关的基础沉淀，中间件，系统框架，监控，日志，集成部署等等

1.2.6 运维中台

不一定存在，系统运维相关的内容，硬件，机房，包括企业云平台的建设等可以划分为单独的运维中台

1.3 面临挑战

1.3.1 考量维度

（*根据项目情况有所偏重，例如分布式与一致性是一对矛盾*）

- 高性能：提供快速的访问体验。
- 高可用：网站服务7*24正常访问。
- 可伸缩：硬件弹性增加/减少能力（快速扩容与释放）。

- 扩展性：方便地增加/减少新的功能/模块（迭代与服务降级）。
- 安全性：安全访问和数据加密、安全存储等策略。
- 敏捷性：快速应对突发情况的能力（灾备等）。

1.3.2 内部瓶颈

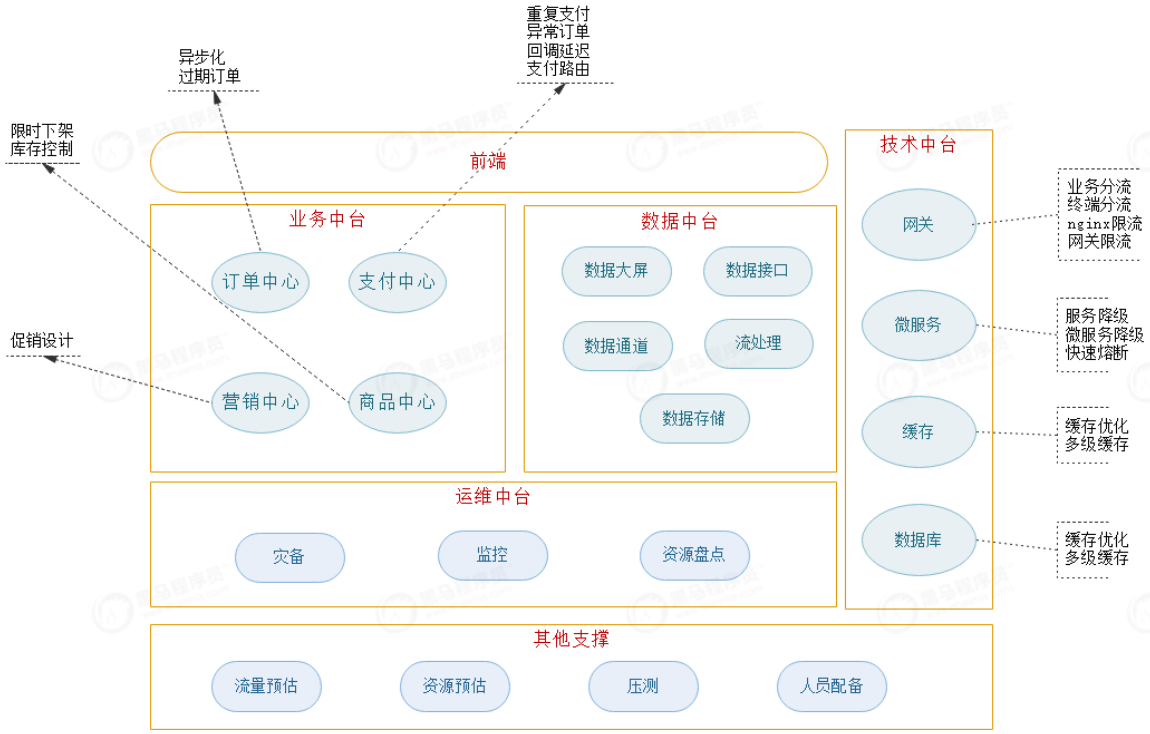
- 木桶效应：水管最细的地方决定流量，水桶最低的地方决定容量（*QPS压测调优为例*）
- CPU：序列化和反序列化、高频日志输出、大量反射、大量线程的应用
- 内存：使用内存的中间件或服务，如redis，memcache，jvm大量对象堆积内存的应用等
- 网络带宽：大流量高并发环境下，双11用户访问量激增，造成网络拥堵
- 磁盘IO：文件上传下载，数据库频繁读写，不合理或大批量的日志输出
- 数据库连接数：应对双11，应用服务器连接池大批扩容，警惕底层数据库、Redis等连接数瓶颈

1.3.3 外部服务

- 短信：外部短信延迟与送达率问题，可以搭建短信平台，多家渠道做路由和切换分流（*短信平台架构?*）
- 支付：银行支付与回调延迟，搭建支付中心，对接多支付渠道
- 快递对接：快递服务对接（快递100）
- 外部云存储：云存储文件访问，流量扩容（*大家所使用的存储? nfs的架构与事故*）
- CDN：外部静态文件访问提速服务（*使用过的项目?*）

