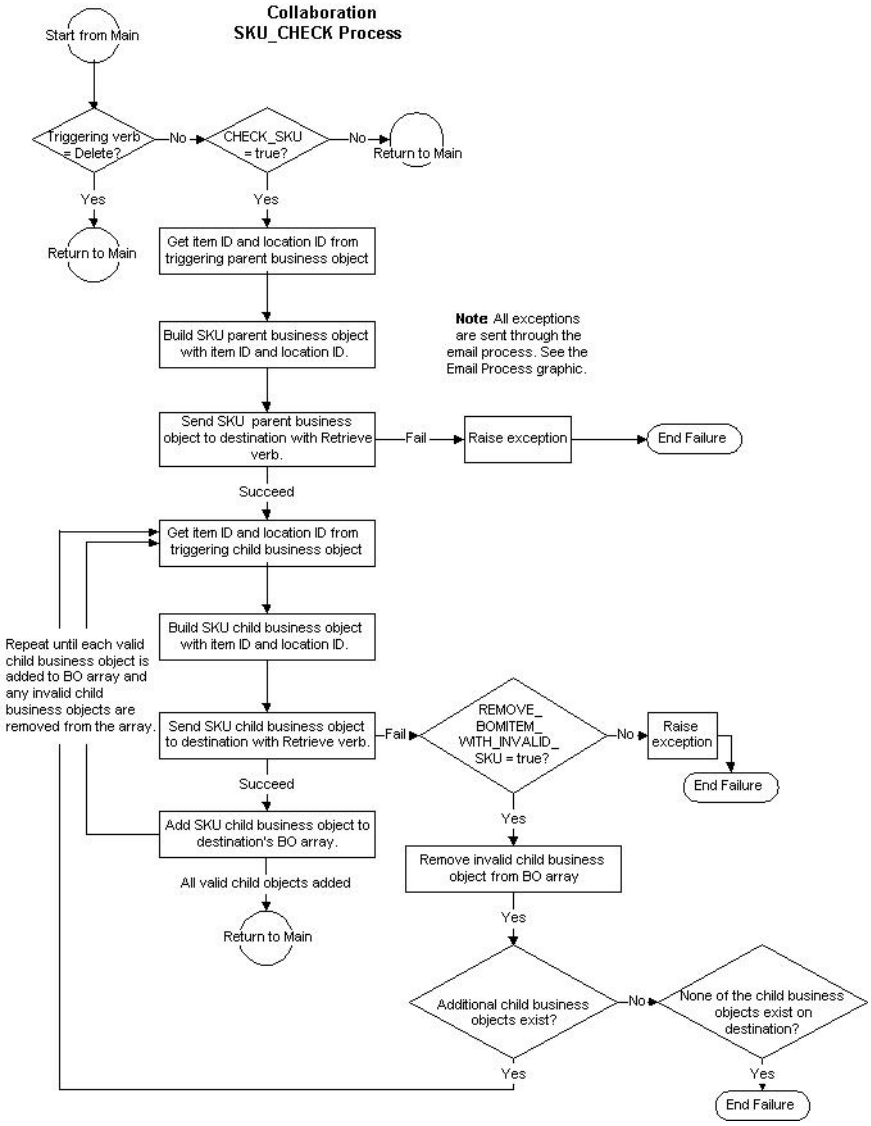
sku和高并发处理

商品处理

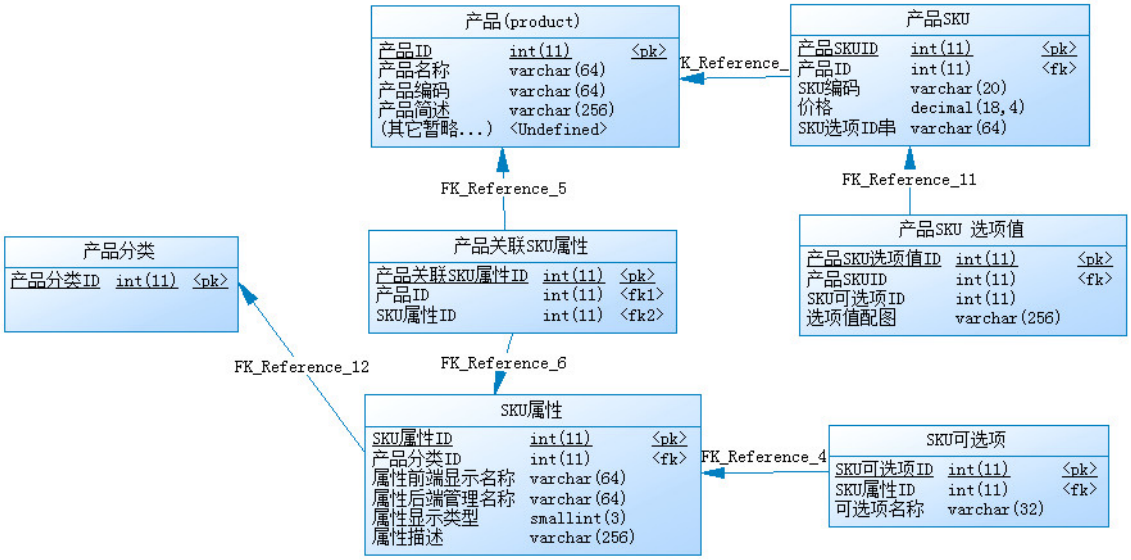
sku是什么?

SKU=Stock Keeping Unit(库存量单位?，即库存进出计量的单位，可以是以件，盒，托盘等为单位。SKU这是对于大型连锁超市DC（配送中心）物流管理的一个必要的方法。当下已经被我们引申为产品统一编号的简称，每种产品均对应有唯一的SKU号
针对电商而言，SKU有另外的注解

1、SKU是指一款商品，每款都有出现一个SKU，便于电商品牌识别商品。
2、一款商品多色，则是有多个SKU，例：一件衣服，有红色、白色、蓝色，则SKU编码也不相同，如相同则会出现混淆，发错货。



sku表的设计原理图：



电商中的sku表

Goods（商品表）

商品ID	商品名称	商品的货号	模型ID	销售价格	市场价格	成本价格	上架时间	下架时间	创建时间	库存	原图	宣传图片	0正常 1已删 2下架 3申请上架	商品描述
goods_id	name	goods_no	model_id	sell_price	market_price	cost_price	up_time	down_time	create_time	store_nums	img	ad_img	is_del	description
产品搜索词库,逗号分隔	重量	积分	计量单位	品牌ID	浏览次数	收藏次数	排序	序列化存储规则	经验	评论次数	销售	评分总数	卖家ID	共享商品 0不共享 1共享
search_words	weight	point	unit	brand_id	visit	favorite	sort	spec_array	exp	comments	sale	grade	seller_id	is_shar

products （货品表）

自增ID	货品ID	货品的货号(以商品的货号加横线加数字组成)	json规格数据	库存	市场价格	销售价格	成本价格	重量
id	goods_id	products_no	spec_array	store_nums	market_price	sell_price	cost_price	weight

