**高并发设计方案一**

**设计思路：**

（1） 服务器：

均衡负载(如：nginx，阿里云SLB)

资源监控

分布式

（2） 数据库

主从分离，集群

DBA 表优化，索引优化，等

分布式

（3） nosql

redis 铭记：Redis只放热数据

主从分离，集群

mongodb

主从分离，集群

memcache

主从分离，集群

（4） cdn

html

css

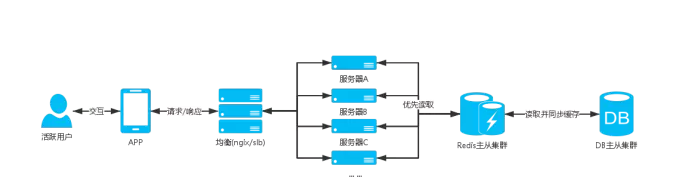
js

image

**1、通用方案**

日用户流量大，但是比较分散，偶尔会有用户高聚集的情况；

场景： 用户签到，用户中心，用户订单，等

服务器架构图：   
   
说明：

场景中的这些业务基本是用户进入APP后会操作到的，除了活动日(618，双11，等)，这些业务的用户量都不会高聚集，同时这些业务相关的表都是大数据表，业务多是查询操作，所以我们需要减少用户直接命中DB的查询；优先查询缓存，如果缓存不存在，再进行DB查询，将查询结果缓存起来。

更新用户相关缓存需要分布式存储，比如使用用户ID进行hash分组，把用户分布到不同的缓存中，这样一个缓存集合的总量不会很大，不会影响查询效率。

**使用场景：**

（1） 用户签到获取积分

计算出用户分布的key,redis hash中查找用户今日签到信息

如果查询到签到信息，返回签到信息

如果没有查询到，DB查询今日是否签到过，如果已经签到过，就把签到信息同步redis缓存。

如果DB中也没有查询到今日的签到记录，就进行签到逻辑，操作DB添加今日签到记录，添加签到积分(这整个DB操作是一个事务)

缓存签到信息到redis，返回签到信息

注意这里会有并发情况下的逻辑问题，如：一天签到多次，发放多次积分给用户。

我的博文[《大话程序猿眼里的高并发》](http://mp.weixin.qq.com/s?__biz=MzA5ODM5MDU3MA==&mid=2650862038&idx=1&sn=42fa92e2ff9f61a10af6ef7b360e94b9&chksm=8b661693bc119f85a265ec4ad2de599c5268adc9f694631a6946e3287dfa713b5fc2702a1ff7&scene=21" \l "wechat_redirect" \t "_blank)有相关的处理方案。

（2） 用户订单

这里我们只缓存用户第一页的订单信息，一页40条数据，用户一般也只会看第一页的订单数据

用户访问订单列表，如果是第一页读缓存，如果不是读DB

计算出用户分布的key,redis hash中查找用户订单信息

如果查询到用户订单信息，返回订单信息

如果不存在就进行DB查询第一页的订单数据，然后缓存redis，返回订单信息

***如上所示，缓存用户第一页的订单数据进redis，那如果用户又添加了一个订单，这缓存的第一页数据如何更新呢？***

***两种方案：***

***1.当用户有新订单的时候删除缓存，等到用户进入订单列表时优先读缓存，缓存无数据时就从DB读出最新的订单缓存到Redis中。***

***2.将需要更新订单的用户加入【更新订单用户队列】中，然后多线程程序去消耗队列，更新用户订单缓存。***

***根据你的实际的业务情景去选择吧***

（3） 用户中心

计算出用户分布的key,redis hash中查找用户订单信息

如果查询到用户信息，返回用户信息

如果不存在进行用户DB查询，然后缓存redis，返回用户信息

（4） 其他业务

上面例子多是针对用户存储缓存，如果是公用的缓存数据需要注意一些问题，如下

注意公用的缓存数据需要考虑并发下的可能会导致大量命中DB查询，可以使用管理后台更新缓存，或者DB查询的锁住操作。

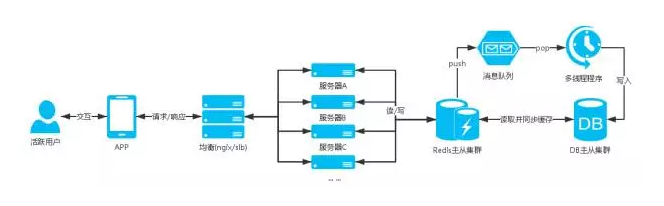
我的博文[[大话Redis进阶]](https://segmentfault.com/q/1010000006222395)对更新缓存问题和推荐方案的分享。