2 应对措施

该小节从中台的各个团队角度，介绍双11期间的一些应对措施和遇到的问题。



2.1 业务中台

2.1.1 订单中心

1）异步化

（*大家使用过的mq？遇到的问题？*）

场景：

大促期间新增许多需要获取订单状态的服务，比如应对双11而临时增加的数据中台订单大屏展示等

解决：

异步化，并对消息队列调优，多队列分流

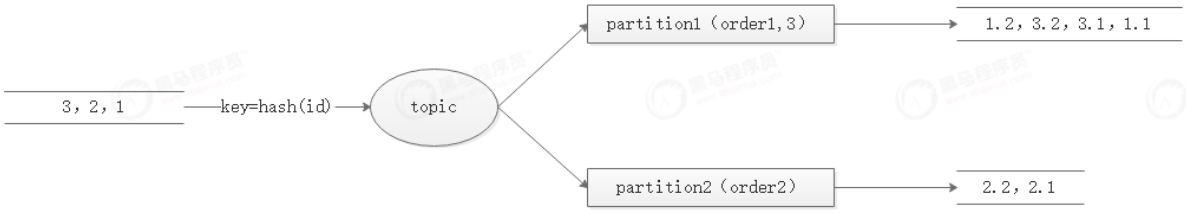
问题：

注意异步化引发的乱序问题，一是传输阶段，二是消费阶段

图解：

rabbitmq传输：队列级别顺序保障，单消费者消费一个队列可以严格保障顺序性，需要扩充队列数提升性能

kafka传输：分区级别顺序保障，只能保障投放和传输阶段的顺序性



consumer：1对1消费存在性能问题，接收消息后对key做二次分发，放入多个内存队列，开启多线程消费

2）过期订单

（*场景及思考，如果让你做架构设计有什么方案？这些方案有什么优缺点*）

双11抢单是最常见的场景，抢单不支付会占据大批量资源，如商品库存。如何取消过期订单是架构师必须面对的问题。主要有以下几种方案：

扫表实现

原理：

通过定时任务轮询扫描订单表，超时的批量修改状态

优点：

- 实现非常简单

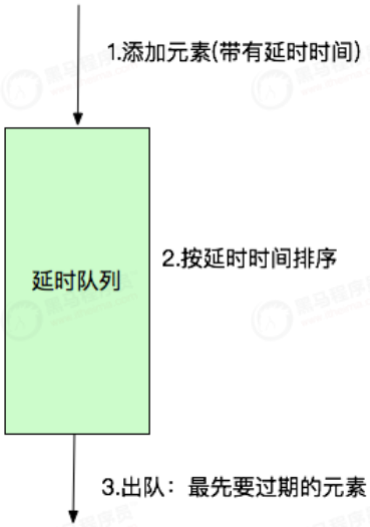
缺点:

- 大量数据集，对服务器内存消耗大。
- 数据库频繁查询，订单量大的情况下，IO是瓶颈。
- 存在延迟，间隔短则耗资源，间隔长则时效性差，两者是一对矛盾。
- 不易控制，随着定时业务的增多和细化，每个业务都要对订单重复扫描，引发查询浪费

java延迟队列实现

原理：

通过DelayQueue，每下一单，放入一个订单元素并实现getDelay()方法，方法返回该元素距离失效还剩余的时间，当<=0时元素就失效，就可以从队列中获取到。启用线程池对数据监听，一旦捕获失效订单，取出之后，调用取消逻辑进行处理。



优点:

- 基于jvm内存，效率高，任务触发时间延迟低。

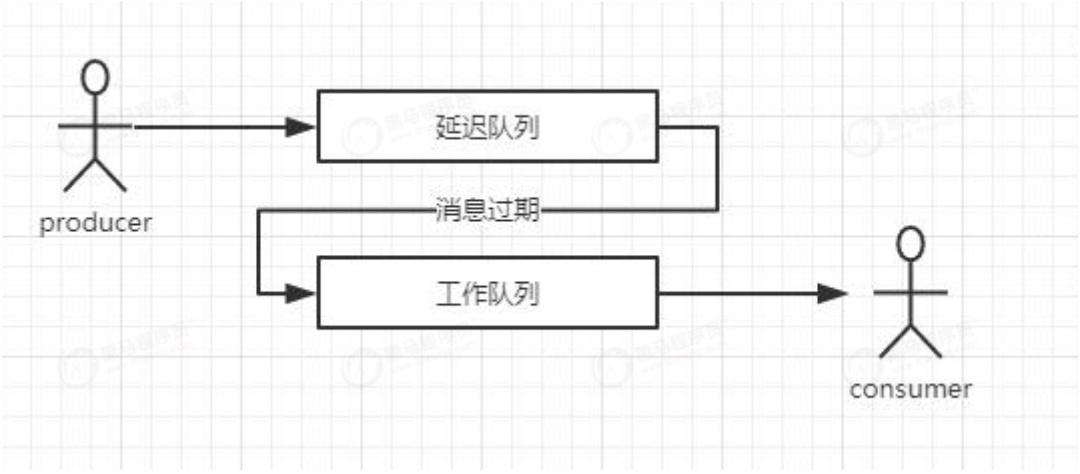
缺点:

- 存在jvm内存中，服务器重启后，数据全部丢失。
- 依赖代码硬编码，集群扩展麻烦
- 依赖jvm内存，如果订单量过大，无界队列内容扩充，容易出现OOM
- 需要代码实现，多线程处理业务，复杂度较高
- 多线程处理时，数据频繁触发等待和唤醒，多了无谓的竞争

消息队列实现

原理：

设置两个队列，每下一单放一条进延迟队列，设定过期时间。消息一旦过期，获取并放入工作队列，由consumer获取，唤起超时处理逻辑



如果采用的是RabbitMQ，其本身没有直接支持延迟队列功能，可以针对Queue和Message设置 x-message-ttl，用消息的生存时间，和死信队列来实现，具体有两种手段， A: 通过队列属性设置，队列中所有消息都有相同的过期时间，粗粒度，编码简单 B: 对消息进行单独设置，每条消息TTL可以不同，细粒度，但编码稍微复杂。

优点：

- 可以随时在队列移除，实现实时取消订单，及时恢复订单占用的资源（如商品）
- 消息存储在mq中，不占用应用服务器资源
- 异步化处理，一旦处理能力不足，consumer集群可以很方便的扩容

缺点：

- 可能会导致消息大量堆积
- mq服务器一旦故障重启后，持久化的队列过期时间会被重新计算，造成精度不足
- 死信消息可能会导致监控系统频繁预警

redis实现

原理：

利用redis的notify-keyspace-events，该选项默认为空，改为Ex开启过期事件，配置消息监听。每下一单在redis中放置一个key（如订单id），并设置过期时间。

优点:

- 消息都存储在Redis中，不占用应用内存。
- 外部redis存储，应用down机不会丢失数据。
- 做集群扩展相当方便
- 依赖redis超时，时间准确度高

缺点:

- 订单量大时，每一单都要存储redis内存，需要大量redis服务器资源

被动取消

原理：

在每次用户查询订单的时候，判断订单时间，超时则同时完成订单取消业务。

优点：

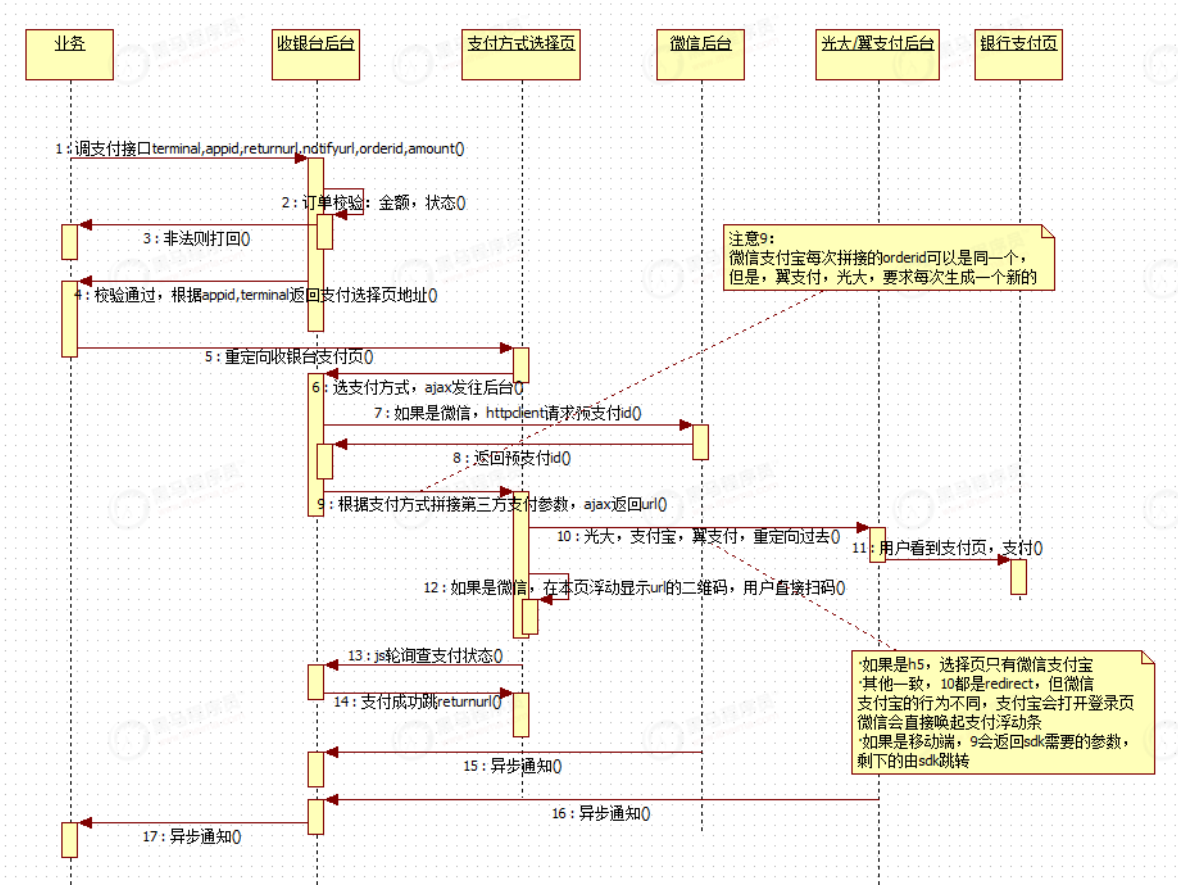
- 实现极其简单
- 不会有额外的性能付出
- 不依赖任何外部中间件，只是应用逻辑的处理

缺点：

- 延迟度不可控，如果用户一直没触发查询，则订单一直挂着，既不支付也未取消，库存也就被占着

2.1.2 支付中心

支付交互流程，支付系统设计偏重，关于做过的那些支付系统2014与2018的架构变化，政策的变动?



1）重复支付

（*2018重复支付事故*）

原因：

在第一步发起的时候，用户进入支付方式选择页。选第一个支付方式并支付完后因为通知延迟，以为支付失败。在支付又选了第二种，再次支付。

应对方案：

程序屏蔽，前端js触发按钮置灰或者遮罩提示（支付成功？遇到问题？），或者在支付方式选择页直接跳转。

后端处理，发现不同通道下的支付成功回调，抛消息队列或记录日志。

数据修复：

首先查支付日志，确认针对同一笔订单收到了不同支付渠道的回调。

其次，在支付平台管理后端可以查到入账记录，人工介入。

最后对账阶段会发现对方多帐，我方补单时出现重复订单。

问题处理：

调取退款接口或者在支付渠道的管理后台操作退款（一定要多次确认无误）。

2）异常订单

支付但未开单

场景：

用户明明支付成功，但未开通订单

问题分析：

一般支付渠道会间隔性多次回调开单链接，如果支付未开单，银行未回调的可能性比较小，着重排查开单接口是否可用。如果可用追查日志是否出现异常记录。

应对措施：

- 对账阶段可以查漏，程序自动完成补单，但是处理相对延迟，取决于支付渠道的对账文件下发周期（*2011-2013年，支付测试数据与财务人工对账的历程*）
- 人工补单，人工查询支付渠道后台数据，确认已支付的情况下，介入补单流程人工处理

未支付但已开单

场景：

用户未支付，或者财务中心未收到这笔款项，订单状态已开通。这种就问题比较严重了

应对措施：

首先排除人为操作因素。其次排查系统是否存在漏洞或者级联开单的情况

3）回调延迟

场景：

用户是期望支付完成的同时立马看到结果。但是中间多层远程的调用，可能发生订单状态更新延迟问题。

解决：

主动查询。在用户查看订单的时候，如果是类似“支付中”的中间态时，触发远程订单状态查询接口。（*大家看到的点击‘支付完成’跳转的过程，触发远程支付结果查询*）

4）支付路由

（*你所看到的收银台图标内情..*）

背景：

保障支付可用性及支付分流，支付中心对接多家渠道

方案：

- 支付中心对接多个支付渠道，支付宝，微信，各银行或第三方支付供应商。
- 对不同用户，进入支付方式选择页时，做支付分流。
- 做好监控统计，一旦某个支付渠道不可用或者延迟较大，切掉，下线，或者降权。

2.1.3 营销中心

1）概述

大促和活动不分家，一般营销中心所面对的主要是促销策略、优惠方式等业务上的架构问题。

从促销活动的范围来看，分为单品促销活动、套装促销活动、店铺促销活动，平台促销活动。

从促销类型来看，分为满减、折扣、赠品等。

业务复杂度高，一般遵循 “同类营销仅可选其一，不同类营销可叠加” 的规则。同类叠加意义不大且会造成系统复杂度上升，引发用户困惑。

2）前端设计

用户体验上的设计，比如购物车里商品的排序，按商铺分门别类。优惠总价格及时调整。这些依赖于前端的ui设计和交互体验。

3）赠品设计

(*SPU , SKU 基础概念*)

