多级缓存设计

由 王梓晨 创建, 最后修改于2018-08-08

自古兵家多谋，《谋攻篇》，“故上兵伐谋，其次伐交，其次伐兵，其下攻城。攻城之法，为不得已”，可见攻城之计有很多种，而爬墙攻城是最不明智的做法，军队疲惫受损、钱粮损耗、百姓遭殃。故而我们有很多迂回之策，谋略、外交、军事手段等等，每一种都比攻城的代价小，更轻量级，缓存设计亦是如此。

为什么要设计缓存呢?

其实高并发应对的解决方案不是互联网独创的，计算机先祖们很早就对类似的场景做了方案。比如《计算机组成原理》这样提到的cpu缓存概念，它是一种高速缓存，容量比内存小但是速度却快很多，这种缓存的出现主要是为了解决cpu运算速度远大于内存读写速度，甚至达到千万倍。

传统的cpu通过fsb直连内存的方式显然就会因为内存访问的等待，导致cpu吞吐量下降，内存成为性能瓶颈。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...

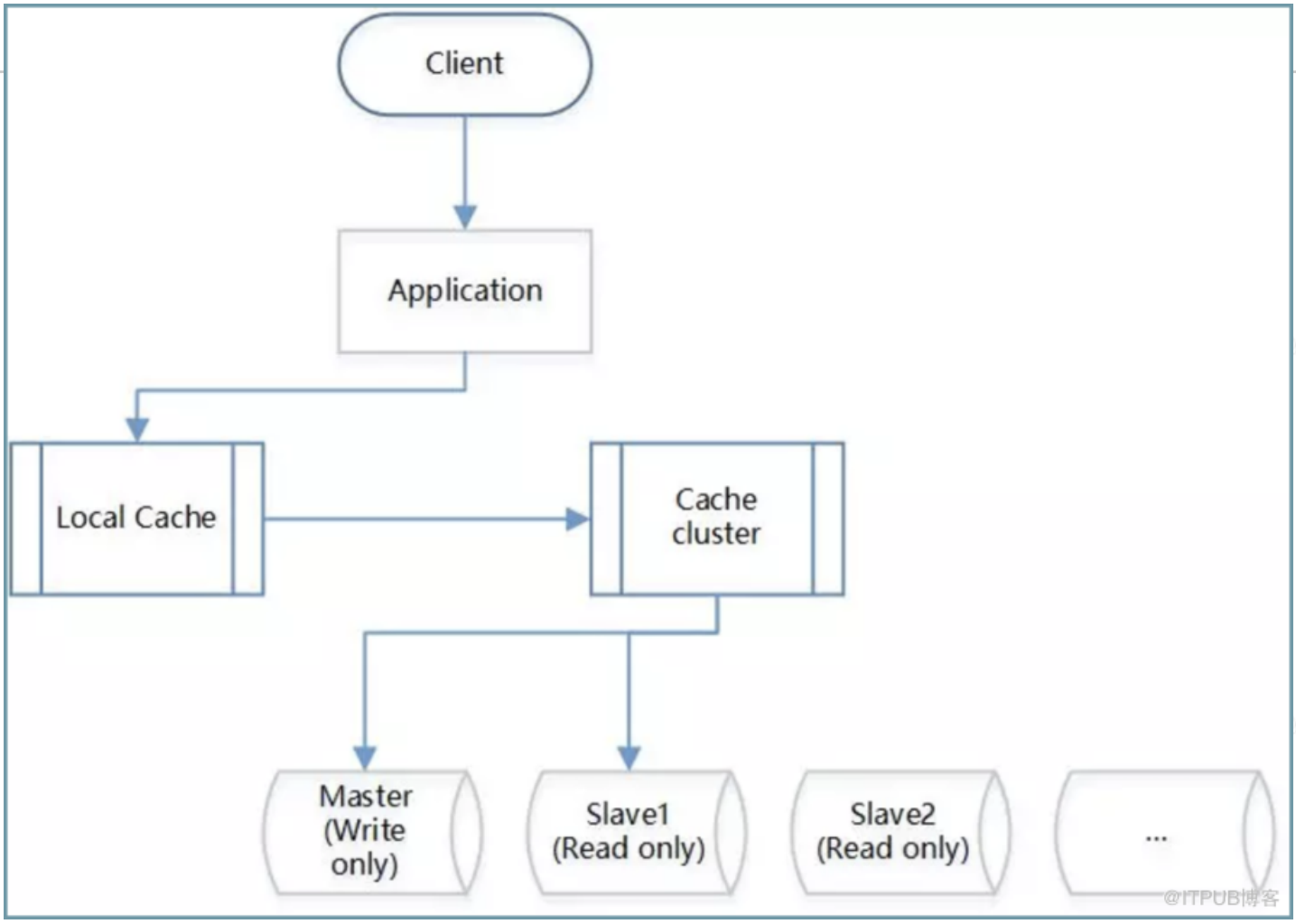
<table>
<tr><td colspan="2">cpu core1</td><td colspan="2">cpu core2</td></tr>
<tr><td>L1d
(一级数据缓存)</td><td>L1i
(一级指令缓存)</td><td>L1d
(一级数据缓存)</td><td>L1i
(一级指令缓存)</td></tr>
<tr><td colspan="2">L2</td><td colspan="2">L2</td></tr>
<tr><td colspan="2">L3</td><td colspan="2">L3</td></tr>
</table>