以上例子是一个相对简单的高并发架构，并发量不是很高的情况可以很好的支撑，但是随着业务的壮大，用户并发量增加，我们的架构也会进行不断的优化和演变，比如对业务进行服务化，每个服务有自己的并发架构，自己的均衡服务器，分布式数据库，nosql主从集群，如：用户服务、订单服务；

**消息队列**

**秒杀、秒抢**等活动业务，用户在瞬间涌入产生高并发请求

**场景：定时领取红包，等**

服务器架构图：   
   
说明：

场景中的定时领取是一个高并发的业务，像秒杀活动用户会在到点的时间涌入，DB瞬间就接受到一记暴击，hold不住就会宕机，然后影响整个业务；

像这种不是只有查询的操作并且会有高并发的插入或者更新数据的业务，前面提到的通用方案就无法支撑，并发的时候都是直接命中DB；

设计这块业务的时候就会使用消息队列，可以将参与用户的信息添加到消息队列中，然后再写个多线程程序去消耗队列，给队列中的用户发放红包；

方案如：

**定时领取红包**

一般习惯使用 redis的 list

当用户参与活动，将用户参与信息push到队列中

然后写个多线程程序去pop数据，进行发放红包的业务

这样可以支持高并发下的用户可以正常的参与活动，并且避免数据库服务器宕机的危险

附加：

通过消息队列可以做很多的服务。

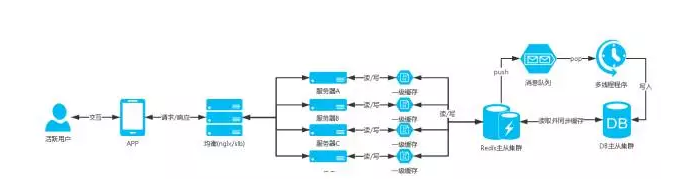
如：定时短信发送服务，使用sset(sorted set)，发送时间戳作为排序依据，短信数据队列根据时间升序，然后写个程序定时循环去读取sset队列中的第一条，当前时间是否超过发送时间，如果超过就进行短信发送。

**一级缓存**

高并发请求连接缓存服务器，超出服务器能够接收的请求连接量，部分用户出现建立连接超时无法读取到数据的问题；

因此需要有个方案当高并发时候时候可以减少命中缓存服务器；

这时候就出现了一级缓存的方案，一级缓存就是使用站点服务器缓存去存储数据，注意只存储部分请求量大的数据，并且缓存的数据量要控制，不能过分使用站点服务器的内存，从而影响了站点应用程序的正常运行，一级缓存需要设置秒单位的过期时间，具体时间根据业务场景设定，目的是当有高并发请求的时候可以让数据的获取命中到一级缓存，而不用连接缓存nosql数据服务器，减少nosql数据服务器的压力；

比如**APP首屏商品数据接口**，这些数据是公共的，不会针对用户自定义，而且这些数据不会频繁的更新，像这种接口的请求量比较大就可以加入一级缓存；   
服务器架构图：   


合理和规范使用nosql缓存数据库，根据业务拆分缓存数据库的集群，这样基本可以很好支持业务，一级缓存毕竟是使用站点服务器缓存，所以还是要善用。

**静态化数据**

高并发请求数据不变化的情况下，如果可以不请求自己的服务器获取数据，那就可以减少服务器的资源压力。

对于更新频繁度不高，并且数据允许短时间内的延迟，可以通过数据静态化成JSON、XML、HTML等数据文件上传CDN，在拉取数据的时候优先到CDN拉取，如果没有获取到数据再从缓存数据库中获取，当管理人员操作后台编辑数据再重新生成静态文件上传同步到CDN，这样在高并发的时候可以使数据的获取命中在CDN服务器上。

CDN节点同步有一定的延迟性，所以找一个靠谱的CDN服务器商也很重要

对于单页面访问量巨大的可以将其专门做成静态页面

**主从分离、分库分表**

读写分离是大家看到的第二个高并发的架构了，也很常规，因为一般情况下读比写要多得多，所以数据库的主库写，从库们提供读操作，一下就把数据库的并发性能提高了。

如果还不够，那么分库分表把，把数据分到各个数据库的各个机器上，进一步的减少单台机器的压力，从而达到高并发的目的。

如果是分库分表，有时候使用的就是哈希技术了，以某个字段哈希一下然后来分库分表，读写分离的读操作，基本也是通过哈希技术把读落到不同的机器上去减轻单机压力。

<http://blog.csdn.net/u013372487/article/details/51612161>

**合理数据结构->优化代码->外部通用方法**

解决性能问题，首先要有合理的数据结构，之后要优化代码，只有在最优化之后再考虑外部通用方法如缓存等，因为一开始使用了缓存等方法以后就没有太多进一步优化空间了。

**其他方案**

对于**更新频繁度不高的数据**，APP 和 PC浏览器可以缓存数据到本地，然后每次请求接口的时候上传当前缓存数据的版本号，服务端接收到版本号判断版本号与最新数据版本号是否一致，如果不一样就进行最新数据的查询并返回最新数据和最新版本号，如果一样就返回状态码告知数据已经是最新。减少服务器压力：资源、带宽。