赠品有两种设计方案，一种是不做单独的SKU，只有一个空的描述，设计简单，缺点是没有商品详情页，无法给用户直观的查看和估值。

另一种是单独做SKU，赠品也会作为一个商品存在，与主商品关联，下单的时候将会自动加到商品列表，价格降为0。这种更为常见。整个商品有完善的详情页，用户可以直接看到价格甚至单独下单购买。

4）排他与优先级

检查同类别促销，将最大优惠力度的规则应用到订单，并且满足排他性，同类只享受其一。比如满10减3，满20减5，那么用户购买大于20时，只减5即可。

不同类别不做排斥，如购物车整体满减后，不影响单个商品的折扣。在记录数据时，优惠要细化到每个单独的订单明细上。退款也做到明细级别的单独退。

5）价格分摊

（*有没有遇到精度问题？价格字段如何设计？*）

满减或平台券等优惠，在多个商品下单时，涉及到金额的分摊。即 优惠总额度/购物车总额 ，得到比例后再按比例均分到每个商品。只有分摊才能在发生部分退款时退回真实金额。

但是这会涉及到一个精度问题。举例如下：满99减9活动，假设用户购买了 30+40+50=120，3件商品应付111元。按比例折算的话，9/99取4位小数是0.9090，那么分摊后为30x0.9090+40x0.9090+50x0.9090=109.08与实际支付金额出现偏差。这会造成财务无法平账。

解决方案：记账时在订单明细记录，将误差 111-109.08=1.92计入金额最大的明细，也就是50元商品上。那么最终记账为：30x0.9090 + 40x0.9090 +（50*0.909+1.92）= 111

6）退单处理

退单后要同时恢复用户的权益，比如优惠券的再次使用，限购次数等。确保用户体验。

2.1.4 商品中心

1）限时商品的下架控制

这个和超时订单设计方案类似，前面已经提到不再赘述。

2）库存管理

普通商品可以直接借助数据库锁实现，一般分乐观锁和悲观锁两种方案，如果采用悲观锁（如select语句带forupdate），会带来很大的性能阻塞，所以更多的采用乐观锁设计。（*幂等性课题的锁机制有详细讲解*）

乐观锁就是在最后执行库存扣减操作时，将事务开始前获取的库存数量带入到SQL语句中作为更新的where条件，如果数量相等，则该条更新库存的语句成功执行返回update条数为1；如果不相等，则表示该商品的库存信息已经被其他事务修改，需要放弃该条update的执行，采用重试处理。

库存秒杀商品因为大批量的访问在一瞬间涌入，数据库扛不住。可以采用 redis缓存做decr处理，正常下单后，再使用mq异步更新到db。（*秒杀不超卖课题的库存控制*）

2.2 技术中台

2.2.1 数据库优化

数据库层的调优，一般发生在大促前的预备阶段，一旦大促开始，对数据库的优化已经来不及了。

- 在大促开始前梳理耗时查询业务，对关键业务压测。
- 开启mysql的慢查询日志（两种方式）

```
#配置文件方式，需要重启mysql
#日志文件位置
log-slow-queries=/opt/data/slowquery.log
#超时时间，默认10s
long_query_time=2

#临时开启，不需要重启
set global slow_query_log=on;
set global long_query_time=10;
set global slow_query_log_file='/opt/data/slow_query.log'
```

- 使用mysqldumpslow命令解析mysql慢查询日志

```
-- 慢查询日志以文本打开，可读性很高
-- 查询次数，耗时，锁时间，返回结果集条数（扫描行数），执行者
Count: 1  Time=10.91s (10s)  Lock=0.00s (0s)  Rows=1000.0 (1000),
mysql[mysql]@[10.1.1.1]
SELECT * FROM order_history
```

- 借助explain查看sql执行计划，对sql调优，或其他优化工具。

2.2.2 缓存优化

（*业务篇红包雨课题里有缓存结构的深度应用*）

1） 策略

热点数据预热：

（*常规加载机制画图展示*）

常规缓存设计趋向于懒加载，大促期间的热点数据尽量做到预热加载。比如某个促销专题，不要等待活动开始的一瞬间再读库加缓存，搞不好引发击穿。

细粒度设计：

（*细粒度缓存结构画图展示*）

集合与单体分开存储，缓存结构细粒度化。如某个橱窗的推荐商品列表，常规存储一个key，value为整个商品集合。优化为列表与每个商品详细信息设置两个独立缓存值，在查询环节组装，可以降低发生修改时对缓存的冲击。新增一个推荐则失效列表，修改商品则仅仅失效当前商品缓存。

