<https://mp.weixin.qq.com/s/ziJ8OFiLA1it3sQvycN7Mg>

翻越缓存的三座大山

### 前言

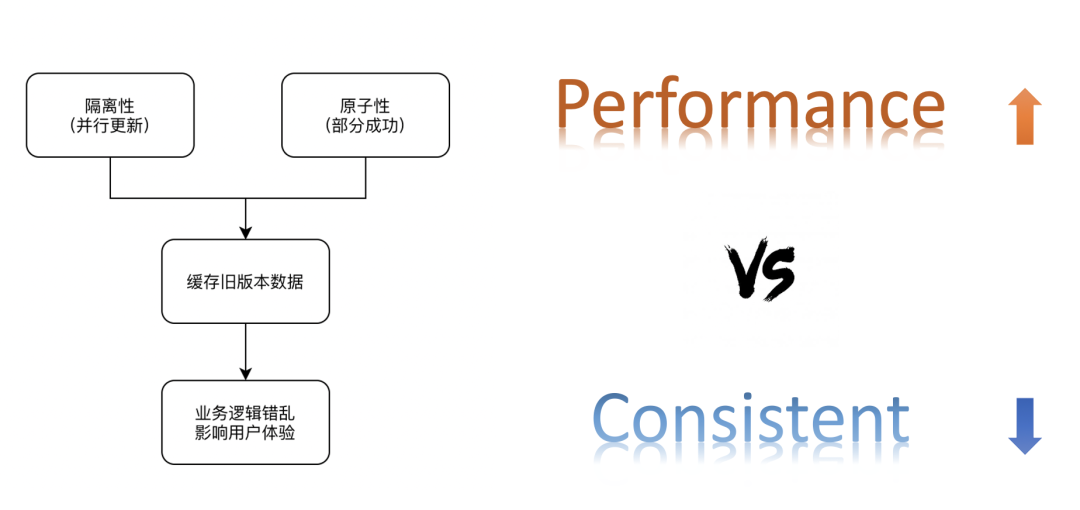
在互联网和移动互联网两波浪潮的推动下，存储技术有了飞速发展。移动互联网用户在过去十年增长了 10 倍，用户的增长带动了数据量的指数级增长，因为激烈的市场竞争，企业和用户对应用程序的响应性能要求越来越高，在完美应对庞大的用户规模和海量数据集的同时保证优秀的产品体验，是数据库面临的挑战。在机械硬盘普及的时代，企业需要通过缓存技术加速数据的访问，在 SSD 存储介质普及后，企业需要缓存技术支撑高并发和大吞吐，通过引入分布式缓存方案，提升应用程序性能，消除数据库热点。但是**缓存技术的引入增加了业务架构的复杂度，降低了开发效率，同时还面临着缓存一致性、缓存击穿、缓存雪崩等挑战**。

