

# 缓存设计

---

## 1.最早的缓存，计算机内部缓存

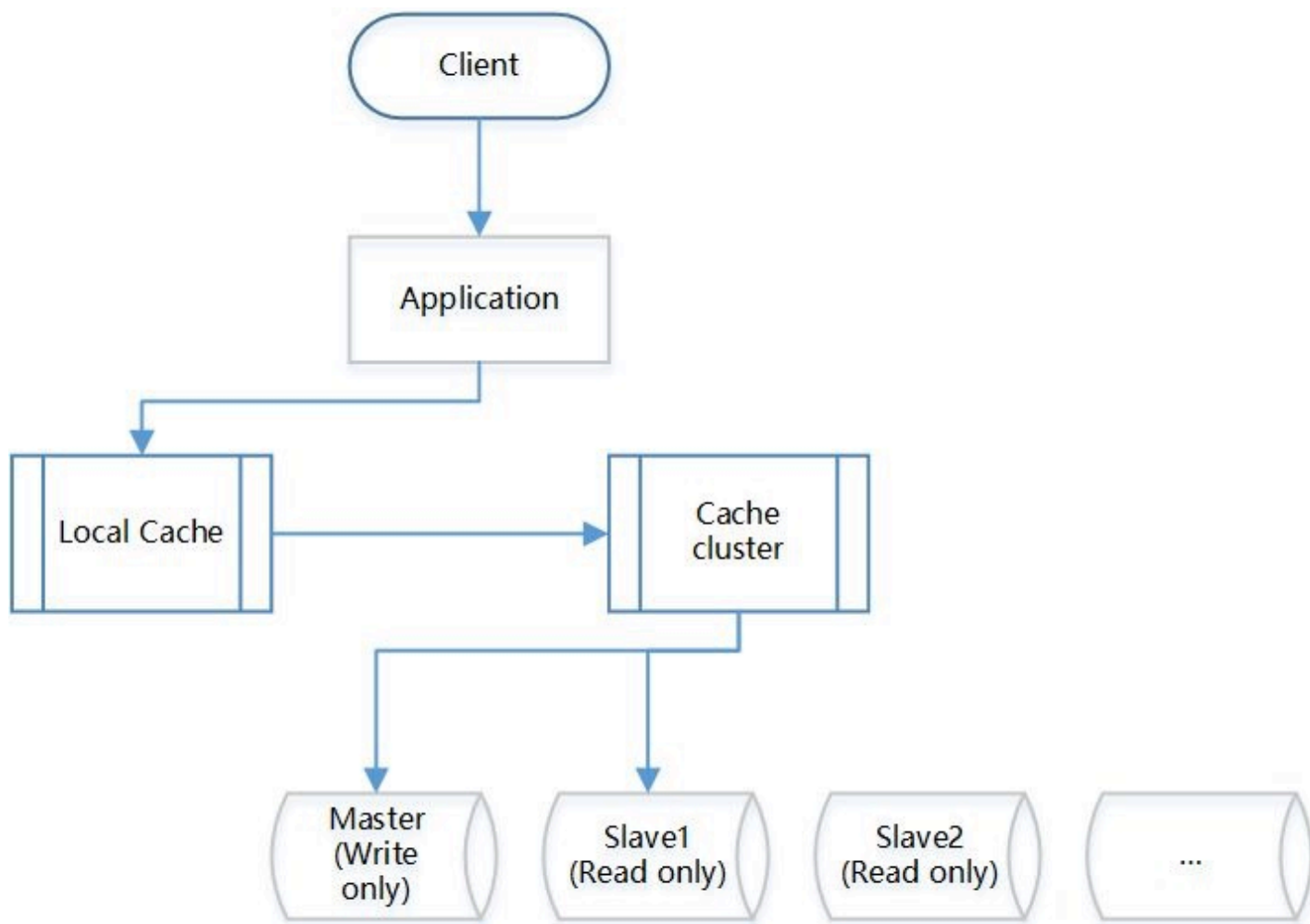
传统cpu通过fsb(前端总线)直连内存会造成内存等待，导致cpu吞吐量下降，内存成了性能瓶颈。同时又由于内存访问的热点数据集中性，所以需要在cpu与内存之间做一层临时的存储器作为高速缓存。

随着系统复杂性的提升，这种高速缓存和内存之间的速度进一步拉开，由于技术难度和成本等原因，所以有了更大的二级、三级缓存。根据读取顺序，绝大多数的请求首先落在一级缓存上，其次二级...