**代码才是关键，架构都是锦上添花的东西，既然是锦上添花的，必然坑多，没有什么捷径。**

**优化方向：**

**（1）将请求尽量拦截在系统上游**（不要让锁冲突落到数据库上去）。传统秒杀系统之所以挂，请求都压倒了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小。以12306为例，一趟火车其实只有2000张票，200w个人来买，基本没有人能买成功，请求有效率为0。

**（2）充分利用缓存**，秒杀买票，这是一个典型的读多写少的应用场景，大部分请求是车次查询，票查询，下单和支付才是写请求。一趟火车其实只有2000张票，200w个人来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%，非常适合使用缓存来优化。

**常见秒杀架构：**

   
（1）浏览器端，最上层，会执行到一些JS代码   
（2）站点层，这一层会访问后端数据，拼html页面返回给浏览器   
（3）服务层，向上游屏蔽底层数据细节，提供数据访问   
（4）数据层，最终的库存是存在这里的，mysql是一个典型（当然还有会缓存）

**各层次优化细节：**

**第一层，客户端怎么优化（浏览器层，APP层）**

问大家一个问题，大家都玩过微信的摇一摇抢红包对吧，每次摇一摇，就会往后端发送请求么？回顾我们下单抢票的场景，点击了“查询”按钮之后，系统那个卡呀，进度条涨的慢呀，作为用户，我会不自觉的再去点击“查询”，对么？继续点，继续点，点点点。。。有用么？平白无故的增加了系统负载，一个用户点5次，80%的请求是这么多出来的，怎么整？

（a）产品层面，用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求；

（b）JS层面，限制用户在x秒之内只能提交一次请求；

（c）APP层面，可以做类似的事情，虽然你疯狂的在摇微信，其实x秒才向后端发起一次请求。

这就是所谓的“将请求尽量拦截在系统上游”，越上游越好，浏览器层，APP层就给拦住，这样就能挡住80%+的请求，这种办法只能拦住普通用户（但99%的用户是普通用户）对于群内的高端程序员是拦不住的。firebug一抓包，http长啥样都知道，js是万万拦不住程序员写for循环，调用http接口的，这部分请求怎么处理？

**第二层，站点层面的请求拦截**

怎么拦截？怎么防止程序员写for循环调用，有去重依据么？ip？cookie-id？…想复杂了，这类业务都需要登录，用uid即可。在站点层面，对uid进行请求计数和去重，甚至不需要统一存储计数，直接站点层内存存储（这样计数会不准，但最简单）。一个uid，5秒只准透过1个请求，这样又能拦住99%的for循环请求。   
5s只透过一个请求，其余的请求怎么办？缓存，页面缓存，同一个uid，限制访问频度，做页面缓存，x秒内到达站点层的请求，均返回同一页面。同一个item的查询，例如车次，做页面缓存，x秒内到达站点层的请求，均返回同一页面。如此限流，既能保证用户有良好的用户体验（没有返回404）又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。   
页面缓存不一定要保证所有站点返回一致的页面，直接放在每个站点的内存也是可以的。优点是简单，坏处是http请求落到不同的站点，返回的车票数据可能不一样，这是站点层的请求拦截与缓存优化。

**第三层 服务层来拦截（反正就是不要让请求落到数据库上去）**

服务层怎么拦截？大哥，我是服务层，我清楚的知道小米只有1万部手机，我清楚的知道一列火车只有2000张车票，我透10w个请求去数据库有什么意义呢？没错，请求队列！   
对于写请求，做请求队列，每次只透有限的写请求去数据层（下订单，支付这样的写业务）   
1w部手机，只透1w个下单请求去db   
3k张火车票，只透3k个下单请求去db   
如果均成功再放下一批，如果库存不够则队列里的写请求全部返回“已售完”。

对于读请求，怎么优化？cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的。如此限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99.9%的请求被拦住了。

当然，还有业务规则上的一些优化。回想12306所做的，分时分段售票，原来统一10点卖票，现在8点，8点半，9点，…每隔半个小时放出一批：将流量摊匀。   
其次，数据粒度的优化：你去购票，对于余票查询这个业务，票剩了58张，还是26张，你真的关注么，其实我们只关心有票和无票？流量大的时候，做一个粗粒度的“有票”“无票”缓存即可。   
第三，一些业务逻辑的异步：例如下单业务与 支付业务的分离。这些优化都是结合 业务 来的，我之前分享过一个观点“一切脱离业务的架构设计都是耍流氓”架构的优化也要针对业务。

