Cache

2021年11月17日　　17:29

==========

https://zhuanlan.zhihu.com/p/351551317

高性能缓存设计

设计和开发高性能系统，基本离不开缓存的设计，无论是CPI的L1,L2,L3级缓存，数据库的sql语句执行缓存，系统应用的本地缓存，乃至于现在用得做多的memcache，redis集中式缓存等。

缓存是解决性能的一把利器，本文主要从6个方面总结缓存及缓存的使用：

为什么需要缓存，缓存带来了什么
缓存的类型及常用缓存选型
缓存的基本算法思想
缓存带来的问题及解决
缓存组件的设计

为什么需要缓存，缓存带来了什么

实际工作中，在2种情况下，一定要引入缓存：

系统涉及大量频繁的读操作，例如，评论留言，商品页面 这种主要需要大量读的功能模块。
需要引入一个分布式集中存储，比如，保存客户端会话信息，权限统一控制等这种功能模块。遇到的实际情况是：系统没有数据库交互的模块，数据是本地存储的，但是会话信息需要在集群统一管理，所以使用redis作为统一的集中式缓存。

引入缓存，可以提高系统的性能，甚至可以很方便地实现一些功能，例如，会话统一管理。

引入缓存对于系统带来了什么呢？

缓存可以帮助系统提高用户体验，缓存解决的最大问题就是将读压力从数据库分离，同时，无论是嵌入式缓存还是redis这种集中式kv缓存，读的性能都比关系型数据库高很多。
解决数据库大量读操作带来的性能问题，数据库在修改和删除时才会对数据加锁，但是如果这种数据是大量需要读的数据，可能很多读操作会被修改操作阻塞，影响性能。
通过redis解决一些功能实现的问题，例如，一些分布式系统并没有数据库，例如mqtt这种broker，涉及到集群数据共享时，可以用redis解决，还有系统的分布式锁，得分

排序等功能都可以用redis解决。

引入缓存也带来了一定的复杂性：

数据一致性问题，引入缓存后，数据需要保存2份，一份是数据库，一份是缓存，那么是先更新缓存，还是先更新数据库？先更新缓存，可能导致读取最新数据，但是更新数据库失败了，需要回滚，那么缓存中数据就是脏数据，如果先更新数据库，那么读取的缓存数据是旧数据，也就是脏数据了。。。。肯定先更新数据库吧，然后刷新/删除缓存，刷新缓存基本不会失败的。

可维护性，引入缓存后，除了需要维护自己的系统外，还需要维护缓存

系统可用性，如果系统严重依赖缓存，例如将缓存当做中央存储，那么缓存挂掉，整个系统也就不可用了。导致系统可用性降低。。。。。集群。。

缓存带来的其他问题，例如本地缓存需要考虑内存大小，自己设计过期策略等等，集中式缓存需要考虑，缓存雪崩，缓存穿透，数据热点分布等一系列问题

缓存类型及常用缓存选型

嵌入式(本地)缓存

直接与应用一起启动，在工程中引入jar包即可使用

mapdb：使用较多的嵌入式kv数据库，也可以当做缓存使用，支持hash map set结构，在3.0已经不支持list和queue了，也支持数据落盘等。

h2：嵌入式关系数据库，可以当做缓存

encache：应用最多的本地缓存，可以实现分布式缓存，不过使用比较重

hazelcast：分布式应用缓存，使用不好会带来严重性能问题

自己实现本地缓存：适合数据量小，同时不需要数据分布式缓存的情况，实现简单。

嵌入式缓存一般用作二级缓存或目前系统还不需要引入外部缓存的情况，在业务规模较小时，本地缓存一般都可以解决问题了

集中式缓存

例如memcache，redis。

选项时的观点：

redis支持更多的数据库结构

redis支持数据备份，一定程度保证数据的高可靠

redis支持数据持久化

redis支持主从复制，容易实现故障恢复

redis的redis-cluster机制，可以帮助使用者不用在客户端做数据分片了。

缓存的基本算法思想

缓存一般基于内存，内存是珍贵的资源。各种缓存，内部的数据结构实现都比较复杂且多种多样，这里介绍一些常用的缓存过期算法：

FIFO，例如 Java线程池中的阻塞队列，线程池之所以能缓存执行任务，就是将执行任务放入了一个阻塞队列，每次从该队列中获取头结点的任务进行处理，队列是有限的，如果任务超过队列大小，有以下几种处理方法：创建新的线程/立刻执行该任务/抛弃该任务并抛出异常

