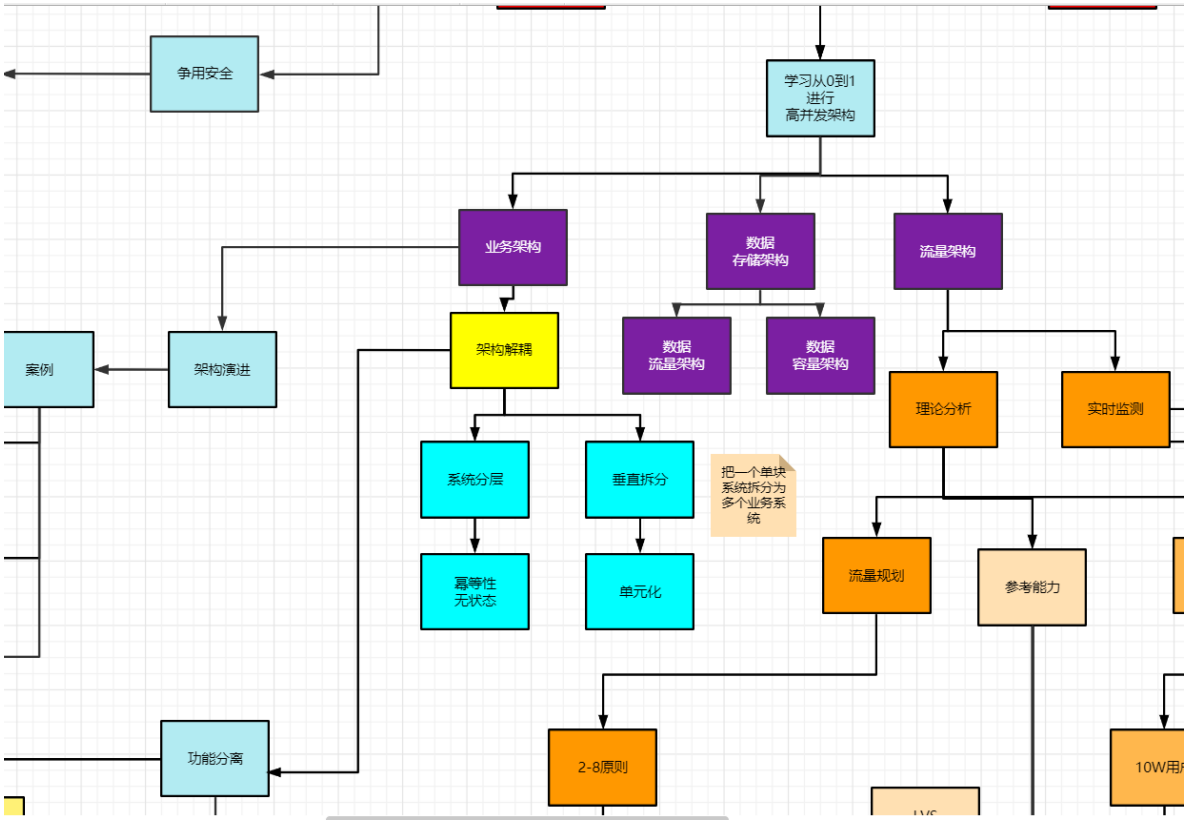
亿级流量秒杀架构

那么我们怎么设计秒杀系统，才能保证秒杀系统的高性能和稳定性，在高并发场景下能高可用，不会出现系统性的雪崩呢？

先看看秒杀场景特点。

> 巨大的突发流量： 秒杀开始前几分钟，大量用户开始进入秒杀，开始频繁刷新秒杀商品详情页，这时秒杀商品详情页访问量会猛增。
>
> 巨大的后端无效流量： 流量越大，说明诱惑越大，说明是割肉式的售卖，所以，真正的有效商品，不会太多，可能是10W人，秒杀100件，或者100W人，秒杀1W件。
>
> 后端压力也大： 大量用户开始抢购操作，这时创建订单，扣库存压力会显著增大。

> 下面，从业务架构 + 流量架构 + 数据架构的维度，参考秒杀系统，进行亿级流量的架构设计。



功能架构（功能分离）

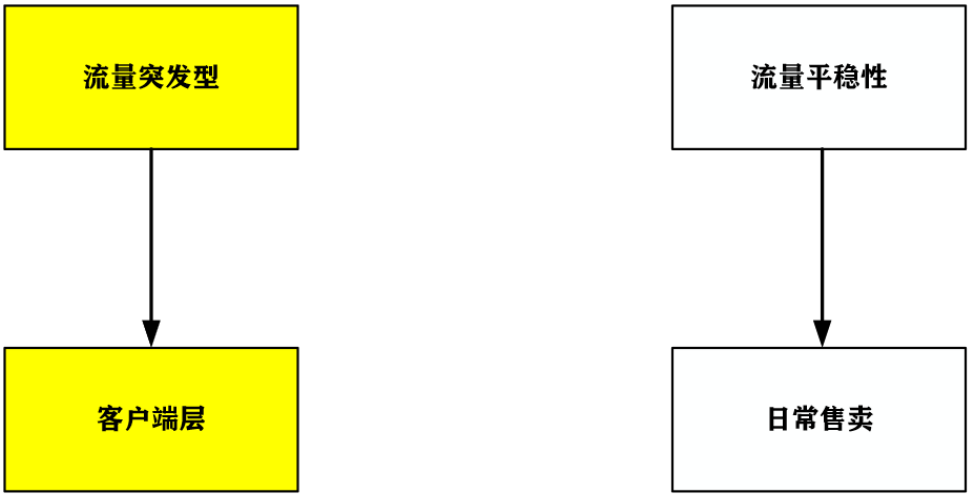
通过对系统业务的仔细分析，对业务功能进行分级，分成核心功能和非核心功能，确保核心功能的高可用和高并发。