故而应用于SOA甚至微服务的场景，内存相当于存储业务数据的持久化数据库，其吞吐量肯定是远远小于缓存的，而对于java程序来讲，本地的jvm缓存优于集中式的redis缓存。

关系型数据库操作方便、易于维护且访问数据灵活，但是随着数据量的增加，其检索、更新的效率会越来越低。所以在高并发低延迟要求复杂的场景，要给数据库减负，减少其压力。

给数据库减负

缓存分布式，做多级缓存



1、读请求时写缓存

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：

1. 不要复制粘贴，避免重复代码
2. 切忌和业务耦合太紧，不利于后期维护
3. 开发初期刚刚上线阶段，为了排查问题，常常会给缓存设置开关，但是开关设置多了则会同时升高系统的复杂度，需要结合一套统一配置管理系统，京东物流有一套叫做UCC，且听下回分解.....

综上所述，高耦合带来的痛，弥补的代价是很大的，所以可以借鉴Spring cache来实现，实现也比较简单，使用时一个注解就搞定了。

```
 */
@Slf4j
@Service
@CacheConfig(cacheNames = "address:key:")
public class AddressKeyCacheService {

    private final static String LOG_PREFIX = "addressKey缓存服务[AddressKeyCacheService]";

    @Cacheable(key = "#addressKey")
    public String getJdByAddressKey(String addressKey){
        Slf4jLogUtil.infoLog(log, LOG_PREFIX + "查询缓存结果为null，addressKey = {}", addressKey);
        return null;
    }
```


2、写缓存失败了怎么办？应该先写缓存还是数据库呢?

既然是缓存的设计，那么策略一定是保证最终一致性，那么我们只需要采用异步消息来补偿就好了。

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。

最后补充一点，数据库出现异常时，不要一股脑的catch RuntimeException，而是把具体关心的异常往外抛，然后进行有针对性的异常处理。

3、关于其他性能方面

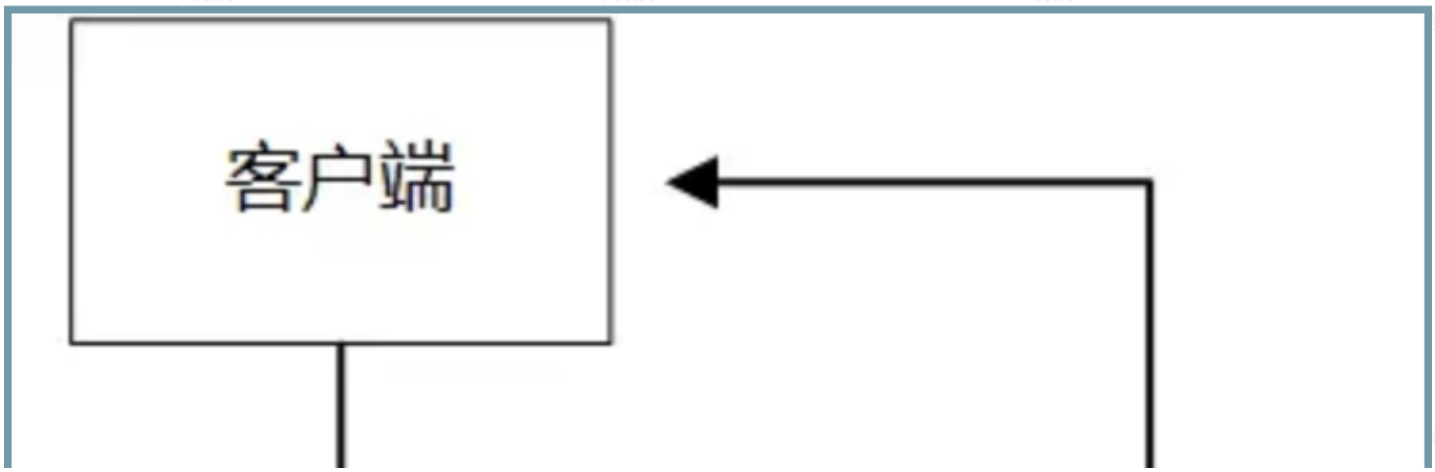
缓存设计都是占用越少越好，内存资源昂贵以及太大不好维护都驱使我们这样设计。所以要尽可能减少缓存不必要的数据，有的同学图省事把整个对象序列化存储。另外，序列化与反序列化也是消耗性能的。