LFU，Least Frequently Used，不常用，基本思想：如果数据过去被访问越多次，那么将来被访问的频率越高。关系数据访问频次，在实现时，需要维护一个队列专门记录所有数据的访问记录，每个数据需要维护引用计数，实现先对比较复杂，由于要维护一个计数队列，所以内存消耗较高，需要基于引用计数进行排序，性能消耗较高。

LRU，Least Recently Used，基本思想：如果数据最近被访问过，那么将来被访问的几率更高，CPU的L1，L2缓存都是该缓存算法，redis中缓存过期也是该算法思想。大

多数缓存也采用该思想，相对于LFU，实现难度较低，缓存命中率低于LFU算法。基本实现思想：新数据插入到链表头部，缓存命中时将数据移动到头部，链表满时移除链表尾部的数据

Redis中缓存淘汰策略基于LRU主要有2种思想：

主动淘汰：Redis会周期性地从设置了 过期时间的缓存数据里获取一部分数据，判断这些数据是否过期，过期则淘汰，如果过期的数据在本次周期中超过一定比例，则立刻在执行一次该方法。当缓存已经内存超过maxmemory限定时，会触发主动清除策略。

被动淘汰：当该key被访问时，判断是否已经失效，如果失效就删除它

缓存带来的问题及解决

本地缓存带来的数据不一致问题

数据不一致，如果使用encache，hazelcast等支持分布式缓存同步的还好，但是对于自定义缓存，mapdb等这种 本地缓存 无解。 这种要分析需求，看数据的变更频不频繁，实时性要求。

缓存数据的一致性问题

缓存是为了解决读的性能问题，但是数据更新时，是先更新缓存还是先更新数据库

先更新缓存，然后更新数据库：数据库会回滚，所以这种方法一般不可取

先更新数据库，在更新缓存：在更新完数据库到更新缓存这一段时间内，读到的是旧数据，但是问题较小，推荐使用

先删除缓存，然后更新数据库，最后同步缓存：数据库写成功后更新缓存，这样读缓存时读不到数据会从数据库读，数据是最新的，带来的问题是数据库压力在这一过程中较大。。。。。这个cache是需要手动设置的，不是想象中的，从数据库读到后就自动设置到cache中。。。

缓存雪崩

是指缓存失效(过期)后导致所有读操作都在数据库执行，导致数据库压力瞬间增大，性能下降。主要解决方法：

数据的过期时间不要设置成一样的，防止数据批量失效

采用互斥锁，这里需要使用到分布式锁，在缓存失效后，如果访问同一数据的操作需要访问数据库并更新缓存时，对这些操作加锁，保证只有一个线程访问数据库并更新缓存。

后台更新，业务操作需要访问缓存没有获取到数据时，不访问数据库更新缓存，只返回默认值，通过后台线程去更新缓存，有2种方式：

启动定时任务定时扫描所有缓存，如果不存在就更新，该方法导致扫描key间隔时间过长，数据更新不实时，期间业务操作会一直返回默认值，用户体验较差

业务线程发现缓存失效后通过MQ去更新缓存，这里因为是分布式的所以可能有很多条消息，需要考虑消息的幂等性。这种方式依赖消息队列，但是缓存更新及时，用户体验较好，缺点是系统复杂度增高了。