重要程度，仅仅是功能分离的一种维度。

还可以按照其他维度进行划分，比如流量特点.

秒杀的功能分离

在秒杀系统中，这里需要区分。可以区分为突发流量型、与平缓流量型的功能，从业务上把秒杀和日常的售卖区分开来，对突发流量的功能做好隔离。



把秒杀做为营销活动，要参与秒杀的商品需要提前报名参加活动，这样我们就能提前知道哪些商家哪些商品要参与秒杀，对应于秒杀系统的核心功能，例如：商品详情，下单等。

对于核心功能的秒杀服务，最好做好动态扩展的方案，如基于k8s做动态扩展，或者结合公有云做动态扩展。

系统架构（系统分层）

常见互联网分层架构

常见互联网分层架构，分为：

（1）**客户端层：**客户端层是浏览器browser或者手机应用APP；

（2）**接入层**：系统入口，负载均衡、反向代理；

（3）**服务层**：实现核心应用逻辑，返回html或者json；

（4）**缓存层**：缓存加速访问存储；

（5）**数据库层**：结构化db和异构db；

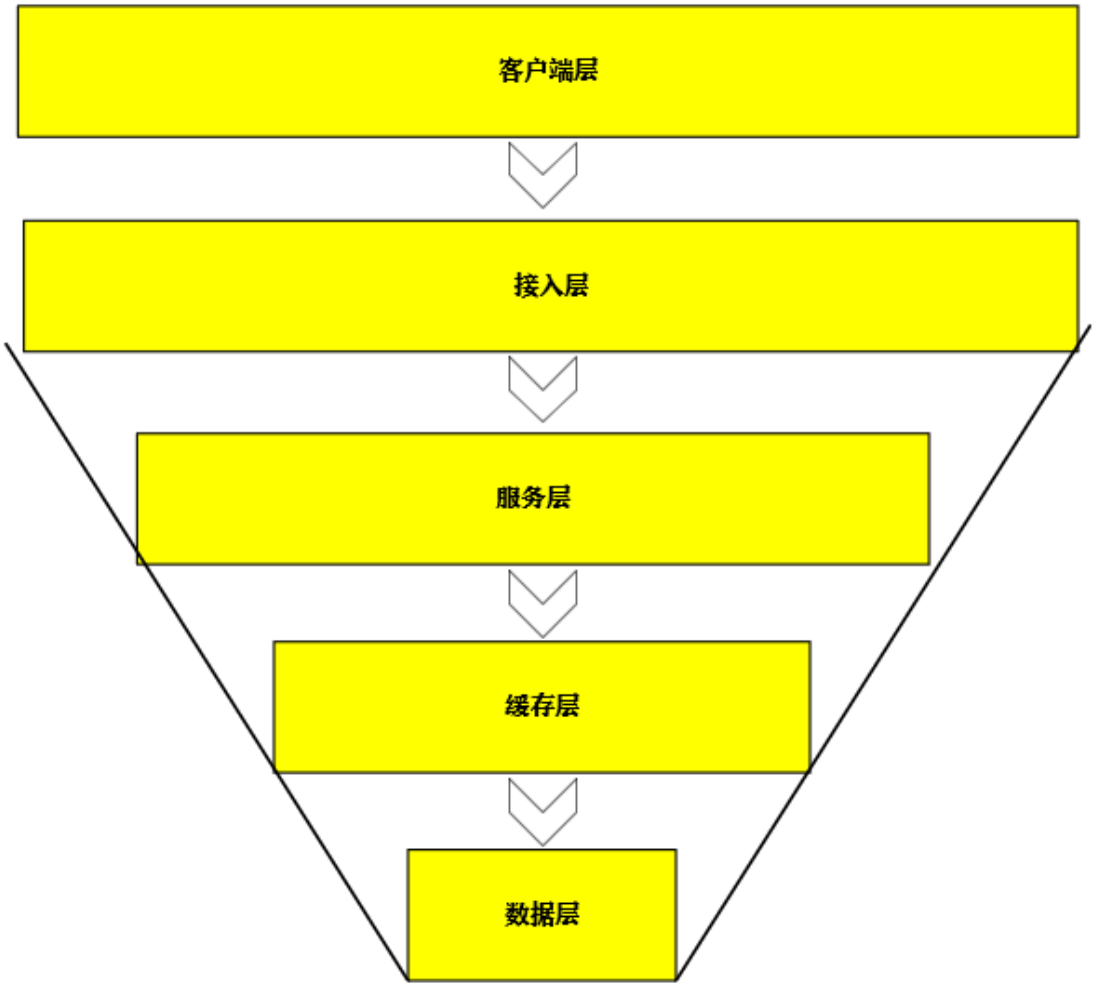
（6）**中间件：** zk、xxl-job，rocketmq

漏斗型

漏斗型业务，指的是，用户的请求，从客户端到 db 层，层层递减，递减的程度视业务而定。

例如当 10w 人去抢 1 00个物品时，db 层的请求在个位数量级 1000以内，这就是比较理想的模型。

如下图所示



读请求的处理(读高并发)：

争取在接入层解决战斗。

秒杀商品的静态页面：

接入层搞定：流量走CDN、最多到达Ng

亿级用户，10WQPS，走到NG，那么就是 LVS+10-20个NG，就可以搞定

秒杀商品的动态数据：