cpu core1	
L1d（一级数据缓存）	L1i（一级指令缓存）
L2	
L3	

## 缓存分布式，做多级缓存

---



## 1.读请求是写缓存

写缓存时一级一级写，先写本地缓存，再写集中式缓存。具体些缓存的方法可以有很多种，但是需要注意几项原则：

- 不要复制粘贴，避免重复代码
- 切忌和业务耦合太紧，不利于后期维护
- 为配合调试设置缓存开关要统一管理

## 2.写缓存失败了怎么办？应该先写缓存还是数据库呢？

大部分缓存应用的场景是读写比差异很大的，读远大于写，在这种场景下，只需要以数据库为主，先写数据库，再写缓存就好了。

## 3.其他性能发面

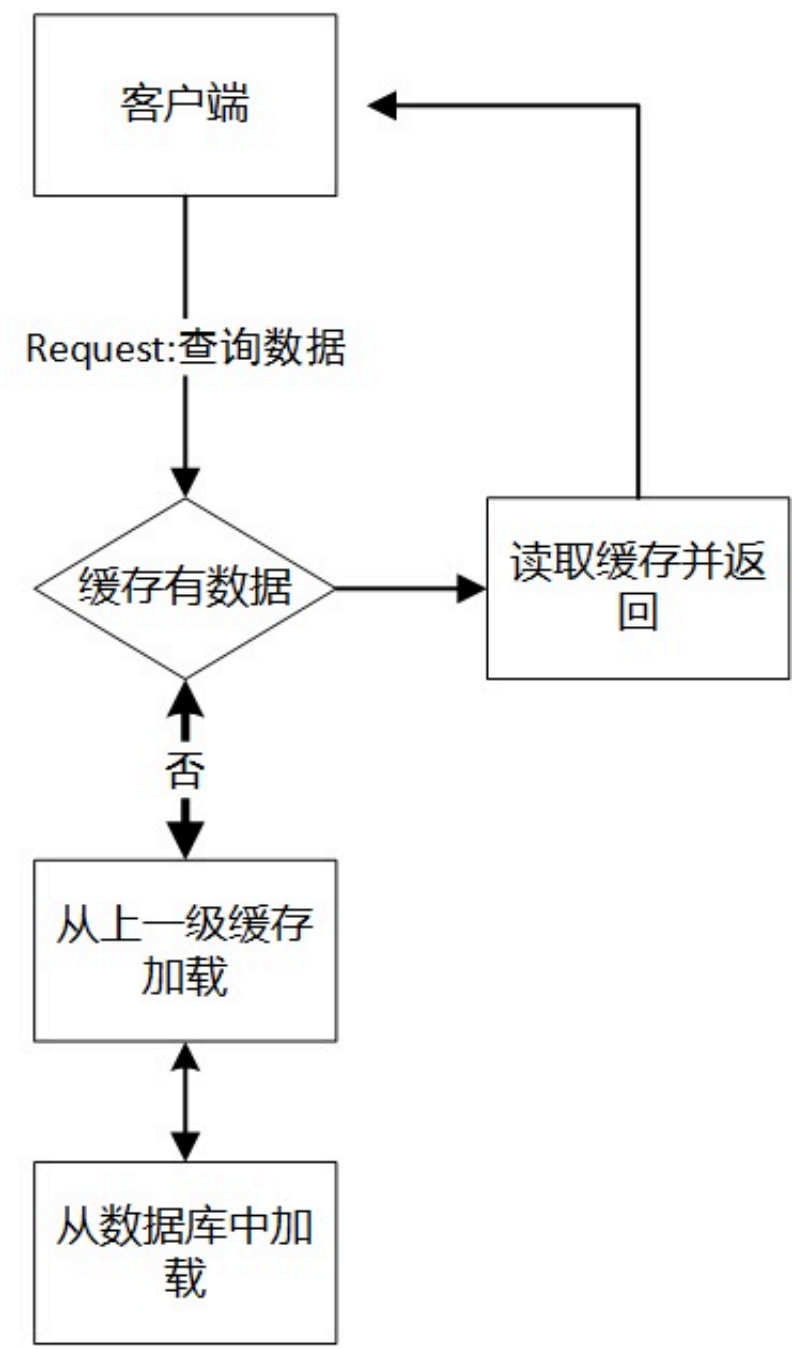
缓存设计都是占用越少越好，内存资源昂贵以及太大不好维护都驱使我们这样设计。所以要尽可能减少缓存不必要的数据，有的同学图省事把整个对象序列化存储。另外，序列化与反序列化也是消耗性能的。