### 缓存的三座大山



#### 缓存一致性

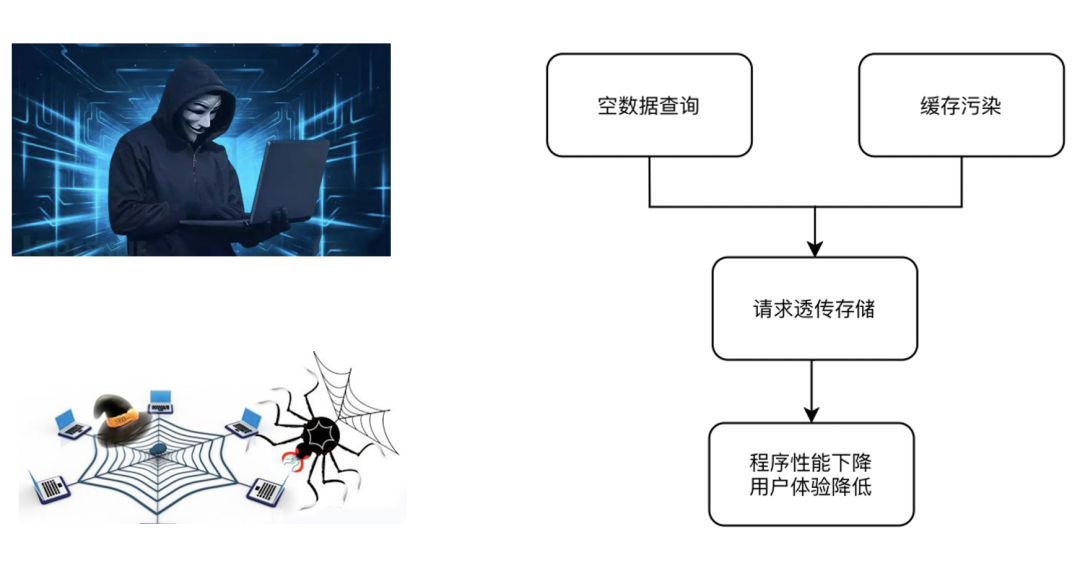
缓存一致性是指业务在引入分布式缓存系统后，业务对数据的更新除了要更新存储以外还需要同时更新缓存，对两个系统进行数据更新就要先解决分布式系统中的隔离性和原子性难题。目前大多数业务在引入分布式缓存后都是通过牺牲小概率的一致性来保障业务性能，因为要在业务层严格保障数据的一致性，代价非常高，业务引入分布式缓存主要是为了解决性能问题，所以在性能和一致性面前，通常选择牺牲小概率的一致性来保障业务性能。



#### 缓存击穿

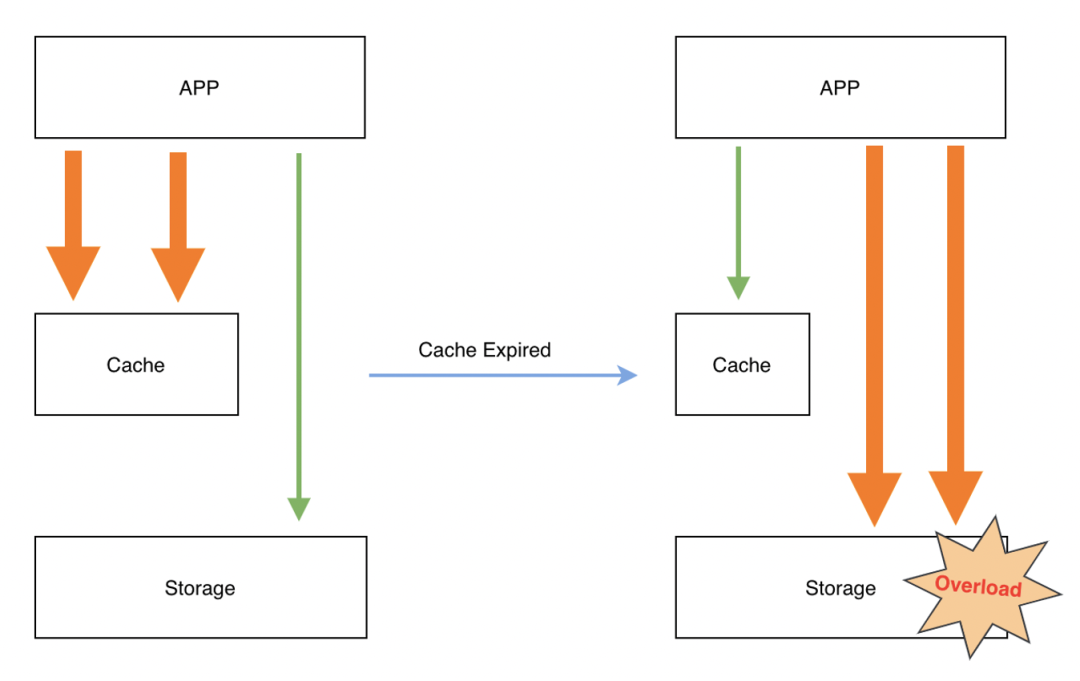
缓存击穿是指查询请求没有在缓存层命中而将查询透传到存储 DB 的问题，当大量的请求发生缓存击穿时，将给存储 DB 带来极大的访问压力，甚至导致 DB 过载拒绝服务。空数据查询(黑客攻击)和缓存污染（网络爬虫）是常见的引发缓存击穿的原因。什么是空数据查询？空数据查询通常指攻击者伪造大量不存在的数据进行访问（比如不存在的商品信息、用户信息）。缓存污染通常指在遍历数据等情况下冷数据把热数据驱逐出内存，导致缓存了大量冷数据而热数据被驱逐。缓存污染的场景我们目前还没有发现较好的解决方案，但是在空数据查询问题上我们可以改造业务，通过以下方式防止缓存击穿：