接入层搞定：Ng+本地缓存+分布式缓存 搞定

写请求的处理(写高并发)：

> 假设10W用户，在1m内秒杀一个商品的1000个sku。也就是QPS为10W、假设秒杀之前进行库存检查，没有就秒杀失败。

诀窍：

1 有效流量过滤，过滤出有效请求

2 有效流量削峰，有效请求 达到或者超出 下一层的能力瓶颈， 同步处理，变成（降级为）异步处理

3 有效流量限流，1000Qps， 接入层、服务层、数据库

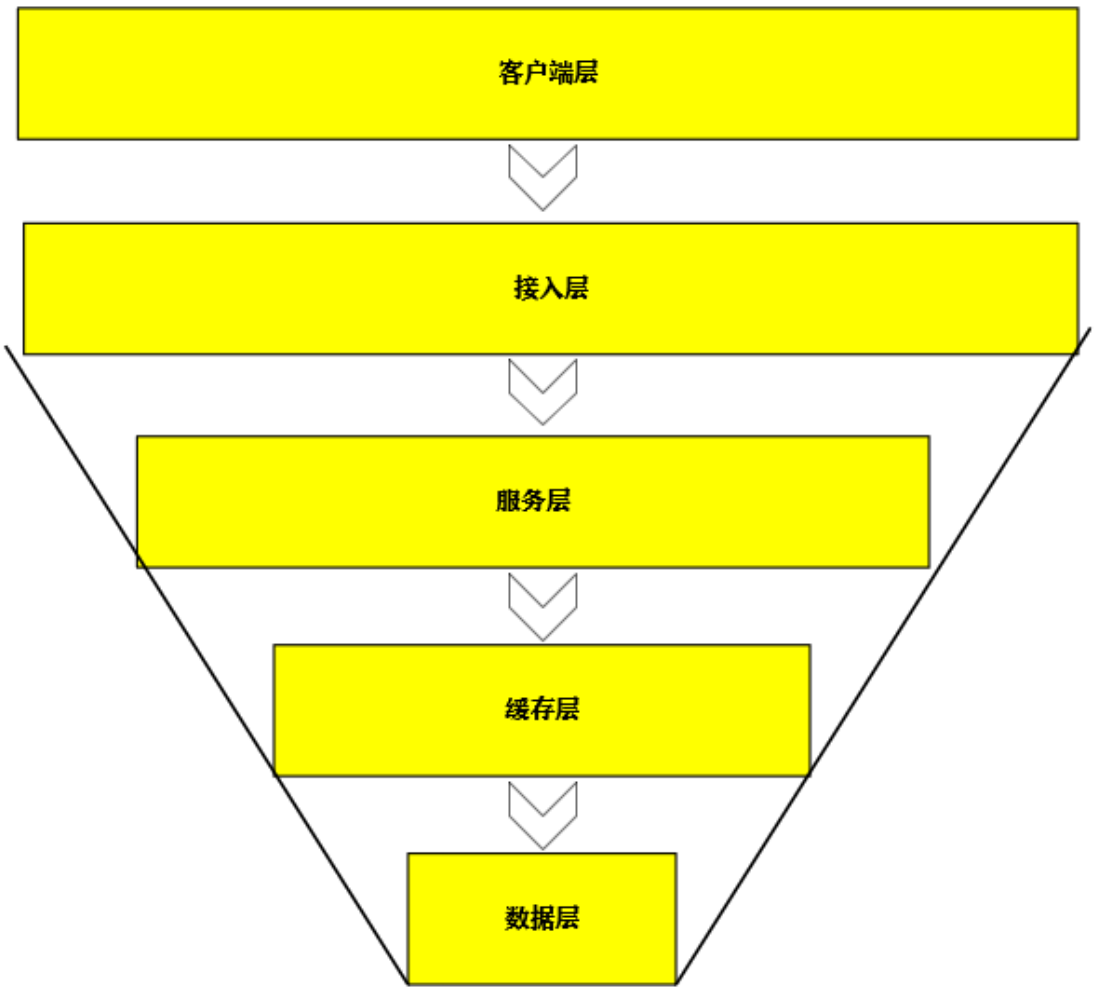
秒杀系统的分层架构

漏斗型的请求处理模型

漏斗型业务，指的是，用户的请求，从客户端到 db 层，层层递减，递减的程度视业务而定。

例如当 10w 人去抢 1 000个SKU时，db 层的请求在 1 000 数量级，这就是比较理想的模型。

如下图所示



客户层：

前端：*做静态页面的缓存，禁止重复提交。*

禁止重复提交：

秒杀开始之后，可以对用户点击后响应前按钮置灰。

为啥要禁止重复提交呢？

> 如果用户秒杀失败了，频繁重试，会加剧后端的雪崩。

接入层：

本地缓存

> 使用场景：秒杀商品动态数据的 、 读高并发

本地缓存的数据一致性策略：

策略1： 秒杀系统的商品是事先可知的，可以将参加秒杀的商品信息除了事先缓存到redis等缓存系统中，还可以事先缓存到nginx的本地缓存（三级缓存），这样可以大大的提高系统的吞吐量。

策略2： 如果三级缓存没有， 再从二级缓存读取，回写到三级缓存中。

秒杀商品动态数据的读高并发的流量架构：

10W人，1秒内响应到商品的查询信息

每个NG都是 5WQPS ， 2个NG节点，如果命中 本地缓存，则为 10WQPS