缓存穿透

业务操作访问缓存时，没有访问到数据，又去访问数据库，但是数据库也没有查到数据，也不写入缓存，从而导致这些操作每次都需要访问数据库，造成缓存穿透。

解决方法一般有2种：

将每次从数据库获取的数据，即使是空值 也先写入缓存，但是过期时间设置得比较短。例如3秒，后续的访问会从缓存中获取空值然后返回

对所有有结果的查询参数进行hash算法，利用Bloom filter将有查询结果的存储下

来，一个一定不存在的数据首先会在这个Bloom filter中过滤掉，从而防止后续访问底层存储。

缓存热点
对于缓存的数据，可能只有20%是经常需要被访问的。 对于缓存的数据，如果访问某个key的读操作很多，会导致这台缓存服务器压力十分大。
解决方法：为热点数据缓存多个副本，缓存的数据都是一样的，缓存的key按编号区分，所有的读操作随机读取其中的一份缓存，这样就可以放置所有的读操作都集中到一台缓存服务器上，这样的缓存(相同数据的多个缓存)也需要设置不同的缓存时间，防止缓存同时失效，引发缓存雪崩。

缓存组价的设计
设计缓存，一般从以下角度设计和考虑
什么样的数据应该缓存
什么时候应该触发缓存，怎样触发，什么时候取更新缓存，怎样更新
缓存的层次和粒度
缓存的命名规则和淘汰规则
缓存的健康以及故障对应方案
缓存数据的可视化及缓存的key内存设计和大小等。

==========

缓存穿透
cache和db都不存在。

缓存空查询，会占用额外空间
验证过滤，例如布隆过滤器
黑名单
业务逻辑前置校验

缓存击穿
cache没有，db有，大量请求。

设置热点key不过期，后台异步更新，适用于 不严格要求 缓存一致性的场景
加分布式锁
限流
使用互斥锁Mutex key，只让一个线程构建cache，其他线程等待cache构建完毕，重新从cache获得数据。单机通过synchronized 或lock来处理。分布式 使用分布式锁。

提前使用互斥锁：在value中设置一个 比 cache 过期时间 更短的 过期时间，当异步线程发现 该值快要过期时，重新从数据库中加载数据，设置到cache中，并且延长这个 内置的时间。

缓存雪崩
cache集中失效，导致全部走数据库

随机过期时间
考虑队列或锁，保证缓存单线程写，会影响并发量。
热点数据不过期，后台异步更新cache，适用于 不严格 要求缓存一致性的场景
双key策略，主key设置过期时间，备key不设置过期时间，主key失效，返回备key的值
构建缓存高可用集群
雪崩时，服务熔断，限流，降级

缓存一致性问题
解决db和cache 不一致问题的 主流做法是 延迟双删

延迟双删流程：

1. 更新操作先 删除 redis缓存
2. 更新db中的数据
3. 更新操作休眠一段时间
4. 更新操作再次删除redis缓存

```
def update_data(key, obj):
    del_cache(key)      # 删除 redis 缓存数据。
    update_db(obj)      # 更新数据库数据。
    logic_sleep(_time) # 当前逻辑延时执行。
    del_cache(key)      # 删除 redis 缓存数据。
```

。。？但是第一次删除有什么意义？ 在sleep的时候 依然可能有request，然后穿透到db吗，然后。。ok，此时db返回新数据。
sleep时间需要大于一次写操作的时间，一般3-5秒。

。还是感觉第一次删除有点多余， 你要求严格，就加锁。 要求不严格，更新到数据库后删除cache 就可以了啊。

延迟是因为mysql，redis 的 主从节点数据不是实时的，需要一段时间，来增强它们的数据一致性。
延迟是指当前请求逻辑处理延时，而不是当前线程睡眠延时。

服务节点删除 redis 主库数据。
服务节点修改 mysql 主库数据。
服务节点使得当前业务处理 等待一段时间，等 redis 和 mysql 主从节点数据同步成功。
服务节点从 redis 主库删除数据。
当前或其它服务节点读取 redis 从库数据，发现 redis 从库没有数据，从 mysql 从库读取数据，并写入 redis 主库。

延时双删，有等待环节，如果系统要求低延时，这种场景就不合适了。
延时双删，不适合“秒杀”这种频繁修改数据和要求数据强一致的场景。
延时双删，延时时间是一个预估值，不能确保 mysql 和 redis 数据在这个时间段内都实时同步或持久化成功了。

==========

https://juejin.cn/post/7126188464713760776

。。这篇质量不太好。。。1.4种策略，只说了一种。。 2. 有网络恢复这种。。

如何保证数据库与缓存数据一致性？

通常的缓存策略有以下几种：
1. Cache-Aside Pattern（旁路缓存，业务系统常用）
2. Read-Through Pattern
3. Write-Through Pattern
4. Write-Behind Pattern