* 通过 bloomfilter 记录 key 是否存在，从而避免无效 Key 的查询；
* 在 Redis 缓存不存在的 Key，从而避免无效 Key 的查询；



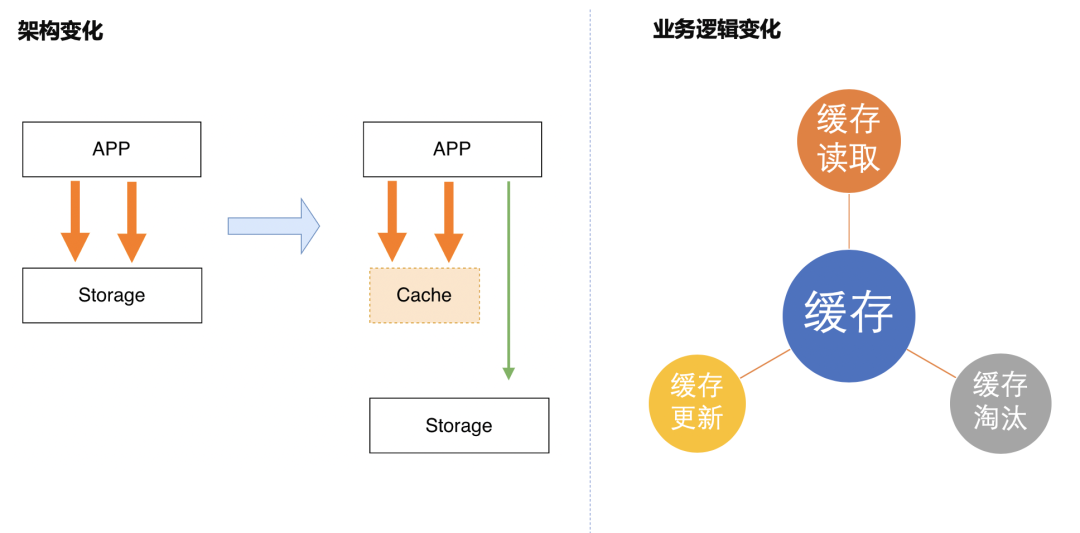
#### 缓存雪崩

缓存雪崩是指由于大量的热数据设置了相同或接近的过期时间，导致缓存在某一时刻密集失效，大量请求全部转发到 DB，或者是某个冷数据瞬间涌入大量访问，这些查询在缓存 MISS 后，并发的将请求透传到 DB，DB 瞬时压力过载从而拒绝服务。目前常见的预防缓存雪崩的解决方案，主要是通过对 key 的 TTL 时间加随机数，打散 key 的淘汰时间来尽量规避，但是不能彻底规避。