退一步说，每个NG都是 2.5WQPS ， 4个NG节点，如果命中 本地缓存，则为 10WQPS

如果 100W人同时查看一个商品的话，那么就是 20-40个NG节点，

要点：

- 读请求尽量命中Nginx的本地缓存
- Nginx 还需要对 商品详情接口做 限流保护（根据商品限流），比如限制在1WQPS， 超过后降级，返回兜底的提示信息，前端可以做出重试，或者提示稍后重试

秒杀限流

ng接入层维度限流

当用户流量远远大于ng瓶颈时，开始随机丢弃请求。

单用户限流

在网关层对下单等接口按userID限流，——》接口防刷，

> 假如限制同一个用户10分钟能下一次单，一般情况下10分钟内，商品早已经被抢光了，用户也就没有再次下单的机会了。

可以结合风控系统，在网关层把羊毛党等有问题的用户请求直接拒掉。——》恶意用户 拦截

商品维度限流

网关层除了对userID做限流外，还要从商品 skuId 维度限流。

> 在实际访问量超过预估访问量时，整体限流可以起到保护作用，避免系统被压垮。

一单秒杀开始，实际秒杀成功的用户只是库存的数量，在库存没有之后，将前端的秒杀入口关闭。

写流量处理

诀窍：

1 有效流量过滤，过滤出有效请求

2 有效流量削峰，有效请求 达到或者超出 下一层的能力瓶颈， 同步处理，变成（降级为）异步处理

3 有效流量限流，1000Qps， 接入层、服务层、数据库

秒杀两段式操作

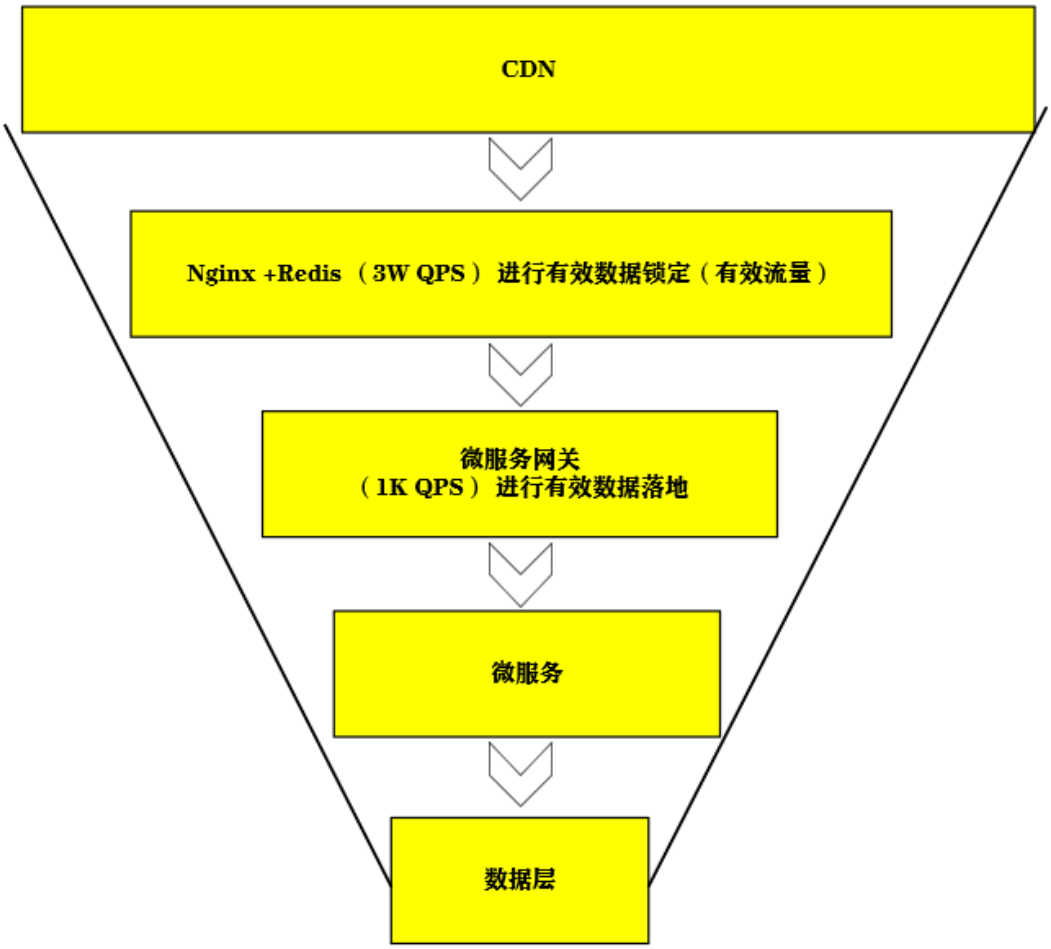
- 第一阶段为有效流量锁定， 将有效的流量识别出来，过滤掉无效的流量
- 第二阶段为有效流量落地，将有效的流量，落地到服务层