spec （规格表）

自增ID	规格名称	规格值	显示类型 1文字 2图片	备注说明	是否删除1删除	商家ID
id	name	value	type	note	is_del	seller_id

category （产品分类表）

分类ID	分类名称	父分类ID	排序	首页是否显示1显示 0不显示	SEO关键词和检索关键词	SEO描述	SEO标题title	商家ID
category_id	name	parent_id	sort	visibility	keywords	descript	title	seller_id

category_extend （商品扩展分类表）

自增ID	商品ID	商品分类ID
id	goods_id	category_id

attribute （ 属性表）

属性ID	模型ID	输入控件的类型,1:单选,2:复选,3:下拉,4:输入框	名称	属性值(逗号分隔)	是否支持搜索0不支持1支持
id	model_id	type	name	value	search

goods_attribute （属性值表）

自增ID	商品ID	属性ID	属性值	模型ID	排序
id	goods_id	attribute_id	attribute_value	model_id	order

model （模型表）

模型ID	模型名称	规格ID逗号分隔
id	name	spec_ids

秒杀高并发

大规模并发带来的挑战：

1 、请求接口的合理设计：一个秒杀或者抢购页面，通常分为2个部分，一个是静态的HTML等内容，另一个就是参与秒杀的Web后台请求接口。 通常静态HTML等内容，是通过CDN的部署，一般压力不大，核心瓶颈实际上在后台请求接口上。这个后端接口，必须能够支持高并发请求，同时，非常 重要的一点，必须尽可能“快”，在最短的时间里返回用户的请求结果。为了实现尽可能快这一点，接口的后端存储使用内存级别的操作会更好一点。仍然直接面向 MySQL之类的存储是不合适的，如果有这种复杂业务的需求，都建议采用异步写入。
当然，也有一些秒杀和抢购采用“滞后反馈”，就是说秒杀当下不知道结果，一段时间后才可以从页面中看到用户是否秒杀成功。但是，这种属于“偷懒”行为，同时给用户的体验也不好，容易被用户认为是“暗箱操作”。