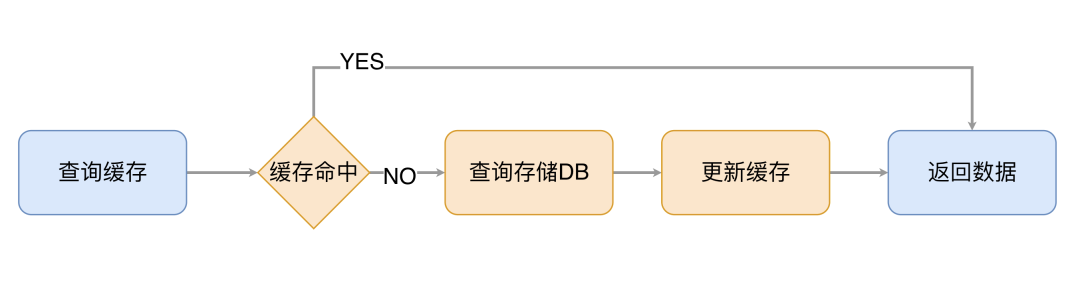
### 传统分布式缓存方案

在引入分布式缓存后，我们的业务架构由原有两层架构（应用+数据库）变成了三层架构（应用+缓存+存储），缓存层缓存热数据，存储层负责全量数据持久化存储。存储架构的变化要求业务对数据的存取逻辑进行相应调整，而且这个调整是巨大的。在缓存系统的选择上，常见的缓存数据库包括 Memcached、Redis，目前使用最广泛的是 Redis，存储数据常见的包括关系型数据库 MySQL、PG、Oracle、SQL Server 等，NoSQL 数据库 MongoDB、Hbase 等。**在引入分布式缓存后，业务逻辑需要做三个点的变化，缓存读取、缓存更新、缓存淘汰。**



#### 缓存读取

引入缓存层后，读数据就变得不是那么简单直接了，APP 需要先去缓存读取数据，如果缓存 MISS（数据没有被缓存），则需要从存储中读取数据，并将数据更新到缓存系统中，整个流程和代码如下所示：



#### 缓存更新

我们把常见的缓存更新方案总结为两大类，业务层更新和外部组件更新，比较常见的是通过业务更新的方案。

##### 业务层更新缓存

###### 缓存更新的难点

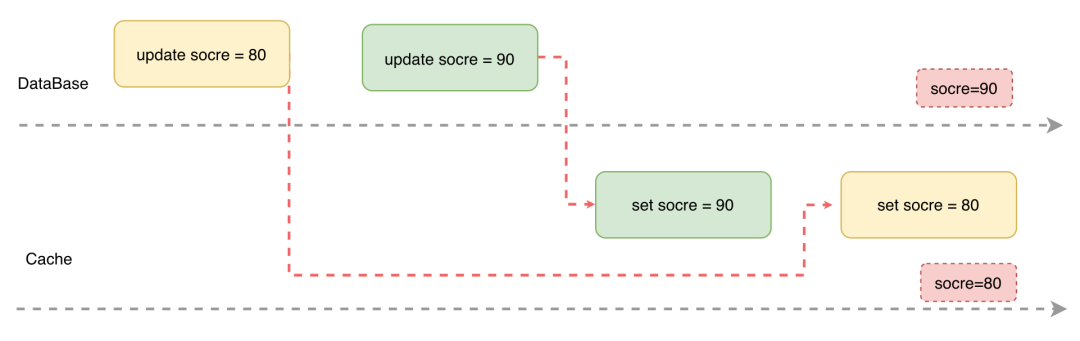
刚开始接触缓存方案的同学可能会纠结几个点，先更新缓存还是先更新存储，缓存的处理是通过删除来实现还是通过更新来实现。这里我们面临的问题本质上是一个数据库的分布式事务的问题，需要处理数据可靠性的挑战，并发更新带来的隔离性挑战，和数据更新原子性的挑战。