可用性：

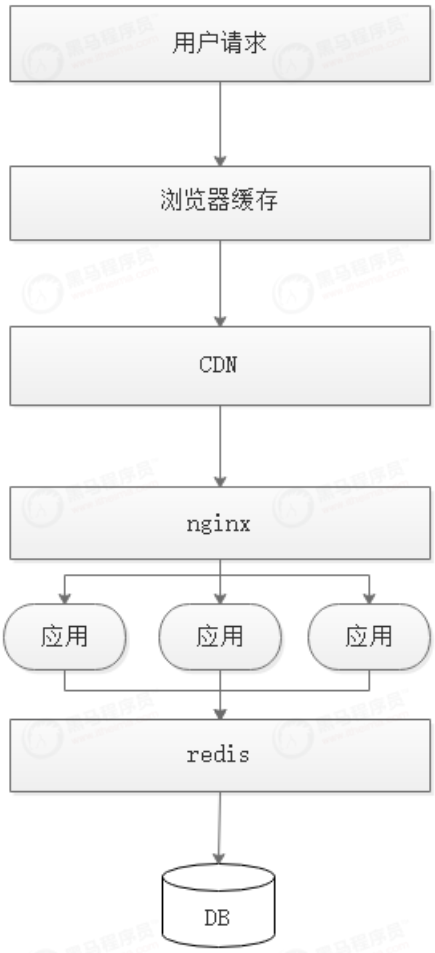
（*回顾三种缓存问题*）

- 只要缓存失效时间设置分散，雪崩的可能性不大
- 防范恶意攻击引发穿透，前端做到防刷，业务层面要注意合法性校验，非法key的失效时间需要评估。

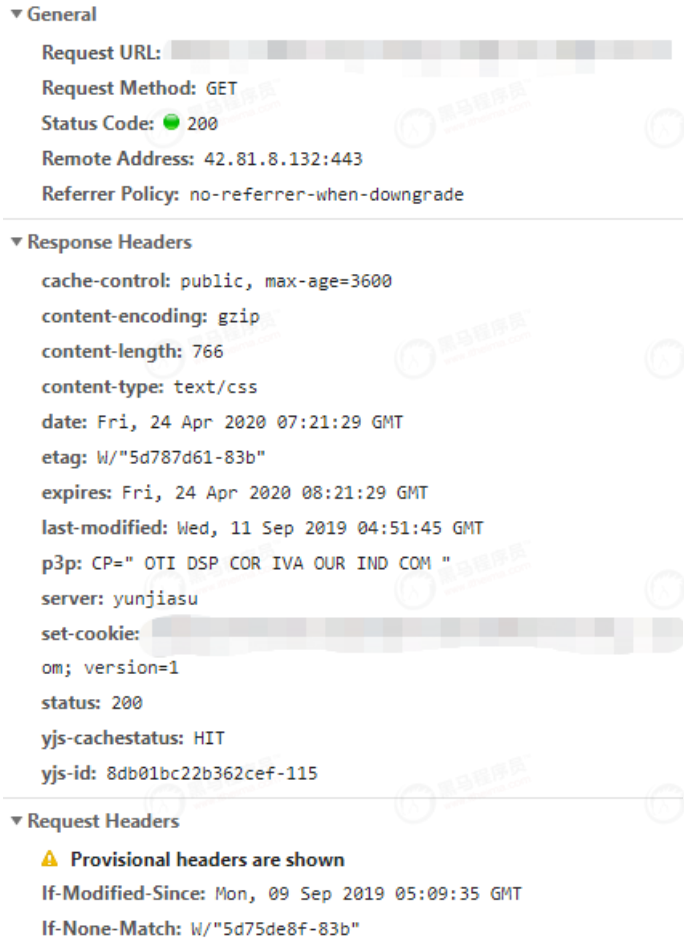
- 击穿可能性升高，大促高并发下，修改时，如果采用key删除策略很可能触发击穿，修改少可以优化为双写。

2）多级缓存

优化缓存体系，对关键业务请求，如商品详情页，采用多级缓存处理



首先看浏览器缓存，一般浏览器缓存可分为两种手段，分别交给浏览器和服务端执行



- 客户端判决：为请求Header设置Expires（http1.0）和 Cache-Control（http1.1），客户端本地比较决定是否使用缓存
- 服务端判决：借助Last-Modified/If-Modified-Since（http1.0）或ETag/If-None-Match，服务器端比较决定返回200带body还是304仅有head头 (*画图展示*)
- Last-Modified < ETag < Expires < Cache-Control

CDN：借助CDN的dns解析，对用户做ip分流，CDN作为应用服务器的代理，抵挡前端的流量洪峰。同样，前面提到的http缓存策略对CDN依然有效。

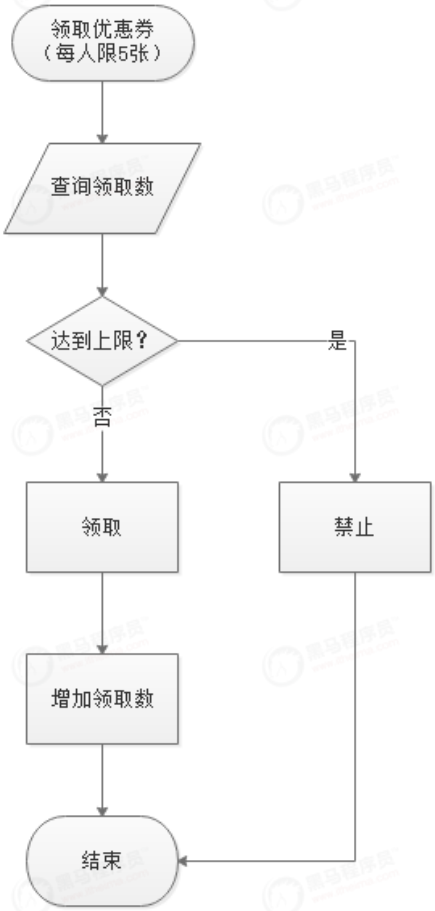
nginx缓存：nginx除了作为负载均衡，也可以作为请求级别的缓存，一段典型配置如下：

```
    # 定义缓存路径、过期时间、空间大小等
    proxy_cache_path  /tmp/nginx/cache levels=2:2:2 use_temp_path=off
keys_zone=my_cache:10m inactive=1h max_size=1g;

    server {
        listen       80;
        server_name  xxx.xxx.com;
        # 添加header头，缓存状态信息
        add_header X-Cache-Status $upstream_cache_status;

        location / {
                # 定义缓存名
                proxy_cache my_cache;
                # 定义缓存key
                proxy_cache_key $host$uri$is_args$args;
```

```
                # 针对返回状态码单独定义缓存时间
                proxy_cache_valid 200 304 10m;
                # url 上设置请求 nocache=true 时不走缓存。
                proxy_cache_bypass $arg_nocache $http_nocahe;

                proxy_pass http://localhost:8080;
        }
    }
```

分布式缓存：redis做应用层缓存，不多解释。但是要注意做好扩容预案和业务层优化

- 依据预估量做相应的扩容或资源申请，纯做缓存时可以关闭持久化，内存超出60%将变得不稳定。
- 频繁交互业务从java端下移到lua脚本实现，一方面可以实现原子性，另一方面有效减少网络延时和数据的冗余传输。以平台优惠券领取为例：（*画图对照redis指令*）