2、 高并发的挑战：一定要“快”我们通常衡量一个Web系统的吞吐率的指标是QPS（Query Per Second，每秒处理请求数），解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为100ms，同时， 系统内有20台Apache的Web服务器，配置MaxClients为500个（表示Apache的最大连接数目）。

那么，我们的Web系统的理论峰值QPS为（理想化的计算方式）：

20*500/0.1 = 100000 （10万QPS）

咦？我们的系统似乎很强大，1秒钟可以处理完10万的请求，5w/s的秒杀似乎是“纸老虎”哈。实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

就Web服务器而言，Apache打开了越多的连接进程，CPU需要处理的上下文切换也越多，额外增加了CPU的消耗，然后就直接导致平均响应时间 增加。因此上述的MaxClient数目，要根据CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过Apache自带的abench来测试一 下，取一个合适的值。然后，我们选择内存操作级别的存储的Redis，在高并发的状态下，存储的响应时间至关重要。网络带宽虽然也是一个因素，不过，这种 请求数据包一般比较小，一般很少成为请求的瓶颈。负载均衡成为系统瓶颈的情况比较少，在这里不做讨论哈。

那么问题来了，假设我们的系统，在5w/s的高并发状态下，平均响应时间从100ms变为250ms（实际情况，甚至更多）：

20*500/0.25 = 40000 （4万QPS）

于是，我们的系统剩下了4w的QPS，面对5w每秒的请求，中间相差了1w。

然后，这才是真正的恶梦开始。举个例子，高速路口，1秒钟来5部车，每秒通过5部车，高速路口运作正常。突然，这个路口1秒钟只能通过4部车，车流量仍然依旧，结果必定出现大塞车。（5条车道忽然变成4条车道的感觉）

同理，某一个秒内，20*500个可用连接进程都在满负荷工作中，却仍然有1万个新来请求，没有连接进程可用，系统陷入到异常状态也是预期之内。其实在正常的非高并发的业务场景中，也有类似的情况出现，某个业务请求接口出现问题，响应时间极慢，将整个Web请求响应时间拉得很长，逐渐将Web服务器的可用连接数占满，其他正常的业务请求，无连接进程可用。

更可怕的问题是，是用户的行为特点，系统越是不可用，用户的点击越频繁，恶性循环最终导致“雪崩”（其中一台Web机器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环），将整个Web系统拖垮。 多个人访问同一个一个文件，处理高并发：承载更多的并发，考虑内存的问题，降低运行成本，从而并发量更大，一天访问次数（访问量）pv，用户访问（uv），计算独立Ip是正常用户7-8倍，统计（网站统计，页面统计），活跃度App100万的访问量，统计独立Ip大约是5倍。防止并发：消息队列，php push pop shift mysql 锁表 行锁 nosql redis memcacheq mongo ;大访问量（静态优化）;优化从硬件开始，cpu 内存，负载均衡：多台服务器，mysql 分表（加速查询）分区（水平分区：将数据转移到其他服务器，业务冲突时，把一条数据分割成几条数据实现）

3、 重启与过载保护：如果系统发生“雪崩”，贸然重启服务，是无法解决问题的。最常见的现象是，启动起来后，立刻挂掉。这个时候，最好在入口层将流量拒绝，然后再将重启。如果是redis/memcache这种服务也挂了，重启的时候需要注意“预热”，并且很可能需要比较长的时间。

秒杀和抢购的场景，流量往往是超乎我们系统的准备和想象的。这个时候，过载保护是必要的。如果检测到系统满负载状态，拒绝请求也是一种保护措施。在 前端设置过滤是最简单的方式，但是，这种做法是被用户“千夫所指”的行为。更合适一点的是，将过载保护设置在CGI入口层，快速将客户的直接请求返回。