* 数据可靠性

如果要保证数据的可靠性，在业务逻辑成功之前，必须保障有一份数据落地，我们有以下两个选择：

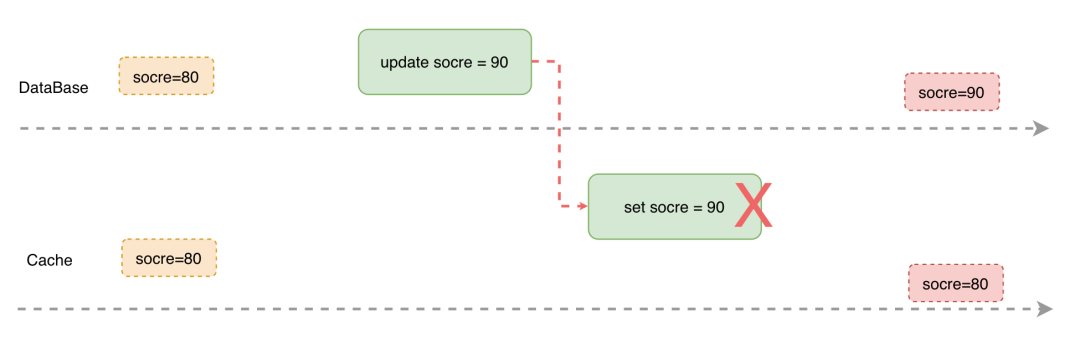
* 先更新成功存储，再更新缓存；
* 先更新成功缓存，再跟新存储，如果存储更新失败，删除缓存；
* 操作隔离性。

一条数据的更新涉及到存储和缓存两套系统，如果多个线程同时操作一条数据，并且没有方案保证多个操作之间的有序执行，就可能会发生更新顺序错乱导致数据不一致的问题。



* 更新原子性

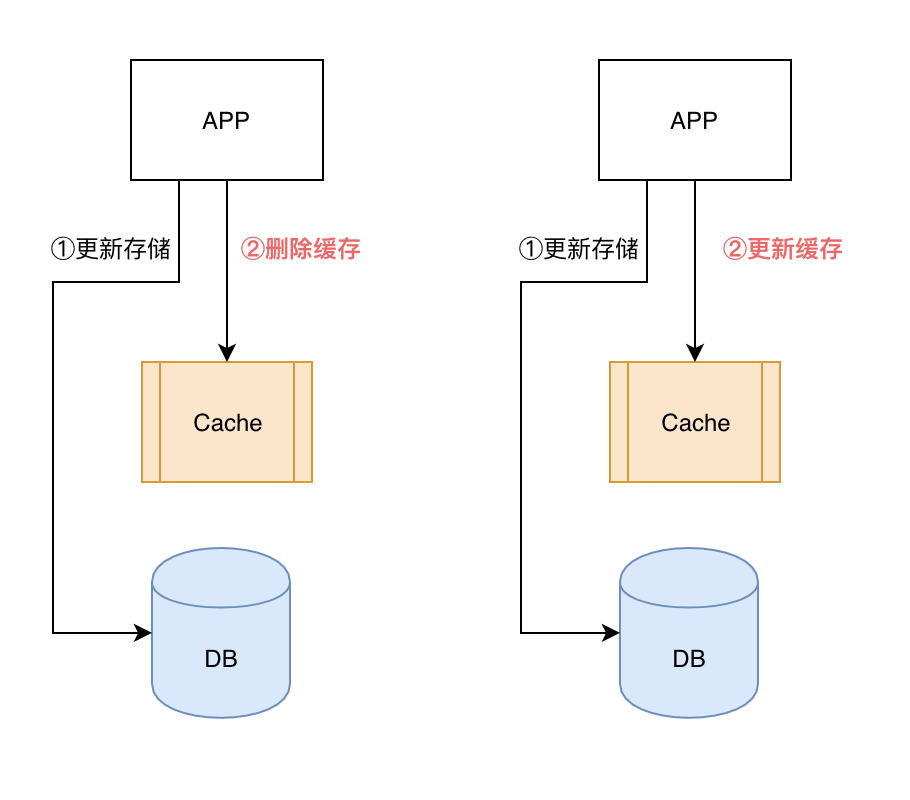
引入缓存后，我们需要保证缓存和存储要么同时更新成功，要么同时更新失败，否则部分更新成功就会导致缓存和存储数据不一致的问题。