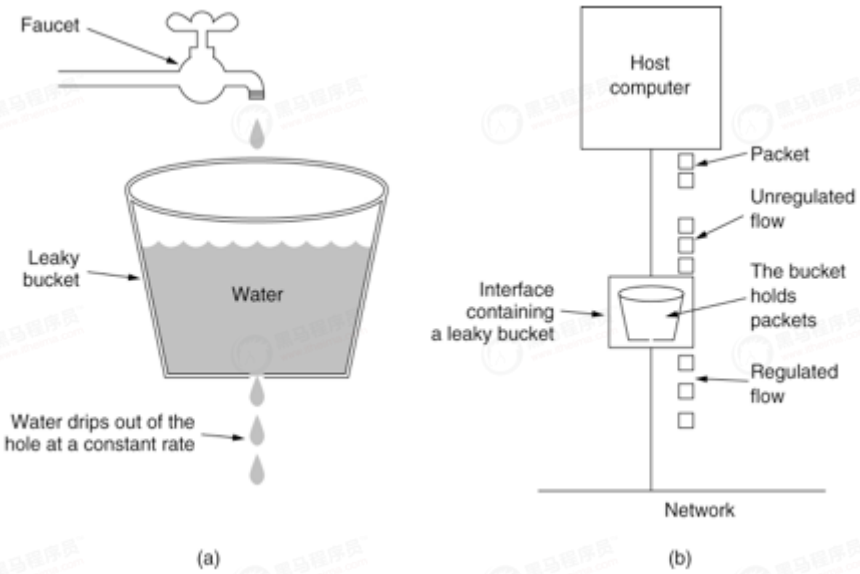


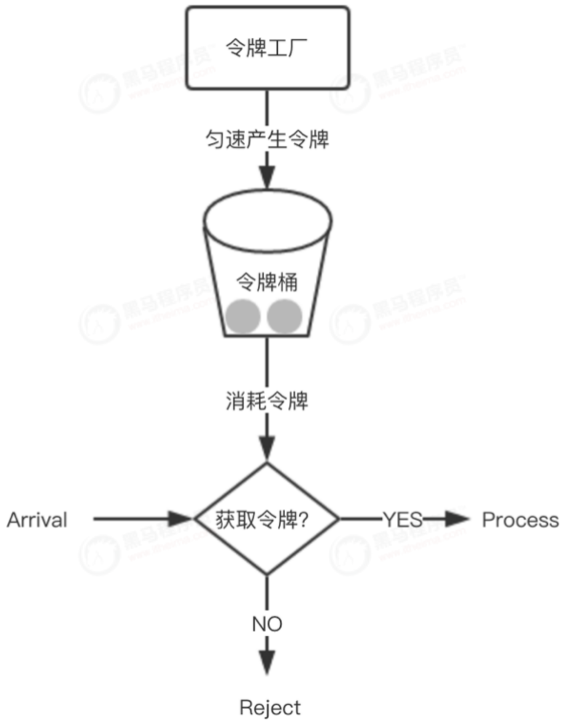
2.2.3 分流与限流

（*算法与数据结构应用 - 限流算法有详细实现*）

CDN的引入本身起到了按ip分流的作用，但是我们可以在下层做到更细粒度化的控制。根据业务情况将不同的请求分流到各自的服务器。

限流不同与分流，是对下层的保护，当系统超过一定流量后，超过的流量做直接拒绝处理，以便保护后端的服务，原则就是要么不进来，进来的都正常服务。常见的限流算法有三种：计数器、漏桶、令牌桶。





1） 业务分流

根据不同的业务线分发请求，配备二级域名如b2b.xxx.com，b2c.xxx.com，或者在nginx软负载层针对不同虚拟主机名做upstream分发

新上的双11活动页，或者促销专题页面，采用新访问入口和机器部署，与主站分离。活动结束后也利于机器资源的快速释放

2） 终端分流

按不同的请求终端分流，在header头的user-agent中可以捕获用户的访问终端。android，ios，pc，根据不同终端设备，做流量分发，到不同的应用机器。同时方便对用户终端流量的监控和统计。

3）nginx限流

评估双11可能的流量，结合具体业务模块，配备对应限流措施。主要有流量限制和连接数限制两个维度。

流量限制：限制访问频率，其目的是基于漏桶算法实现ip级别的防刷。Nginx中使用ngx_http_limit_req_module 模块来限制请求的访问频率，基于漏桶算法原理实现。

```
#$binary_remote_addr 表示通过remote_addr这个标识来做限制
#binary_的目的是缩写内存占用量，是限制同一客户端ip地址
#zone=one:10m表示生成一个大小为10M，名字为one的内存区域，用来存储访问的频次信息
#rate=1r/s表示允许相同标识的客户端的访问频次
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;

server {
    location /b2b/ {
    #zone=one 设置使用哪个配置区域来做限制，与上面limit_req_zone 里的name对应
    #burst=5,设置一个大小为5的缓冲区,当有大量请求时，超过了访问频次限制的请求可以先放到这个缓冲区内
    #nodelay，如果设置，超过访问频次而且缓冲区也满了的时候就会直接返回503,如果没有设置，则所有请求会等待排队
    limit_req zone=one burst=5 nodelay;
    }
}
```

连接数限制：Nginx 的 ngx_http_limit_conn_module模块提供了对资源连接数进行限制的功能。

```
#$binary_remote_addr同上
limit_conn_zone $binary_remote_addr zone=addr:10m;
server {
    location /b2b/ {
        # 限制每个ip下最大只能有一个连接
        limit_conn addr 1;
    }
}
```

4）网关限流

从代理服务器放进来的流量，会进入应用服务器，第一道关卡是微服务的网关。应对大促，针对各个微服务具体业务具体分析，配备对应限流措施。zuul和gateway是团队中最常遇到的网关组件。

```
zuul.routes.userinfo.path=/user/**
zuul.routes.userinfo.serviceId=user-service
zuul.ratelimit.enabled=true
zuul.ratelimit.policies.userinfo.limit=3
zuul.ratelimit.policies.userinfo.refresh-interval=60
zuul.ratelimit.policies.userinfo.type=origin
```

```yaml
routes:
        - id: user_route
          uri: http://localhost:8080
          predicates:
            - Path=/*
          filters:
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 1
                redis-rate-limiter.burstCapacity: 1
                key-resolver: "#{@ipKeyResolver}
```

2.2.4 服务降级

当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。是一种舍车保帅的策略。

比如平时客户来我的店铺购买衣服。平时可以试穿，给出建议，帮助搭配，最后下单支付，送用户祝福卡片等。双11大促则简单粗暴响应下单支付收钱发货。其他不太重要的服务关闭开关，腾出资源让位主交易流程。

服务降级可以从前端页面，后端微服务两个点着手。

1）页面降级

很好理解，针对页面元素处理，将不重要的操作入口置灰或屏蔽。平时调用后端接口实时呈现数据的地方替换为静态页也可以理解为一种降级操作。

2）微服务降级

配置接口开关，并通过配置中心可以灵活开闭。必要时关闭开关，屏蔽接口的实际查询，直接返回mock数据。例如，购买了本商品的用户还购买过哪些商品接口，在业务上需要调用数据中台订单统计服务，访问量大时，关闭对外调用，直接返回设置好的一批相关商品，起到降级保护作用。