# 各种缓存同步方案

缓存同步方案有很多种，在考虑一致性、数据库访问压力、实时性等方面做权衡。总的来说有以下几种方式：

## 1. 懒惰加载

如上段提到的方式，读时顺便加载。为了更新缓存数据，需要过期缓存。

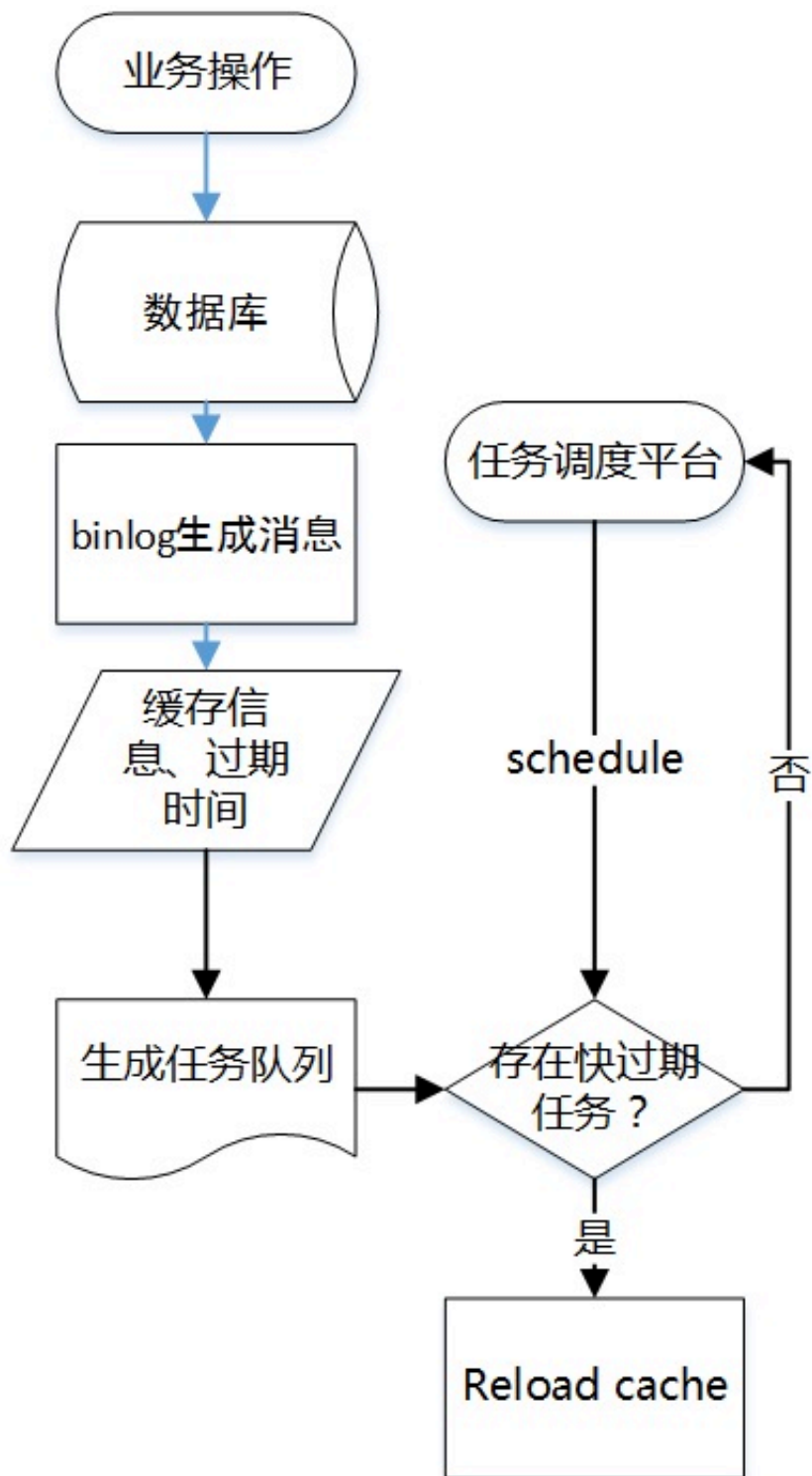


优点 \* 简单直接

缺点 \* 会造成一次缓存不命中 \* 这样当用户并发很大时，恰好缓存中无数据，数据库承担瞬时流量过大会造成风险。

## 2.补充式

可以在缓存时，把过期时间等信息写到一个异步队列里，后台起个线程池定期扫描这个队列，在快过期时主动reload缓存，使得数据会一直保持在缓存中，如果缓存没有也没有必要去数据库查询了。常见的处理方式有使用binlog加工成消息供增量处理。