Cache-Aside Pattern 旁路缓存模式
读取缓存，读取数据库，更新缓存， 都是在 应用系统中完成的，是业务系统最常用的缓存策略。
旁路缓存策略又分为读缓存 和 写缓存

读缓存
读取数据 - 判断缓存中是否有 - 有数据直接返回
|
没有数据 - 从数据库加载数据 - 数据写入缓存 - 返回数据

写缓存
写数据 - 更新数据库中数据 - 删除旧缓存

Cache-Aside Pattern 一致性问题场景分析
对于数据不一致的场景，提供4种更新方案：

1. 先更新缓存，然后更新数据库
2. 先更新数据库，再更新缓存
3. 先删除缓存，再更新数据库
4. 先更新数据库，再删除缓存

存在的问题：
先更新缓存，再更新数据库
更新缓存成功，但是写数据库失败，这会导致 缓存中是最新数据，但是数据库中是 旧数据。
不建议使用。

先更新数据库，再更新缓存
更新数据库成功，但是由于网络问题，导致更新缓存失败。导致缓存中是旧数据。
不建议使用。

先删除缓存，再更新数据库
删除缓存成功，网络问题导致更新数据库失败。
另一线程，由于缓存为空，查询数据库数据，并写入缓存。 从而导致数据不一致。
不建议使用

先更新数据库，再删除缓存
写入数据库成功，网络问题导致 删除缓存失败
另一线程，读取缓存中数据， 但是当网络恢复后，缓存中的数据会被删除，所以可能存在短暂的数据不一致。

虽然存在短暂的数据不一致，但是在 旁路缓存策略中，对于写操作：先更新数据库，再删除缓存。
。。。？网络恢复？ 其他方法，网络恢复后也没有问题的啊。

数据库和缓存一致性解决方案

缓存延迟双删， 流程如下：
先删除缓存，然后更新数据库，确保数据库事务提交成功，然后休眠一段时间，再删除缓存。

删除缓存失败如何处理
删除缓存重试机制
删除失败的key 存入消息队列中，采用异步的方式来删除，如果删除失败的次数已经超过最大次数，发送警告邮件，需要人工介入。

==========
https://juejin.cn/post/6844904037943345165

如何解决高并发场景下缓存+数据库双写不一致问题?

Redis主要解决了 关系型数据库 并发了低 的问题，有助于 缓解 关系型数据库 在高并发情况下 的压力，提高系统的吞吐量

各种等级的 不一致问题 及解决方案
最初级的 缓存不一致 问题及方案

先修改数据库，再删除cache。 如果删除cache失败，那么会导致数据库中是 新数据，cache是旧数据，出现不一致的情况

方案：

反过来，先删除缓存，在修改数据库。 。。。所以要双删？这里可能是删除缓存和 修改数据库 之间 其他线程 读取数据库并修改cache

比较复杂的数据不一致问题分析

先删缓存，然后修改数据库。

如果修改数据库的操作 还没有完成， 又来了一个请求，去读缓存，cache空，所以查询数据库，查到 旧数据，放到cache中。

数据变更操作完成后 数据库的库存是 新值，但cache中 是旧数据，会出现缓存和数据库不一致的情况

高并发后出现 db cache 不一致的情况 会非常高。

更新和读取操作进行异步串行化
异步串行化

在系统内部维护 n个内存队列，更新数据时，根据数据的 唯一标识，将该操作路由之后，发送到其中一个 jvm 内部的 内存队列中 (对同一个数据的 请求发送到 同一个队列)。读取数据时，如果发现数据不在cache中，并且此时队列中有 更新操作，那么将 重新读取数据+更新cache 的操作，根据唯一标识路由之后，也发送到 同一个 jvm 内部 的内存队列中。 然后每个队列 对应一个 工作线程，每个工作线程 串行地拿到对应的操作，然后一条条 执行。

这样的话，一个数据变更的操作，先执行删除缓存，然后再去更新数据库，但是还没完成更新的时候，如果此时一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，排在刚才更新库的操作之后，然后同步等待缓存更新完成，再读库。