注：秒杀和抢购收到了“海量”的请求，实际上里面的水分是很大的。不少用户，为了“抢“到商品，会使用“刷票工具”等类型的辅助工具，帮助他们发送尽可 能多的请求到服务器。还有一部分高级用户，制作强大的自动请求脚本。这种做法的理由也很简单，就是在参与秒杀和抢购的请求中，自己的请求数目占比越多，成功的概率越高。

作弊的手段：进攻与防守:

A： 同一个账号，一次性发出多个请求

Question? 部分用户通过浏览器的插件或者其他工具，在秒杀开始的时间里，以自己的账号，一次发送上百甚至更多的请求。实际上，这样的用户破坏了秒杀和抢购的公平性。

这种请求在某些没有做数据安全处理的系统里，也可能造成另外一种破坏，导致某些判断条件被绕过。例如一个简单的领取逻辑，先判断用户是否有参与记 录，如果没有则领取成功，最后写入到参与记录中。这是个非常简单的逻辑，但是，在高并发的场景下，存在深深的漏洞。多个并发请求通过负载均衡服务器，分配 到内网的多台Web服务器，它们首先向存储发送查询请求，然后，在某个请求成功写入参与记录的时间差内，其他的请求获查询到的结果都是“没有参与记录”。 这里，就存在逻辑判断被绕过的风险。

Settled(解决)： 在程序入口处，一个账号只允许接受1个请求，其他请求过滤。不仅解决了同一个账号，发送N个请求的问题，还保证了后续的逻辑流程的安全。实现方案， 可以通过Redis这种内存缓存服务，写入一个标志位（只允许1个请求写成功，结合watch的乐观锁的特性），成功写入的则可以继续参加。或者，自己实现一个服务，将同一个账号的请求放入一个队列中，处理完一个，再处理下一个。

B： 多个账号，一次性发送多个请求

Question? 很多公司的账号注册功能，在发展早期几乎是没有限制的，很容易就可以注册很多个账号。因此，也导致了出现了一些特殊的工作室，通过编写自动注册脚 本，积累了一大批“僵尸账号”，数量庞大，几万甚至几十万的账号不等，专门做各种刷

的行为（这就是微博中的“僵尸粉“的来源）。举个例子，例如微博中有转 发抽奖的活动，如果我们使用几万个“僵尸号”去混进去转发，这样就可以大大提升我们中奖的概率。
这种账号，使用在秒杀和抢购里，也是同一个道理。例如，iPhone官网的抢购，火车票黄牛党。

Settled(解决)： 这种场景，可以通过检测指定机器IP请求频率就可以解决，如果发现某个IP请求频率很高，可以给它弹出一个验证码或者直接禁止它的请求：
弹出验证码，最核心的追求，就是分辨出真实用户。因此，大家可能经常发现，网站弹出的验证码，有些是“鬼神乱舞”的样子，有时让我们根本无法看清。他们这样做的原因，其实也是为了让验证码的图片不被轻易识别，因为强大的“自动脚本”可以通过图片识别里面的字符，然后让脚本自 动填写验证码。实际上，有一些非常创新的验证码，效果会比较好，例如给你一个简单问题让你回答，或者让你完成某些简单操作（例如百度贴吧的验证码）。
直接禁止IP，实际上是有些粗暴的，因为有些真实用户的网络场景恰好是同一出口IP的，可能会有“误伤“。但是这一个做法简单高效，根据实际场景使用可以获得很好的效果。

高并发下的数据安全如何保证？
A 超发的原因：自由主题假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。（同文章前面说的场景）。

B 悲观锁思路:解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。 悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。
虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待 “锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗 尽，系统陷入异常。

C FIFO队列思路：我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬 间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌 入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。D 乐观锁思路：。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号 （Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。 这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。
Web系统大规模并发——电商秒杀与抢购 - 徐汉彬Hansion - 技术行者
有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。

【总结】：互联网正在高速发展，使用互联网服务的用户越多，高并发的场景也变得越来越多。电商秒杀和抢购，是两个比较典型的互联网高并发场景。虽然我们解决问题的具体技术方案可能千差万别，但是遇到的挑战却是相似的，因此解决问题的思路也异曲同工。