业务层缓存更新方案

我们看到大多数的常见是选择以下方案，保障数据可靠性，尽量减少数据不一致的出现，通过 TTL 超时机制在一定时间段后自动解决数据不一致现象。

* Step1：更新存储，保证数据可靠性；
* Step2：更新缓存，2 个策略怎么选：
  + 惰性更新：删除缓存，等待下次读 MISS 再缓存（推荐方案）；
  + 积极更新：将最新的值更新到缓存（不推荐）；

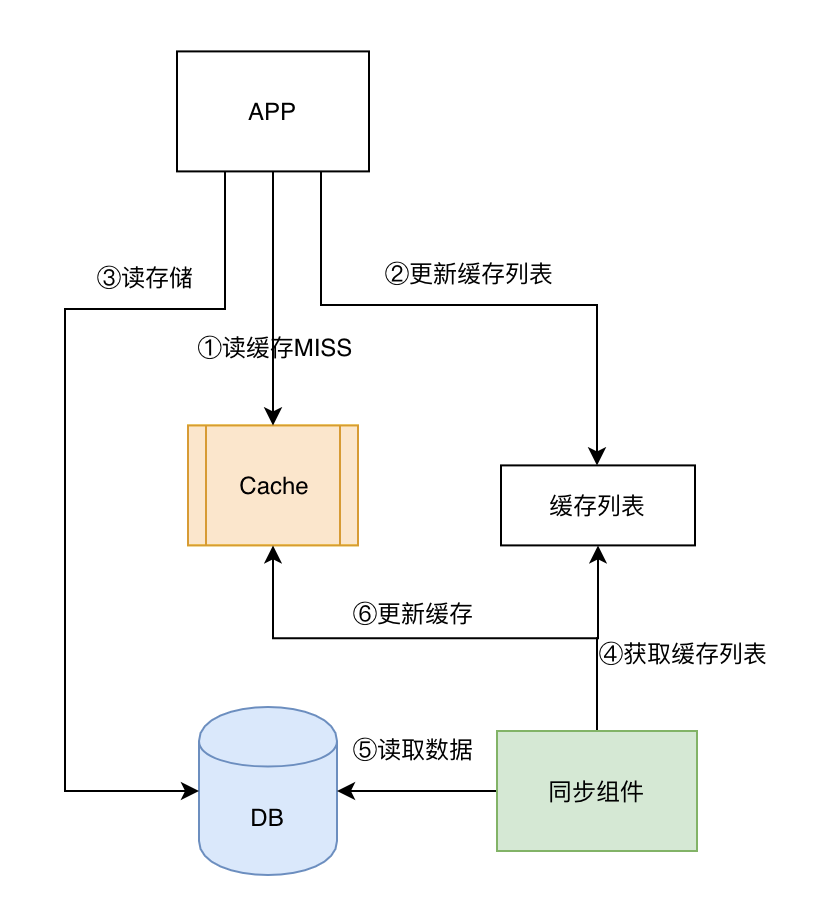


积极更新策略，缓存数据实时性更高，但是在缓存侧带来了更多的更新操作，这会提高更新冲突导致脏数据概率。

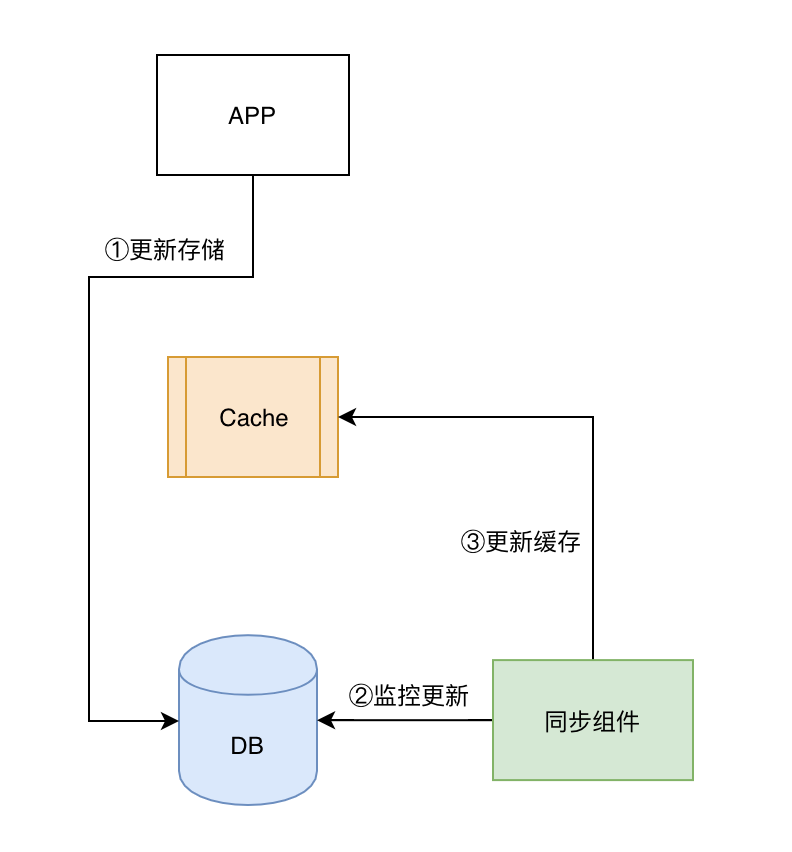
###### 外部组件更新缓存

* 缓存 MISS 处理方案

在通过第三方组件更新的方案中，为了保障数据的一致性，避免对单条数据的并行更新，缓存的所有更新操作都需要交给同步组件，因此缓存 MISS 场景下的逻辑：