> 场景：10W人，1抢1000个商品

ng的qps为2-5w，5个ng，

redis 为qps为 3-5w，可以设置超时的时间长点，比如5s，那么10请求，3s也可以处理完成（在不做分段的情况下）

也就是：



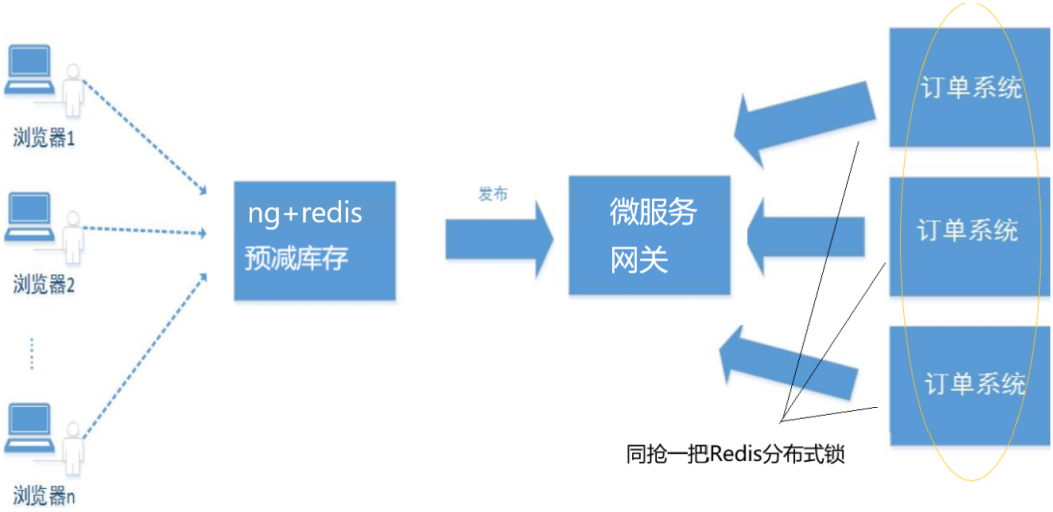
> 有效流量锁定的办法：

参加秒杀系统的商品是事先可知的，可以将参加秒杀的商品信息事先缓存到redis等缓存系统中，这样可以大大的提高系统的吞吐量，减少关系型数据库的读写压力。

秒杀的两段式操作的具体过程：

高并发操作迁移到并发量更高的nginx+redis+lua ，提交操作变成两段式：

- 第一阶段为有效流量锁定， 将有效的流量识别出来，过滤掉无效的流量 ，申请令牌，申请预减减库，申请成功之后，接入层把令牌返回客户端层，客户端层幕后发起有效请求到服务层
- 第二阶段为有效流量落地，进入服务层之后，进入消息队列，秒杀服务从消息队列拉取令牌，确认令牌，然后完成下单操作。 查库存 -> 创建订单 -> 扣减库存。通过分布式锁保障解决多个provider实例并发下单产生的超卖问题。
-



接入层安全

相信不少站长或多或少都经历过DDoS攻击（DDoS攻击是通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的），一旦你的网站被人DDoS攻击后，网站就会无法被访问了，严重时服务器都能卡死