- 优点：刷新缓存变为异步的任务，对数据库的压力瞬间由于任务队列的介入而降低了，削平并发的波峰。
- 缺点：消息一旦积压会造成同步延迟，引入复杂度。

### 3.定时加载

这就需要有个异步线程池定期把数据库的数据刷到集中式缓存，如redis里。

- 优点：保证所有数据最小时间差同步到缓存中，延迟很低。
- 缺点：如补充式，需要一个任务调度框架，复杂度提升，且要保证任务的顺序。如果递进一步还想加载到本地缓存，就得本地应用自己起线程抓取，方案维护成本高。可以考虑使用mq或者其他异步任务调度框架。
- ps：为了防止队列过大调度出现问题，处理完的数据要尽快结转，且要对积压数据以及写入情况做监控。

## 防止缓存穿透

---

缓存穿透是指查询的key压根不存在，从而缓存查询不到而查询了数据库。若是这样的key恰好并发请求很大，那么就会对数据库造成不必要的压力。怎么解决呢？

1. 把所有存在的key都存到另外一个存储的Set集合里，查询时可以先查询key是否存在。
2. 干脆简单一些，给查询不到的key也加一个标识空值的Value，这样就不会去查询数据库了，比如场景为查询省市区街道对应的移动营业厅，若是某街道确实没有移动营业厅，key规则不变，value可以设置为"0"等无意义的字符。当然此种方案要保证缓存集群的高可用。
3. 这些Key可能不是永远不存在，所以需要根据业务场景来设置过期时间。

## 热点缓存与缓存淘汰策略

---

有一些场景，需要只保持一部分的热点缓存，不需要全量缓存，比如热卖的商品信息，购买某类商品的热门商圈信息等等。

### 1.FIFO (First In, First Out)

先进先出，淘汰最早进来的缓存数据，一个标准的队列。以队列为基本数据结构，从队首进入新数据，从队尾淘汰。

### 2.LRU (Least RecentlyUsed)

最近最少使用，淘汰最近不使用的缓存数据。如果数据最近被访问过，则不淘汰。

1. 最近最少使用，淘汰最近不使用的缓存数据。如果数据最近被访问过，则不淘汰。
2. 最近时间被访问的数据移动到头部，实现算法有很多，如hashmap+双向链表等等；
3. 问题在于若是偶发性某些key被最近频繁访问，而非常态，则数据受到污染。

### 3.LFU (Least Frequently used)

最近使用次数最少的数据被淘汰，注意和LRU的区别在于LRU的淘汰规则是基于访问时间。

1. LFU中的每个数据块都有一个引用计数，数据块按照引用计数排序，若是恰好具有相同引用计数的数据块则按照时间排序；
2. 因为新加入的数据访问次数为1，所以插入到队列尾部；
3. 队列中的数据被新访问后，引用计数增加，队列重新排序；
4. 当需要淘汰数据时，将已经排序的列表最后的数据块删除；
5. 有很明显问题是若短时间内被频繁访问多次，比如访问异常或者循环没有控制住，而后很长时间未使用，则此数据会因为频率高而被错误的保留下来没有被淘汰。尤其对于新来的数据，由于其起始的次数是1，所以即便被正常使用也会因为比不过老的数据而被淘汰。所以维基百科说纯粹的LFU算法不经常单独使用而是组合在其他策略中使用。

## 缓存常见问题

---

Q：那么应该选择用本地缓存（local cache）还是集中式缓存（Cache cluster）呢？A：首先看数据量，看缓存更新的成本，如果整体缓存数据量不是很大，而且变化的不频繁，那么建议本地缓存。Q：如果不知道有哪些key怎么定期删除？A：拿redis来说keys \* 太损耗性能，不推荐。可以指定一个集合，把所有的key都存到这个集合里，然后对整个集合进行删除，这样便能完全清理了。Q：一个key包含的集合很大，redis无法做到内存空间上的均匀Shard？A：1、可以简单的设置key过期，这样就要允许有缓存不命中的情况；2、给key设置版本，比如为两天后的当前时间，然后读取缓存时用时间判断一下是否需要重新加载缓存，作为版本过期的策略