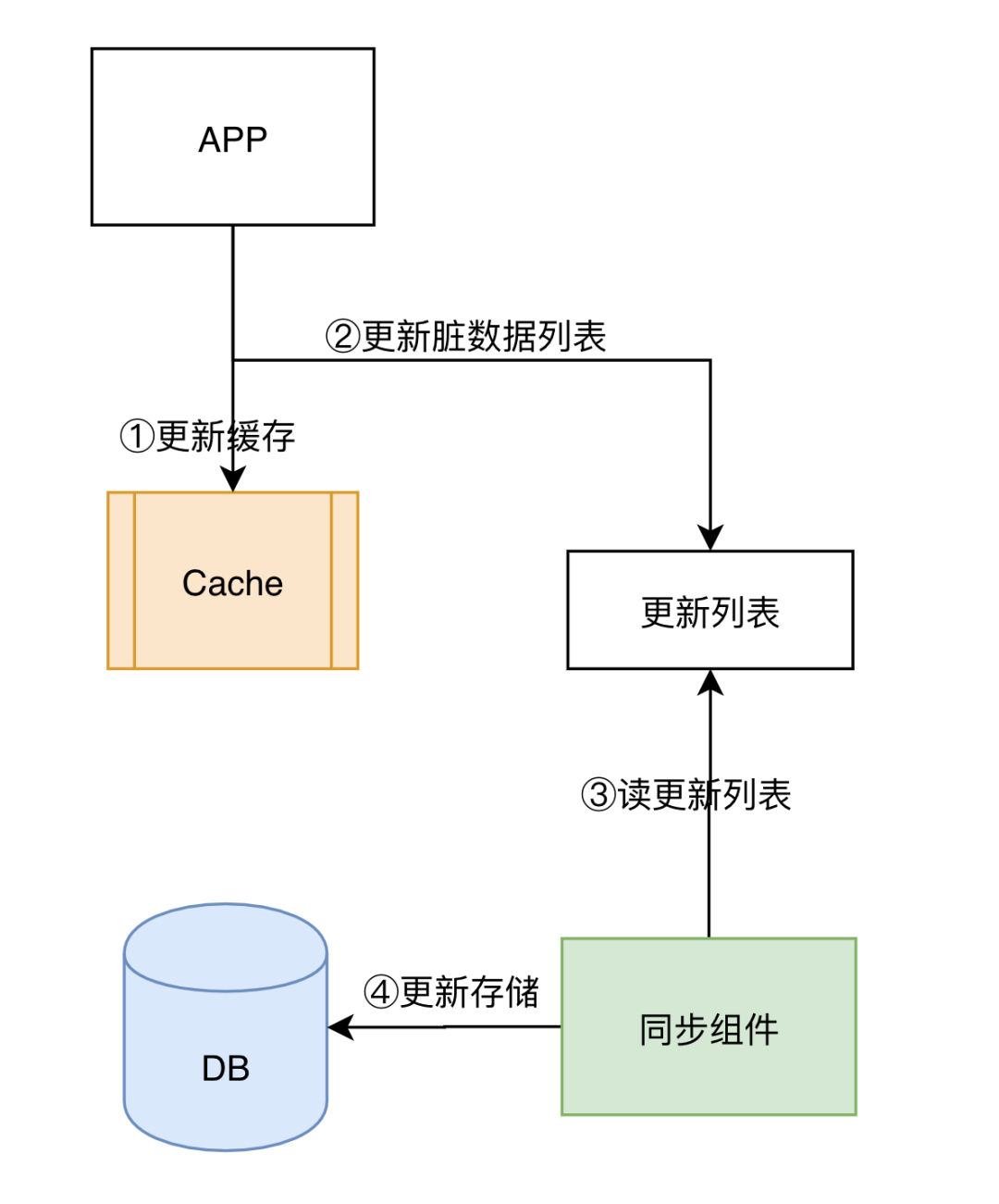
* 缓存更新方案
  + 第一：需要监控存储的日志，或者通过 Triger 来监控存储数据的变更，需要对存储系统非常熟悉；
  + 第二：需要对更新进行过滤，我们的目的是缓存热数据，但是像 DDL、批量更新这一系列的操作是不需要更新缓存的，要把非业务更新操作过滤；
  + 第三：同步组件需要理解数据，不通用；
  + 先更新存储，由第三方组件异步更新缓存；
  + 该方案投入较大，只适合特定的场景，并且有以下 3 个难点：



* 其他缓存更新方案

在实际的生产中，我们还会看到很多先更新缓存，然后通过第三方组件更新存储的场景，但是这个方案也会面临数据一致性和数据可靠性的挑战，虽然不推荐，但是确实还是能看到有在使用这个方案的，我们拿出来探讨下。

* 这个场景数据可靠性，不及先更新存储的方案，但是写入性能高，延迟低；
* 这个方案 APP 和第三方组件都会更新 Cache，会存在数据一致性的问题，因为很难保障两个组件更新的时序。



#### 缓存淘汰

缓存的作用是将热点数据缓存到内存实现加速，内存的成本要远高于磁盘，因此我们通常仅仅缓存热数据在内存，冷数据需要定期的从内存淘汰，数据的淘汰通常有两种方案：

* 主动淘汰，这是推荐的方式，我们通过对 Key 设置 TTL 的方式来让 Key 定期淘汰，以保障冷数据不会长久的占有内存。TTL 的策略可以保证冷数据一定被淘汰，但是没有办法保障热数据始终在内存，这个我们在后面会展开；
* 被动淘汰，这个是保底方案，并不推荐，Redis 提供了一系列的 Maxmemory 策略来对数据进行驱逐，触发的前提是内存要到达 maxmemory（内存使用率 100%），在 maxmemory 的场景下缓存的质量是不可控的，因为每次缓存一个 Key 都可能需要去淘汰一个 Key。