读操作去重
多个读操作在 队列中 是没有意义的，因此可以做出过滤。如果 发现队列中 已经有了 该数据的 更新缓存的请求，那么就不用放进去了，直接等待前面的 更新操作完成 即可，等那个队列 对应的工作线程 完成了 上一个操作 (数据库修改) 之后，才会去执行 下一个操作 (读库 更新缓存)，此时会从数据库中读取最新值，然后写入cache。
。。？ 感觉是 不发 select 到 db和cache，而是直接使用 前一个 update 后的值 作为select的结果， 返回。 但是还是得插入一个 操作的(直接返回上一次的结果)。 不然 等待 队列为空的话， 会饥饿吧(无限的update，队列永不空，select 永远不会返回)。。下面就是 防止饥饿的。但是还是 感觉 我这种更好啊。

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过了一定时长，那么这次 直接 从数据库读取 当前的旧值。
。。还有，我记得 写锁 和 读/写 锁 互斥的。 如果有人 更新的话， 读不了。当然，不

一定加锁。或者 锁很快。
。。队列的话，就意味着 没有事务了啊。

高并发场景下，该方案需要注意的问题。
读请求长时间阻塞
由于读请求进行了 轻度 的异步化，所以一定要注意 超时的问题。
这个解决方案最大的风险是，数据更新很频繁，导致 队列中 积压了大量 更新操作，然后读请求 会大量超时，最后导致 大量的请求直接走数据库 取旧值。 所以要 模拟真实情况，看下数据频繁修改的情况下 会怎么样。
另外，因为一个队列中，可能会积压针对 多个数据项的 更新操作，因此需要根据自己的业务情况 进行测试，确认一个实例中 创建多少个 内存队列，且 可能需要部署多个 服务，每个服务分摊一些数据的 更新操作。

一定要根据 实际业务系统的 运行情况，进行 压力测试。去看 最繁忙的时候，内存队列可能会积压 多少更新操作，可能会导致 读请求 挂起多久。

如果一个 内存队列中 积压的 更新操作特别多，那么就需要 加机器，让每个机器上 部署的 服务实例 处理更少的数据，那么 每个内存队列中 积压的 更新操作 就会越少。

根据项目经验，一般来说 数据的 写频率 是很低的。
大部分情况下：应该是：大量读请求过来，都是直接走 缓存取数据的，少量情况下，可能遇到 读和数据更新 冲突的 情况。如上所述，那么此时 更新操作 如果先入队列，之后可能会 瞬间来了 对这个数据的大量请求，但是因为做了 去重的优化，所以 也就一个 更新缓存(?。不是读？但是读，好像确实不进队列。)的操作 跟在它后面。

读请求并发量过高
必须做压力测试。
还有一个风险，就是 瞬间大量读请求 会在 几十ms 的延时 内 在服务上 挂起，这要看 服务能不能抗住。需要多少机器 才能抗住。

但是因为 并不是所有的数据 都在同一时间更新，缓存也不是同一时间失效，所以每次 可能也就是少量 数据的 缓存失效，然后那些 数据的 读请求过来，并发量 应该也不是特别大。

多服务实例部署的请求路由
可能一个服务有多个实例，那么必须保证，执行数据更新操作，以及 执行 缓存更新操作的请求，对于同一个商品的读写请求 全部路由到 一个 实例上。 可以自己做 服务间 按照某个请求的 参数的hash 路由，也可以通过 nginx 路由功能 路由到 相同的服务实例上。

热点商品的路由问题，导致请求的倾斜
万一某个商品的 读写请求特别高，全部 到了 某台机器的 某个队列中，可能造成这台机器压力过大。
但是因为只有在 商品数据 更新时 才会情况cache，才会导致 读写并发，所以 更新频率不是太高的话，这个问题 的影响不是 特别大。

总结
一般来说，如果你的系统不是 严格要求 db+cache 必须一致性的话，cache 可以稍微 和数据库 偶尔有不一致的情况，那么 最好 不要 使用上述的 串行化 方案，因为 读请求 和写请求串行化 到 一个内存队列中，这样 保证 一定不会 出现 不一致的情况，但是，串行化之后，会导致 系统的 吞吐量 大幅度 下降。