服务层

服务治理：

注册与发现（注册中心）、统一配置（配置中心），服务流控与降级，服务监控

服务扩展与负载均衡

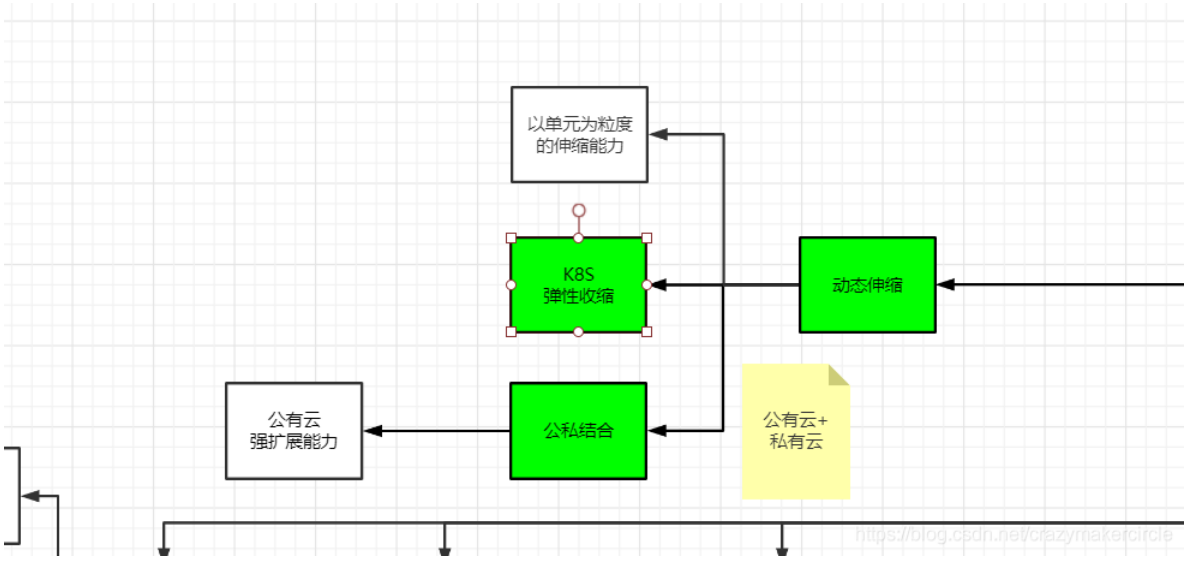
服务的可扩展，可以水平添加机器将用户请求分担到不同的机器上去。

动态扩容与缩容（自动伸缩）

一个tomcat1k，如果10w流量过来来，怎么伸缩1000个实例呢，一个一个的部署，肯定不现实。

自动的伸缩很重要。

> 内容太太多，有机会再介绍哈



异步削峰与限流保护：

异步削峰

秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。

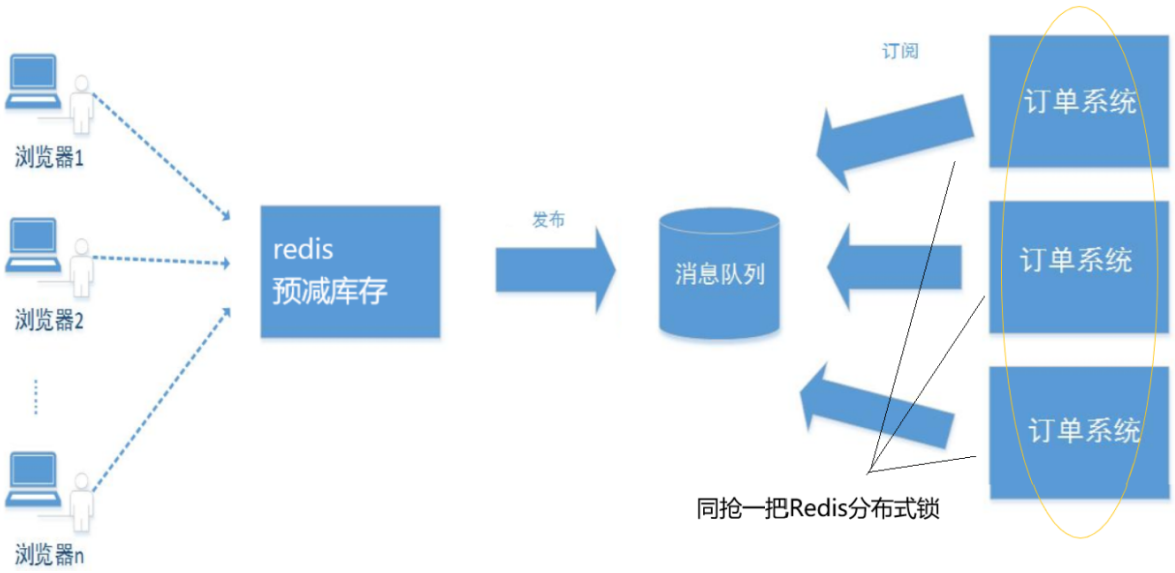
> 场景：10W人，抢10000个商品，或者100W人，抢十万商品

两段式操作的，参考一下；

- 第一阶段为有效流量锁定， 将有效的流量识别出来，过滤掉无效的流量
- 第二阶段为有效流量落地，将有效的流量，落地到服务层

有效流量也很大，如100W人，抢十万商品，如何提升下单阶段的性能呢?