3）快速熔断

快速熔断可以认为是在应对突发情况时，对服务请求结果准确性的一种妥协。避免因单一服务垮台导致整个调用链路崩溃。常用手段如下：

- 抛异常：这种处理需要上层配备异常处理机制，在捕获异常时，导向错误页、等待页或稍后重试等。
- 返回NULL：简单粗暴，可能会出现空白结果，并不友好。
- 调用Fallback处理逻辑：更人性化的手段，也最常用。为每个业务配备一个备选方案。

举个例子：商品页或订单详情页面，一般都会有猜你喜欢这个模块，通过对用户的购买行为、浏览记录、收藏记录等等进行大数据分析，然后给每一个用户推送自己可能喜欢的商品。在双11大促背景下，如果推荐服务压力过大，出现服务出错、网络延迟等等之类突发情况，导致最后调用服务失败，则可以配备一个fallback，直接返回当前商品同类别下的几款商品，作为备选方案，这比抛异常或者返回null空白页面体验要更优。

2.2.5 安全性

大促前做好安全防范。常见的DDos，Arp，脚本等攻击平时也会存在，日常防范已经配备。大促期间需要注意的可能更多的是业务层面的入侵，比如抢购或秒杀时的恶意刷接口。

- 实名制，限制单用户，单ip等维度下的频次
- 必要的地方添加验证码（图片复杂度升级，或滑块等新型方式）
- 黑名单机制，一旦发现恶意行为，列入黑名单，并自动维护

2.3 运维中台

2.3.1 做好灾备

（*2018从一次断电看灾备的背景与经历，30分钟以内*）

灾备是应对大型故障的保底措施，最好的结局是永远不要触发，但是大促前需要做好灾备切换演练，可以选择大促前用户量少的时间段进行：

1）前期准备：两地灾备程序同步维护，大促相关的迭代和活动专题上线确保两地测试ok，镜像版本统一

2）数据库配置两地主从，或双主单写。切换前做好数据同步性检查

3）启用脚本，切换代理服务器，代理流量转入灾备机房，正式环境还需要处理dns指向

4）分布式文件灾备日常采用rsync等实时同步，采用云存储的可以忽略

5）es索引等其他数据确保日常同步

6）注意挂好维护页，友好提示

7）配备自动化测试脚本以便快速验证切换结果

2.3.2 配备监控

1）基础设施监控

包括物理机、Docker 容器、以及对交换机、IP 进行监控

借助zabbix等开源软件对机器资源配置监控，如果采用云化部署，各大云供应商都会配备完善的监控机制

2）应用级监控

主动监控，日志或消息队列形式打点输出，定时汇报 （*日志平台追踪课题*）

被动监控，添加监控接口，监控系统定时请求确认可用性

3）业务监控

对具体业务点做监控处理，如订单量、登录量、注册量、某些页面的访问量等关键点采用异步消息方式推送到监控中心，监控中心针对特定队列的数据做统计和展示。

4）客服一线反馈

主动监控依然无法察觉的情况下，来自客服的一线反馈成为最后关卡。优先级也最高。开发故障快速响应平台，做到实时性保障。做到客服 - 业务线 - 产品 - 技术排查的及时响应，快速排查。

2.3.3 资源盘点

1）网络设施扩容

网络带宽是影响访问流量的重要因素，做好各个机房网络带宽预估，数据在两地机房间传输并且要求低延迟的场景，如数据库主从，可以考虑机房专线。使用公有云的服务，可以购买临时流量。

2）硬件资源盘点

对容量做预估和硬件资源盘点。配合大促期间不同服务的架构设计，以及项目本身的特性，对cpu，内存做评估。偏运算的项目，重度使用多线程的项目偏cpu，需要大量对象或集合处理的项目偏内存。

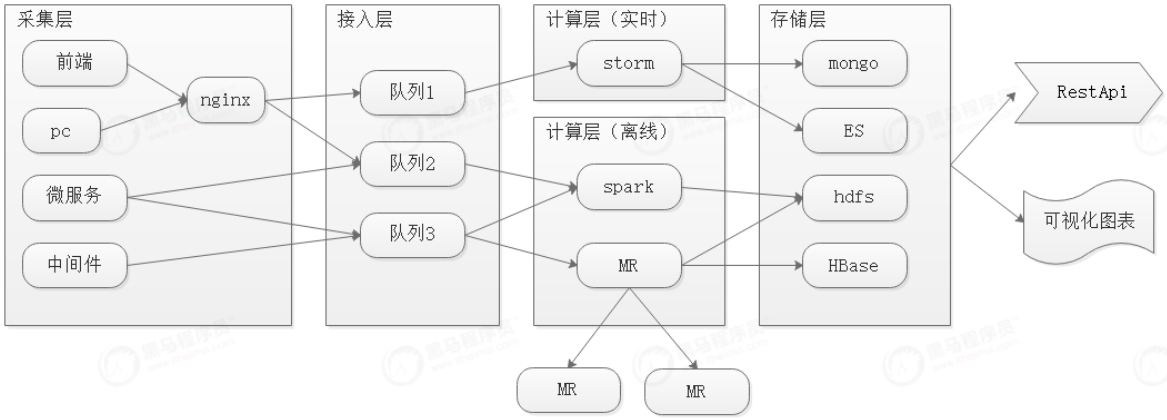
3）容器盘点

所有项目容器化部署，基于镜像即版本理念，打好各个服务的镜像是docker快速复制扩容的基础。大促前对各个中心微服务做统计和盘点。

借助swarm和k8s等编排工具，快速实现容器的伸缩。 （*运维篇会讲到*）

2.4 数据中台

数据中台多为大数据相关架构体系，大促期间，同样可能面临大批数据洪峰，比如订单量激增、用户行为数据暴涨等场景。简单看一下可能需要做的一些应对。（*大屏实时计算课题*）