***问题1、按你的架构，其实压力最大的反而是站点层，假设真实有效的请求数有1000万，不太可能限制请求连接数吧，那么这部分的压力怎么处理？***

答：每秒钟的并发可能没有1kw，假设有1kw，解决方案2个：

（1）站点层是可以通过加机器扩容的，最不济1k台机器来呗。

（2）如果机器不够，抛弃请求，抛弃50%（50%直接返回稍后再试），原则是要保护系统，不能让所有用户都失败。

*问题2、“控制了10w个肉鸡，手里有10w个uid，同时发请求” 这个问题怎么解决哈？*

答：上面说了，服务层写请求队列控制

问题3：限制访问频次的缓存，是否也可以用于搜索？例如A用户搜索了“手机”，B用户搜索“手机”，优先使用A搜索后生成的缓存页面？

答：这个是可以的，这个方法也经常用在“动态”运营活动页，例如短时间推送4kw用户app-push运营活动，做页面缓存。

问题4：如果队列处理失败，如何处理？肉鸡把队列被撑爆了怎么办？

答：处理失败返回下单失败，让用户再试。队列成本很低，爆了很难吧。最坏的情况下，缓存了若干请求之后，后续请求都直接返回“无票”（队列里已经有100w请求了，都等着，再接受请求也没有意义了）

问题5：站点层过滤的话，是把uid请求数单独保存到各个站点的内存中么？如果是这样的话，怎么处理多台服务器集群经过负载均衡器将相同用户的响应分布到不同服务器的情况呢？还是说将站点层的过滤放到负载均衡前？

答：可以放在内存，这样的话看似一台服务器限制了5s一个请求，全局来说（假设有10台机器），其实是限制了5s 10个请求，解决办法：

1）加大限制（这是建议的方案，最简单）

2）在nginx层做7层均衡，让一个uid的请求尽量落到同一个机器上

问题6：服务层过滤的话，队列是服务层统一的一个队列？还是每个提供服务的服务器各一个队列？如果是统一的一个队列的话，需不需要在各个服务器提交的请求入队列前进行锁控制？

答：可以不用统一一个队列，这样的话每个服务透过更少量的请求（总票数/服务个数），这样简单。统一一个队列又复杂了。

问题7：秒杀之后的支付完成，以及未支付取消占位，如何对剩余库存做及时的控制更新？

答：数据库里一个状态，未支付。如果超过时间，例如45分钟，库存会重新会恢复（大家熟知的“回仓”），给我们抢票的启示是，开动秒杀后，45分钟之后再试试看，说不定又有票哟~

问题8：不同的用户浏览同一个商品 落在不同的缓存实例显示的库存完全不一样 请问老师怎么做缓存数据一致或者是允许脏读？

答：目前的架构设计，请求落到不同的站点上，数据可能不一致（页面缓存不一样），这个业务场景能接受。但数据库层面真实数据是没问题的。

问题9：就算处于业务把优化考虑“3k张火车票，只透3k个下单请求去db”那这3K个订单就不会发生拥堵了吗？

答：（1）数据库抗3k个写请求还是ok的；（2）可以数据拆分；（3）如果3k扛不住，服务层可以控制透过去的并发数量，根据压测情况来吧，3k只是举例；

问题10；如果在站点层或者服务层处理后台失败的话，需不需要考虑对这批处理失败的请求做重放？还是就直接丢弃？

答：别重放了，返回用户查询失败或者下单失败吧，架构设计原则之一是“fail fast”。

问题11.对于大型系统的秒杀，比如12306，同时进行的秒杀活动很多，如何分流？

答：垂直拆分

问题12、额外又想到一个问题。这套流程做成同步还是异步的？如果是同步的话，应该还存在会有响应反馈慢的情况。但如果是异步的话，如何控制能够将响应结果返回正确的请求方？

答：用户层面肯定是同步的（用户的http请求是夯住的），服务层面可以同步可以异步。

问题13、秒杀群提问：减库存是在那个阶段减呢？如果是下单锁库存的话，大量恶意用户下单锁库存而不支付如何处理呢？

答：数据库层面写请求量很低，还好，下单不支付，等时间过完再“回仓”，之前提过了。

用户提交订单后减库存，加入队列，若加入成功跳转到支付，支付成功移出队列，记入数据库

（队列可记录库存数 并且 队列限制插入订单数，若插入数大于库存数插入失败，插入太多也没有意义）

（插入失败原因分两块 没库存提示已售罄 有库存但队列已满提示业务繁忙请稍后再试）

加入购物车不减库存，只做记录

已加入队列但在规定时间内未支付订单，在超时后移出队列，恢复库存，记入数据库（未支付状态）