消息队列削峰，其集群吞吐量可以到10Wqps



限流保护

场景：超出了用户的忍耐， 超出系统处理能力

自研组件，redis分布式限流

sentinel 进行限流

缓存层（分布式缓存）

读请求：商品信息提前缓存

参加秒杀系统的商品是事先可知的，可以将参加秒杀的商品信息事先缓存到redis等缓存系统中，这样可以大大的提高系统的吞吐量，减少关系型数据库的读写压力。

为了极致的提高速度，最好是推送到ng的本地缓存。

这中间的数据一致性，消息队列的作用，也很大。

三级缓存

为了解决以上可能出现的问题，让缓存层更稳定，健壮，我们引入三级缓存架构

- 1级为本地缓存，或者进程内的缓存（如 同时支持 Ehcache 2.x、Ehcache 3.x 、Guava、Caffeine) —— 速度快，进程内可用
- 2级为集中式缓存（如 Redis）—— 可同时为多节点提供服务
- 3级为接入层Nginx本地缓存—— 速度快，进程内可用

原则

尽量命中NG缓存， 最坏也需要命中 redis分布式缓存

方案：

可以通过消息队列，保持ng缓存与 db的数据一致性

写的流量（有效请求的过滤）：通过lua提前锁定资格

逻辑层首先应该提前锁定资格，在缓存中进行令牌的获取，如果失败的用户，快速返回，避免请求洞穿到 db。

通过lua进行令牌的发放，令牌的校验

1 有效流量过滤，过滤出有效请求

2 有效流量削峰，有效请求 达到或者超出 下一层的能力瓶颈， 同步处理，变成（降级为）异步处理

3 有效流量限流，1000Qps， 接入层、服务层、数据库

涉及的知识点:

涉及到redis+lua的设计/开发/调试

涉及到nginx+lua的设计/开发/调试

nginx+lua VS redis +lua

流量架构

亿级用户量的压力预估

这个假设这个网站预估的用户数是10000万，那么根据28法则，每天会来访问这个网站的用户占到20%，也就是2000万用户每天会过来访问。

【通过用户量来推算PV】

公式：(总用户数 * 20%) / 每天的大致点击次数（淘宝经验30-50次）= pv数

问：用户数是1000万，pv量是多少?

答：10000万 * 20% * 30= 60000万

> 通常假设平均每个用户每次过来会有30次的点击，那么总共就有60000万的点击（PV）。

【PV推算QPS的公式】

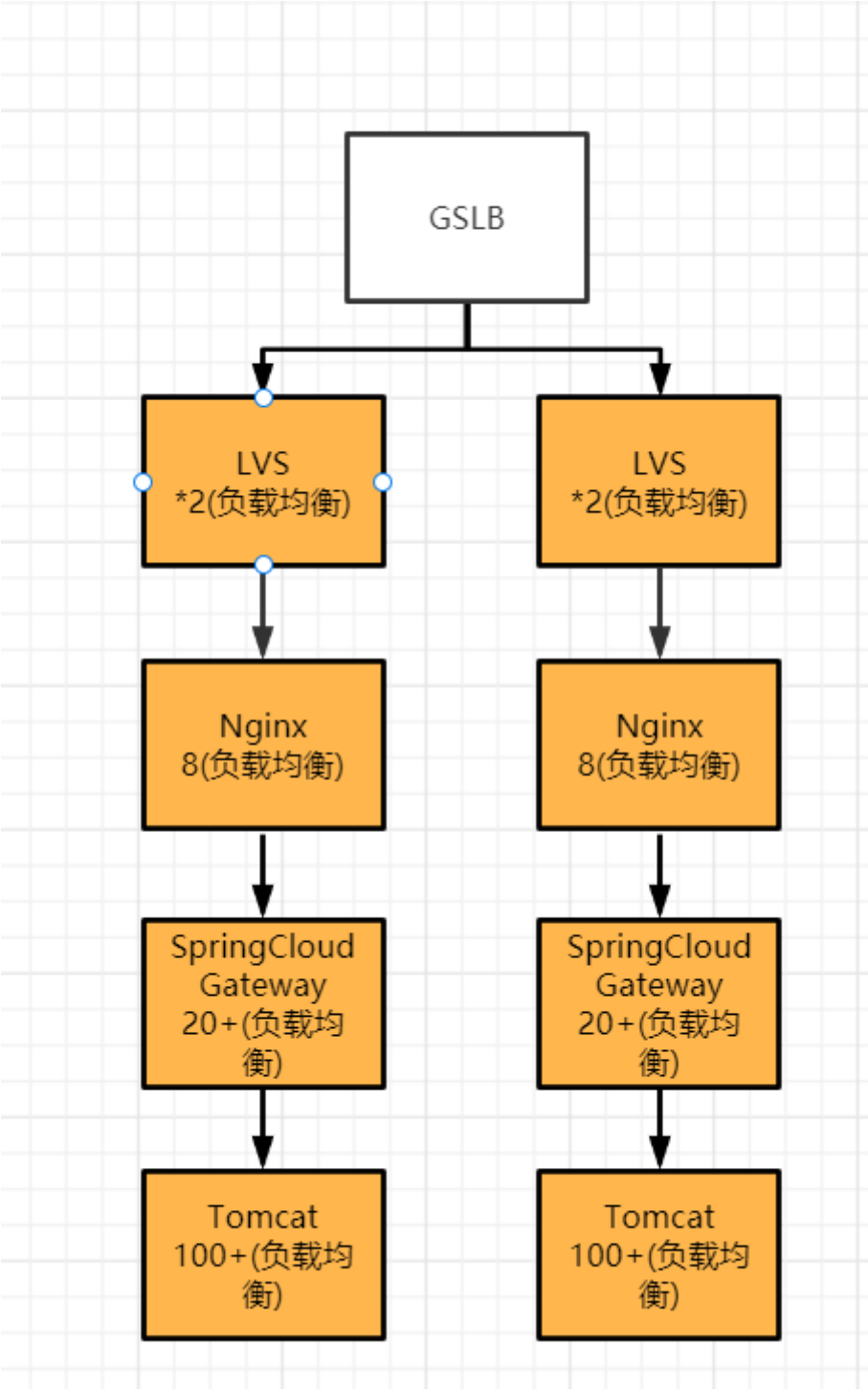
公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