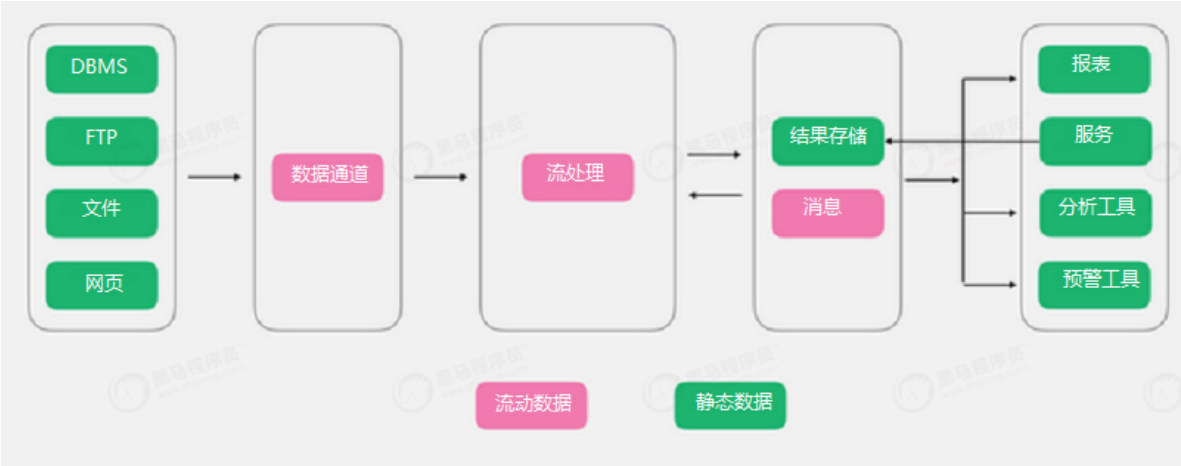
2.4.1 数据通道

对数据传输通道扩容，比如kafka扩大分区数，rabbitmq增加细分队列。一方面实现了扩容，另一方面在传输的起始阶段就对数据做了一定的分类。

数据降级，关闭某些非核心数据的通道采集，让位网络带宽给核心业务数据。

2.4.2 数据展示

数据大屏开发。对实时性有一定要求，多采用流式运算。



2.5 其他准备

2.5.1 流量预估

对关键业务的体量做好预估。如用户的注册、下单量、首页，商品详情页等关键页面的qps，为压测提供参考指标。

2.5.2 资源预估

架构师统计各中心服务关系，对各个服务扩容做预估，汇总。

2.5.3 压测准备

（*全链路压测课题*）

1）线下压测

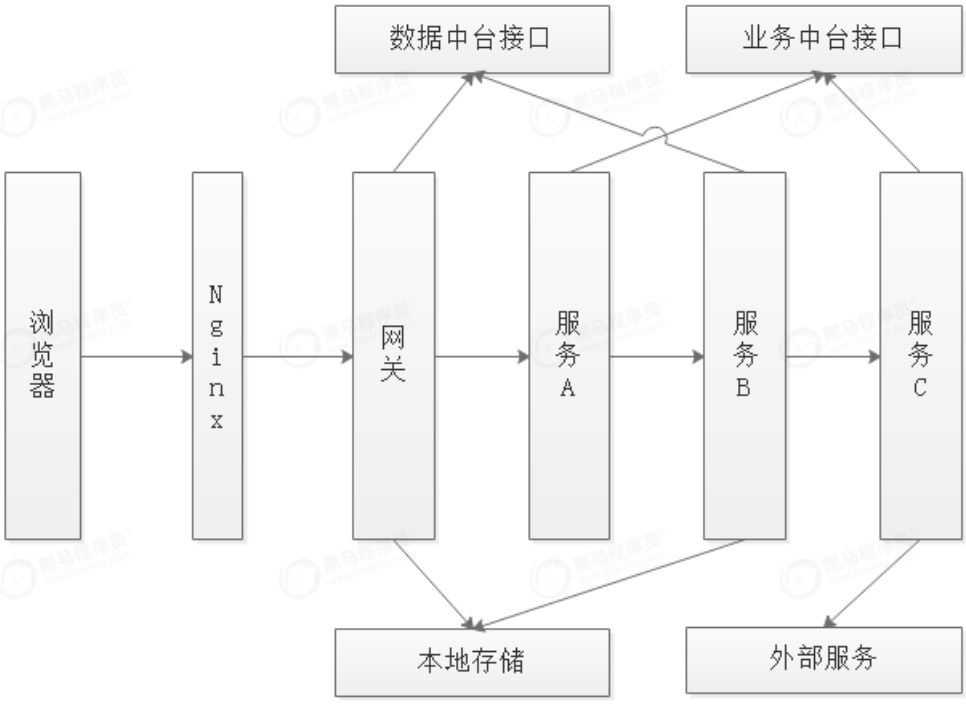
（*大家当前使用的环境都有哪些？上线模式是什么样的*）

当前成熟系统都具备各种环境，开发环境、测试环境、准生产环境等，对线下可以选择准生产环境做为压测，模拟线上。

线下压测数据安全，不必担心对线上造成干扰。所压测的值可以用于相对性比较，比如其中全链路的某个环境哪个是瓶颈。但是无法精准反馈线上的真实场景。

2）线上压测（谨慎！）

重点看线上压测，线上压测压出的数据是最真实有效的。但是因为使用的是生产环境，操作不当可能引发灾难性后果。



1）在全链路压测环境下，服务调用关系错综复杂，最重要的是实现压测流量的标识，以及标识在服务上下文间如何有效传递不丢失。服务内借助threadlocal，但是要注意多线程下失效。服务间通过改写远程调用框架或借助框架提供的Context设置。（*分布式日志平台，访问链路追踪课题*）

2）数据隔离，数据库可以创建影子表，redis等缓存可以设置shadow_等前缀，从开发框架层面封装处理，对数据层持久化框架做二次开发，使其自动发现压测数据。

3）外部服务可以借助服务降级功能，添加开关判断属于压测流量时开关进入降级或mock，比如收银程序添加挡板，直接返回成功，短信应用直接默认一个短信号码。

4）日志打印需要隔离，可以借助分布式日志平台收集时采用不同的输出通道和队列。

5）压测数据最好的方式是流量克隆（TCPCopy工具等），将线上的实际访问请求克隆放大几倍加压到压测入口，如果实现不了，尽量模拟线上的真实数据结构和体量。

5）做好全压流量规划，按预估2~3倍加压，确定流量比例，打压。

2.5.4 人员配备

人员互备，防止故障，及时响应，应对双11不是什么神秘事。

任务

- 梳理前两天的课题中自己不太熟的知识点
- 在日常项目中多换位思考，换成我如何设计，有什么问题。
- 多归纳总结，在设计中参考下架构思想