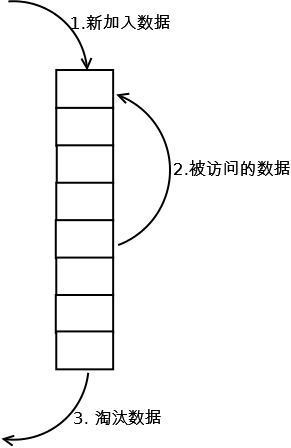
**1. LRU  
1.1. 原理**

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

**1.2. 实现**

最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：



**1. 新数据插入到链表头部；**

**2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；**

**3. 当链表满的时候，将链表尾部的数据丢弃。**

**1.3. 分析**

【命中率】

**当存在热点数据时，LRU的效率很好，**但偶发性的、周期性的批量操作会导致LRU命中率急剧下降，缓存污染情况比较严重。

【复杂度】

实现简单。

【代价】

**命中时需要遍历链表，找到命中的数据块索引，然后需要将数据移到头部。**

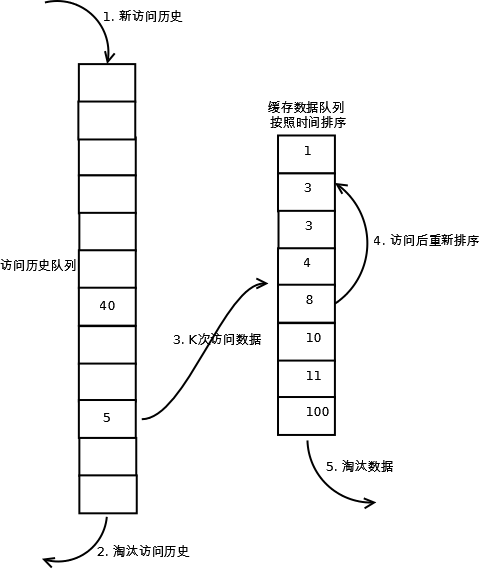
**2. LRU-K**

**2.1. 原理**

**LRU-K中的K代表最近使用的次数，因此LRU可以认为是LRU-1。LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题**，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。

**2.2. 实现**

**相比LRU，LRU-K需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到K次的时候，才将数据放入缓存。**当需要淘汰数据时，LRU-K会淘汰第K次访问时间距当前时间最大的数据。详细实现如下：



**1. 数据第一次被访问，加入到访问历史列表；**

**2. 如果数据在访问历史列表里后没有达到K次访问，则按照一定规则（FIFO，LRU）淘汰；**

**3. 当访问历史队列中的数据访问次数达到K次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；**

**4. 缓存数据队列中被再次访问后，重新排序；**

**5. 需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第K次访问离现在最久”的数据。**

LRU-K具有LRU的优点，**同时能够避免LRU的缺点，实际应用中LRU-2是综合各种因素后最优的选择，LRU-3或者更大的K值命中率会高，但适应性差，需要大量的数据访问才能将历史访问记录清除掉。**

**2.3. 分析**

【命中率】

LRU-K降低了“缓存污染”带来的问题，命中率比LRU要高。

【复杂度】

LRU-K队列是一个优先级队列，算法复杂度和代价比较高。

【代价】

由于LRU-K还需要记录那些被访问过、但还没有放入缓存的对象，因此内存消耗会比LRU要多；当数据量很大的时候，内存消耗会比较可观。

LRU-K需要基于时间进行排序（可以需要淘汰时再排序，也可以即时排序），CPU消耗比LRU要高。

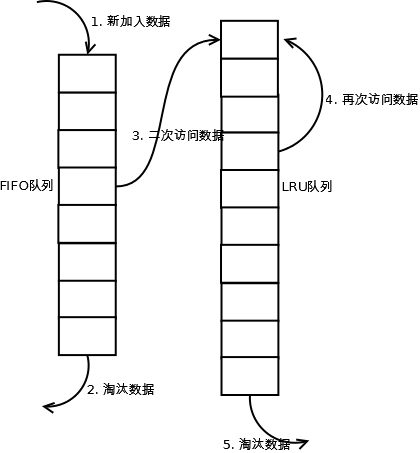
**3. Two queues（2Q）**

**3.1. 原理**

Two queues（以下使用2Q代替）算法类似于LRU-2，不同点在于**2Q将LRU-2算法中的访问历史队列（注意这不是缓存数据的）改为一个FIFO缓存队列**，即：**2Q算法有两个缓存队列，一个是FIFO队列，一个是LRU队列。**

**3.2. 实现**

当数据第一次访问时，**2Q算法将数据缓存在FIFO队列里面，当数据第二次被访问时，则将数据从FIFO队列移到LRU队列里面，两个队列各自按照自己的方法淘汰数据**。详细实现如下：