评论：

通常来说，能用到缓存那说明对数据的实时性和一致性要求肯定没那么高，我觉得可以采用加锁的方式：
1.缓存不设置有效期，但是通过某个字段记录过期时间
2.当某个请求过来的时候发现缓存过期了，那么开始查库更新缓存，这块加一个redis分布式锁，保证只能有一个请求去更新缓存
3.后续请求过来发现缓存过期了，但是因为有锁，这时候依然使用过期的缓存
这样做，还可以顺便解决缓存穿透问题。

==========

https://juejin.cn/post/7151937376578142216

场景：论坛系统； 配置，业务数据，都保存在DB中，随着业务的发展，数据越来越多，整个系统的响应越来越慢。

本地cache
接口维度的 短期cache，对每个接口的 请求参数(帖子ID) 和 响应内容 缓存一定的时间(如1分钟)。
Java 有很多框架提供了这种能力，如：EnCache，Gauva Cache，Coffeine Cache。 可以通过 注解 来实现缓存功能。
比如EhCache，代码如下

```
@Cacheable(value="UserDetailCache", key="#userId")
public UserDetail queryUserDetailById(String userId) {
    UserEntity userEntity = userMapper.queryByUserId(userId);
    return convertEntityToUserDetail(userEntity);
}
```

本地cache 会带来 缓存漂移 问题
因为业务集群 存在多个节点，而缓存是 每个节点本地构建的，所以会出现 节点的缓存不一致，所以可能导致 多次刷新，有些帖子会出现-消失-出现。

集中式缓存
Redis，Memcached。

解决了 缓存不一致问题，单机内存容量限制。

多级缓存的联合
集中式缓存存在滥用的可能，导致 性能没有太大的提升。
虽然Redis 单次请求处理 性能极高，甚至可以达到微妙级，但网络IO，线程的阻塞和唤醒，使得系统在 线程上下文切换层面 浪费巨大。

要降低 网络IO，就回到了 本地cache。

对于一些 变更频率较高的 数据，采用 集中式缓存，这样可以确保数据变更后 所有节点可见，确保数据一致
对于 极少变更的(如 系统配置项)，或者一些 对短期一致性要求不高的 数据 (如 用户昵称，签名 等)，则采用 本地cache，来减少 对 远端集中式cache 的 网络IO 次数。

降低自身CPU消耗
cache 最典型的 使用场景就是 性能优化。在 性能优化层面，一个最经典的 策略就是 空间换时间。如：

1. 在数据库 表中做一些 冗余字段
   比如 用户表 保存了 外键，指向 部门表。除了这个外键外， 还额外保存了 部门 的名称，这样 在 需要展现 用户所属部门的 时候，就不需要去 查询 部门表了。
2. 对一些中间结果进行存储
   数据报表，要对 系统内 的 业务数据进行 统计，需要多张表 作为数据来源，进行汇总计算后 得出最终结果。 如果借助缓存，则可以将一些 中间计算结果 进行cache，然后 报表请求 基于 cache 进行 二次处理，这样可以 降低 基于请求 触发的 实时计算量。

空间换时间 策略中， 缓存是 核心。 借助缓存，可以将 一些 CPU密集 的计算结果 进行保存 以便复用，来降低 CPU 消耗。

减少对外IO交互
上面的缓存 是为了 降低 处理请求时 对 自身CPU 的占用， 从而提升 处理能力。

这里介绍另一种 场景， 减少 系统 对外依赖 的请求频次。 即 将一些 从 远端 返回的结果进行缓存，后面的 直接使用 cache 中的数据，而无需再次发起 请求。

对于服务端而言，通过构建cache 的方式来减少 自身对外的 IO 请求，主要有几个 考量出发点：

1. 从自身性能考虑，减少对外IO，降低 对外接口的 响应时延，对 服务端自身的性能有一定提升
2. 对 远端服务器 稳定性考虑， 避免 远端服务压力过大。 很多时候，调用方和 被调用方 系统的承压能力 是不匹配的。 为了避免 远端服务被压垮，自身需要缓存结果来降低 发出的请求。
3. 从 自身可靠性 层面而言，将一些远端服务器 请求到的 结果 缓存起来， 即使 远端服务器出现故障，自身业务依旧 可以 基于 cache数据 进行正常的 业务处理。 来提升自身的 抗风险能力。