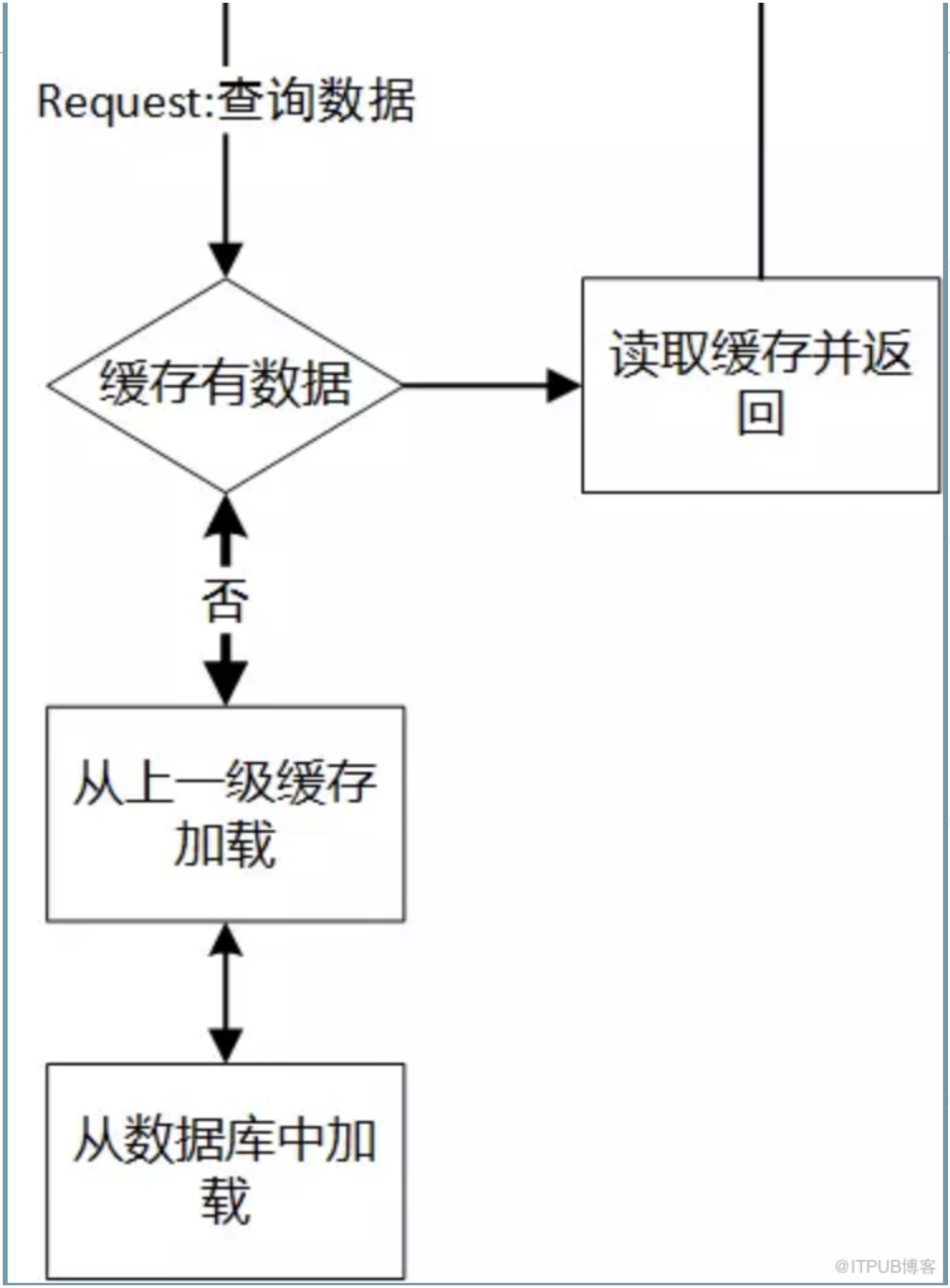
vs各种缓存同步方案

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、懒加载式

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。




Request:查询数据
缓存有数据
读取缓存并返回
否
从上一级缓存加载
从数据库中加载
@ITPUB博客

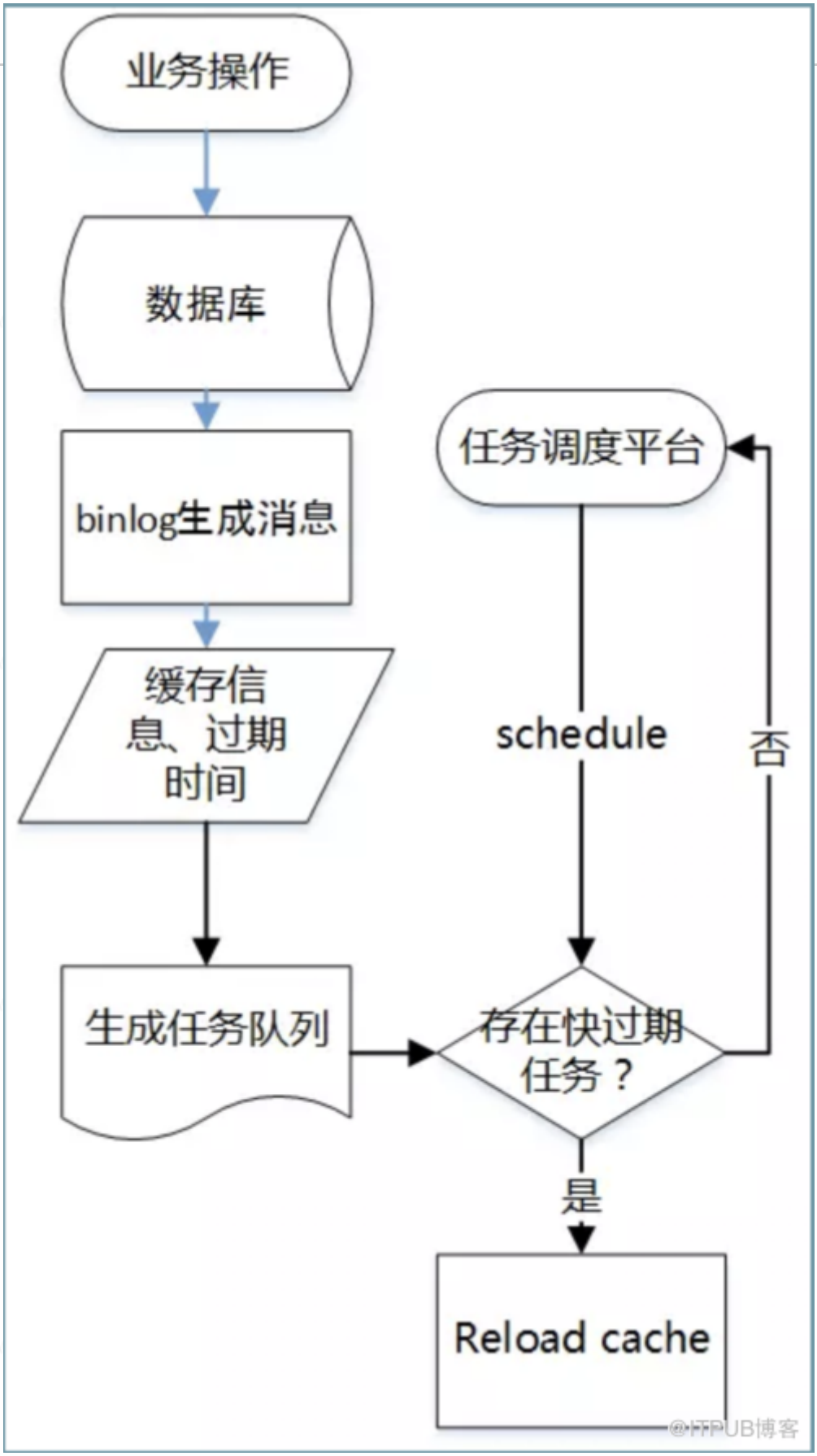
缺点：

- 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。

懒加载式太简单了，没有自动加载，异步刷新等机制，为了弥补其缺陷，请参见接下来的两种方法。

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、定时加载式

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。

综合来讲，缓存过期的策略有以下三种:

1、FIFO（First In，First Out)

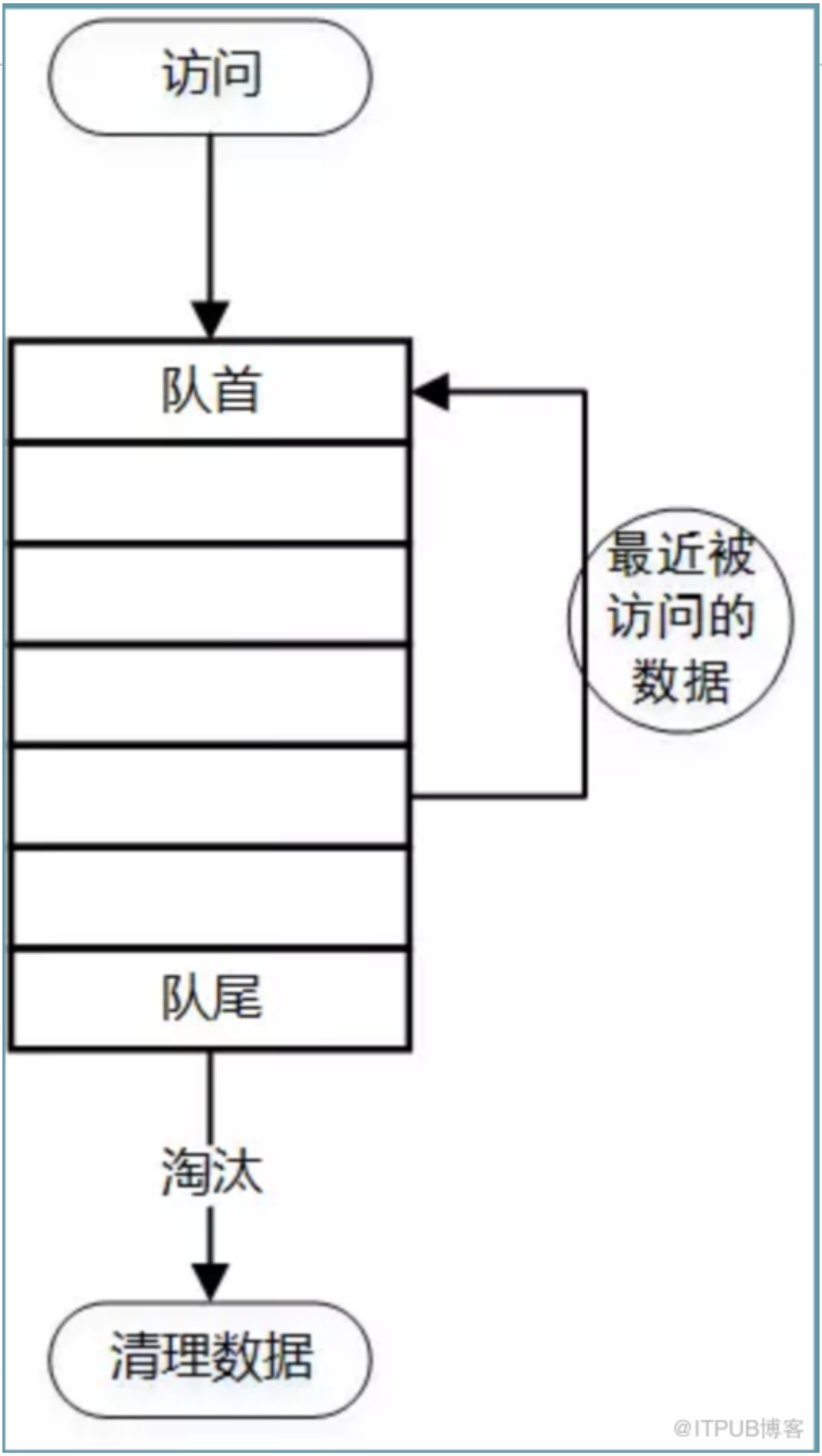
先进先出，淘汰最早进来的缓存数据，一个标准的队列。


访问
队首
队尾
淘汰
清理数据
@ITPUB博客

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）

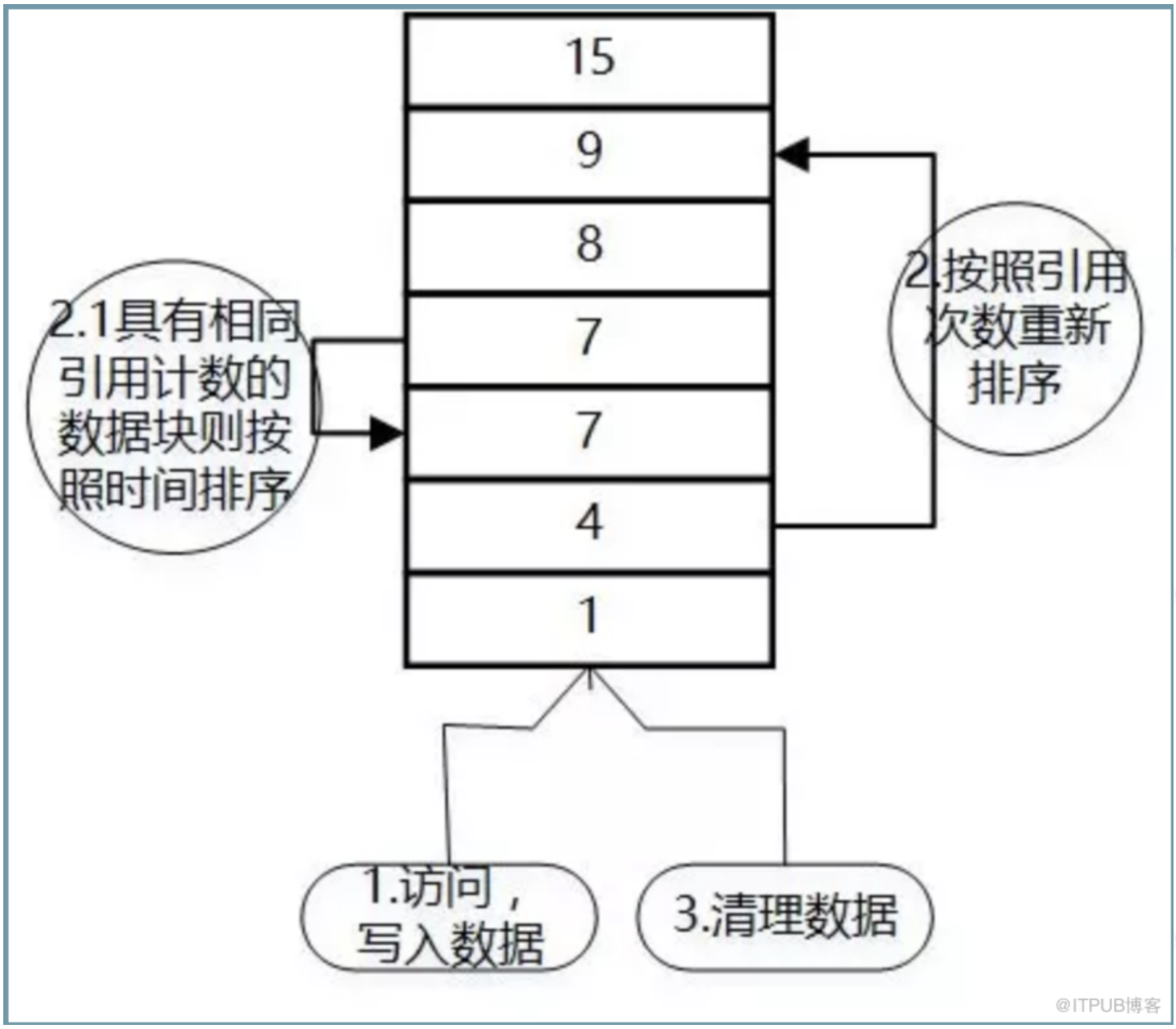
最近最少使用，淘汰最近不使用的缓存数据。如果数据最近被访问过，则不淘汰。



1. 和FIFO不同的是，需要对链表做基本模型，读写的时间复杂度是O（1），写入新数据进入头部，链表满了数据从尾部淘汰；
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LFU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 使用的一些常见问题

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：怎么批量更新一批缓存数据?

A：依次从数据库读取，然后批量写入缓存，批量更新，设置版本过期key或者主动删除。

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。

无标签