问：5小时内会有48万点击， 多少QPS?
答：60000 W * 0.8 / 5 * 3600 = 27000 (QPS)

【加上冗余系统 】

27000 (QPS) * 4 =108000 (QPS)

各层的部署架构



基于Netty+Zk的分布式性能测试框架

数据存储 架构

数据层

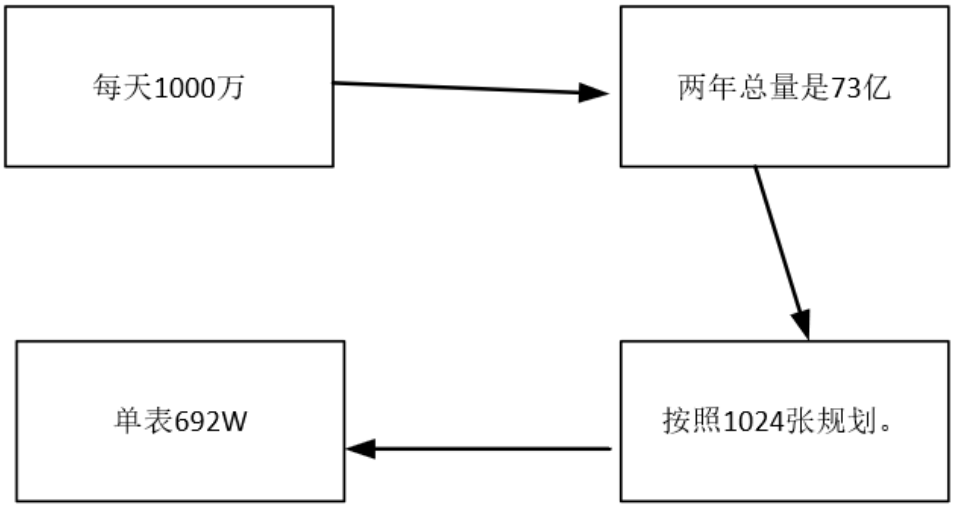
数据库可扩展，支持分库分表，对于用户的请求，映射到不同的数据库，减少单台数据库的压力。

百亿级库表架构设计

表的数据量规划

即便按照**每天1000万，2年内保持稳定的要求**，进行表的数据量规划：

> 两年总量是73亿（每天1000万*730天），假设每张表的标准值为500W，表的数据量平均是1460张，换成2的幂，比较接近1024张，或者说，单表692W（692=73亿/1024），也是可以接受的，所以就按照1024张规划。



库的规划

即便按照QPS峰值**5W的要求**，进行表的库的规划：

> 每个库正常承载的写入并发量是 1500，那么32个库就可以承载32 * 1500 = 48000 的写并发

悲观一点，如果每秒写入超过 5WQPS，可以通过**MQ削峰+批写入**的降级策略，MQ的写入吞吐量，可以轻松到达10W级别。

总的规划

利用 32 * 32 来分库分表，即分为32 个库，每个库里一个表分为 32 张表。一共就是 1024 张表。根据某个 id 先根据 32 取模路由到库，再根据32取模路由到库里的表。

> 分库、分表实操

异地多活

对于结构化 db 的要求，在异地多活的场景下，需要能够保证数据的最终一致性性，并且在尽可能短的时间内。 这个异步复制保证一致性的时间，就是切流之后，部分不可用的时间。

同一个IDC机房

需要有数据的主备，并且在主不可用的场景下，能够进行备库的切换

并且需要保障主备的数据一致性。

其他的内容

数据一致性理论与实操

分布式事务实操

热点数据的更新问题

360度监控实操

一致性

db的副本之间的数据一致性

缓存与db的数据一致性：

含二级缓存与db的数据一致性：

含三级缓存与二级缓存、一级缓存的数据一致性

超卖问题

分布式锁

分布式的分段锁