对于移动端APP 和 H5 界面而言，cache 还可以降低 用户的流量消耗。也会 提升 用户体验(界面渲染快，较少等待时间)

提升用户个性化体验

cache 除了 系统性能，可靠性 提升外，在APP 或者 WEB 类用户侧 产品中，还经常被用于存储一些 临时的 个性化 使用习惯配置 或 身份数据，以提示 用户的 个性化使用体验。

1. 缓存cookie，session 等身份鉴权 信息，避免用户每次访问都需要 进行 身份验证。
2. 记住 用户上次 操作习惯，比如 用户在 页面上 将 列表分页查询 设置为 100条/页，则后续 在系统内访问其他列表页面时，也使用这一设置。
3. 缓存用户的一些 本次设置，主要是在APP 端使用，可以 保存一些 与当前设备绑定的设置信息，仅对当前设备有用。 比如 同一个账户，登录手机时 希望是 深色模式，Pad时 希望 浅色模式。

业务与缓存的集成模式
我们可以在不同的方面 使用 cache 来辅助达成项目 在某些方面的 诉求。
而根据使用场景的不同， 在 结合 缓存与业务逻辑时， 会存在不同的 架构模式，典型的有 旁路型缓存，穿透型缓存，异步型缓存。

旁路型缓存
这种模式下，业务自行负责与 缓存 和数据库 的交互， 自由决定 cache 未命中 时的 处理策略，更契合 大部分业务场景的 定制化 诉求。

由于业务模块自行实现 cache 和db 之间的 数据写入 和 更新逻辑。在实际实现时 要注意高并发下 的 数据一致性问题， 以及 可能的 缓存击穿，缓存穿透，缓存雪崩 等。

旁路型cache 是 最常用的。

穿透型cache
实际业务中 使用较少，主要是 应用在 一些 缓存类的 中间件中，或者在 一些 大型系统中 专门的 数据管理模块中 使用。

一般情况下，业务使用cache 时， 是先尝试读取 cache，再尝试读取DB。 而使用 穿透型缓存架构时， 会有专门的 模块 将这些动作 封装成黑盒， 业务模块 不会直接 和 数据库进行 交互。

这对业务比较友好，业务只需要调用 缓存接口即可，不需要自己实现 cache 和 DB 之间的交互策略。

异步型cache
可以看做是 穿透型cache 的 演进异化版本， 使用场景也较少。
主要策略就是 业务侧请求的 实时读写交互 都基于 缓存进行，任何数据的 读写都是 完全基于 cache的。 有一个 单独的 数据化持久操作(独立线程或进程中 进行)， 用于将cache 中 变更的数据 写入到 db中。

这种情况下，业务数据 读写 完全基于 cache， db 只是一个 数据持久化 存储的 备份

盘。
由于 实时业务 只和 cache 进行交互，所以 性能上 更好。
有一个致命的问题：数据可靠性，因为是 异步执行，所以 在下一次数据 写入DB前，有一段时间，数据只存在于cache中，cache服务器宕机时，这部分数据会丢失。所以 适用于 对数据一致性 要求不是 特别高的 场景。

缓存可以 大幅度提高 并发 和 承压能力，但也可能让系统崩溃，所以需要知道 cache 设计和使用的 一些关键点，才能游刃有余

可删除重建

这是 cache 和 持久化存储 最大的一个区别。 cache 的定位 是为了 辅助业务处理，也就是 有则用，没有也不会影响 业务运转。 此外，即使 cache 数据出了问题，也可以将其删除重建。
在服务端 构建cache 时， 也应该具备这种特性。比如基于 内存的 cache，当 业务进程重启后， 应该有 途径可以将 cache 重建出来。

有兜底屏障

cache 是 高并发系统 中的核心组件，一旦cache 出问题，整个系统 通常会崩溃。
cache 需要有 充足 且 完备的 兜底 和 自恢复 机制：

1. 关注 缓存数据量 超出 承受范围的 处理策略，比如 定好 数据的 淘汰机制
2. 避免cache 集中失效，比如 批量加载数据到 cache 时 随机化 过期时间，避免 大量 cache 同时失效，导致 缓存雪崩
3. 有效地 冷数据预热 加载机制，和 热点数据防过期 机制，避免 出现 大量 对 冷数据的请求 无法命中 cache 或 热点数据 突然失效，导致 缓存击穿 问题。
4. 合理的 防身自保 手段， 比如 采用 布隆过滤器，避免 被恶意请求攻击，导致 缓存穿透 问题

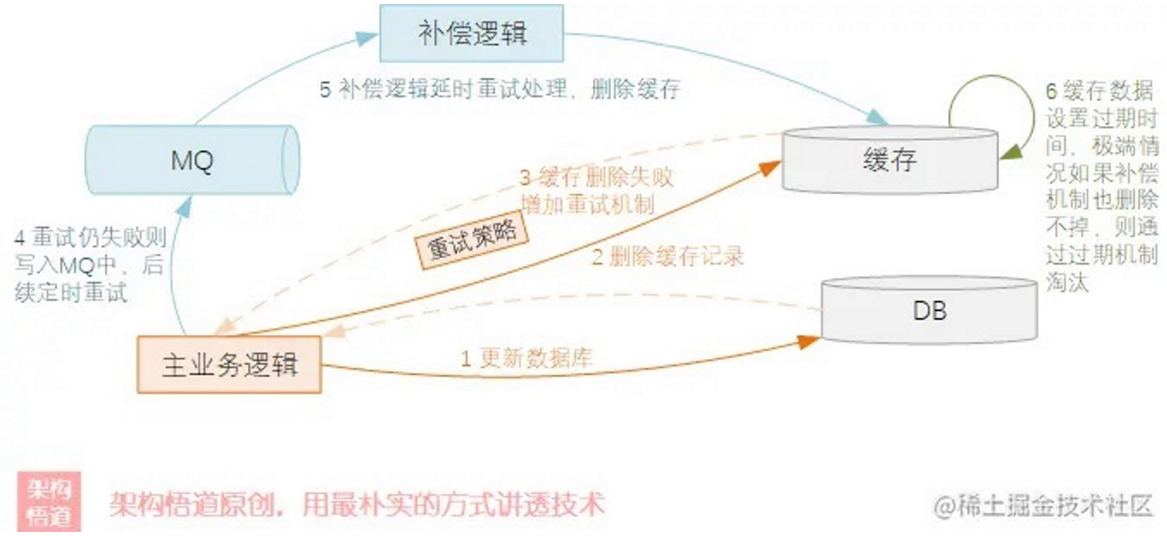
cache 一致性保证

高并发系统，进行数据更新时，cache 和 db 的一致性问题。
对于基于 旁路型 cache 的 大部分业务而言，数据更新操作，一般可以组合 出 几种 不同的 处理策略。

1. 先更新cache，再更新db
2. 先更新db，再更新cache
3. 先删除cache，再更新db
4. 先更新db，再删除cache

由于 大部分db 都支持 事务， 而几乎所有的 cache 都不具有事务性。 所以在一些 写操作并发 不是特别高， 且 一致性要求 不是特别强烈的 情况下，可以 简单地 借助 数据库的 事务进行控制。 比如 先更新数据库 再更新cache，如果 cache 更新失败则 回滚数据库事务。

在一些 并发请求特别高的时候，基于 事务控制 来保证 数据一致性 会对 性能造成影响，且 事务隔离级别越高， 影响越大。 所以可以采用一些 其他辅助策略，来替代 事务的控制，如重试机制，或 异步补偿机制，或 多者结合。


补偿逻辑
5 补偿逻辑延时重试处理，删除缓存
MQ
缓存
6 缓存数据设置过期时间，极端情况如果补偿机制也删除不掉，则通过过期机制淘汰
3 缓存删除失败增加重试机制
重试策略
4 重试仍失败则写入MQ中，后续定时重试
2 删除缓存记录
DB
主业务逻辑
1 更新数据库
架构悟道
架构悟道原创，用最朴实的方式讲透技术
@稀土掘金技术社区

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========