**1. 新访问的数据插入到FIFO队列；**

**2. 如果数据在FIFO队列中一直没有被再次访问，则最终按照FIFO规则淘汰；**

**3. 如果数据在FIFO队列中被再次访问，则将数据移到LRU队列头部；**

**4. 如果数据在LRU队列再次被访问，则将数据移到LRU队列头部；**

**5. LRU队列淘汰末尾的数据。**

注：上图中FIFO队列比LRU队列短，但并不代表这是算法要求，实际应用中两者比例没有硬性规定。

**3.3. 分析**

【命中率】

2Q算法的命中率要高于LRU。

【复杂度】

需要两个队列，但两个队列本身都比较简单。

【代价】

FIFO和LRU的代价之和。

2Q算法和LRU-2算法命中率类似，内存消耗也比较接近，但对于最后缓存的数据来说，2Q会减少一次从原始存储读取数据或者计算数据的操作。**（其实就是把K换成2而已。）**

**4. Multi Queue（MQ）**

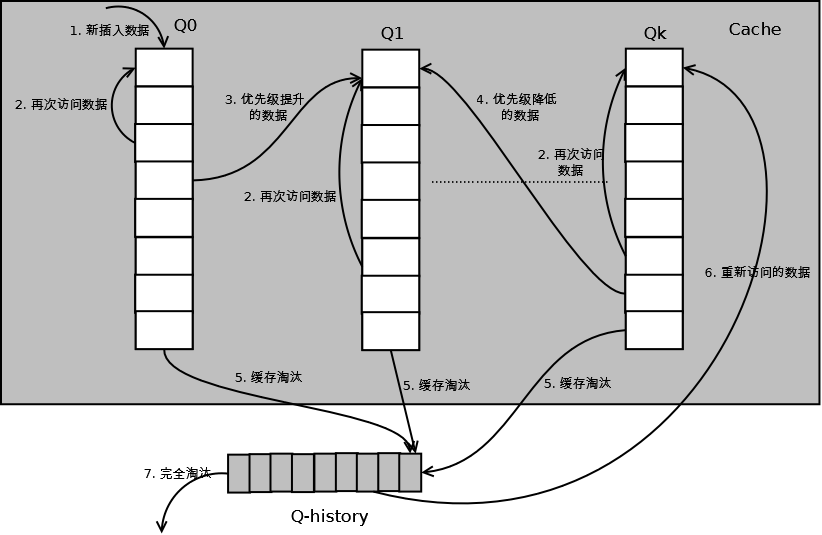
**4.1. 原理**

**MQ算法根据访问频率将数据划分为多个队列，不同的队列具有不同的访问优先级，**其核心思想是：优先缓存访问次数多的数据。

**4.2. 实现**

**MQ算法将缓存划分为多个LRU队列，每个队列对应不同的访问优先级。**访问优先级是根据访问次数计算出来的，例如

详细的算法结构图如下，Q0，Q1....Qk代表不同的优先级队列，Q-history代表从缓存中淘汰数据，但记录了数据的索引和引用次数的队列：



如上图，算法详细描述如下：

1. **新插入的数据放入Q0；**

**2. 每个队列按照LRU管理数据；**

**3. 当数据的访问次数达到一定次数，需要提升优先级时，将数据从当前队列删除，加入到高一级队列的头部；**

**4. 为了防止高优先级数据永远不被淘汰，当数据在指定的时间里访问没有被访问时，需要降低优先级，将数据从当前队列删除，加入到低一级的队列头部；**

**5. 需要淘汰数据时，从最低一级队列开始按照LRU淘汰；每个队列淘汰数据时，将数据从缓存中删除，将数据索引加入Q-history头部；**

**6. 如果数据在Q-history中被重新访问，则重新计算其优先级，移到目标队列的头部；**

**7. Q-history按照LRU淘汰数据的索引。**

**4.3. 分析**

【命中率】

MQ降低了“缓存污染”带来的问题，命中率比LRU要高。

【复杂度】

MQ需要维护多个队列，且需要维护每个数据的访问时间，复杂度比LRU高。

【代价】

MQ需要记录每个数据的访问时间，需要定时扫描所有队列，代价比LRU要高。

注：虽然MQ的队列看起来数量比较多，但由于所有队列之和受限于缓存容量的大小，因此这里多个队列长度之和和一个LRU队列是一样的，因此队列扫描性能也相近。

**5. LRU类算法对比**

由于不同的访问模型导致命中率变化较大，此处对比仅基于理论定性分析，不做定量分析。

|  |  |
| --- | --- |
| 对比点 | 对比 |
| 命中率 | LRU-2 > MQ(2) > 2Q > LRU |
| 复杂度 | LRU-2 > MQ(2) > 2Q > LRU |
| 代价 | LRU-2  > MQ(2) > 2Q > LRU |

实际应用中需要根据业务的需求和对数据的访问情况进行选择，并不是命中率越高越好。例如：虽然LRU看起来命中率会低一些，且存在”缓存污染“的问题，但由于其简单和代价小，实际应用中反而应用更多。

java中最简单的LRU算法实现，就是利用**jdk的LinkedHashMap，覆写其中的removeEldestEntry(Map.Entry)方法即可**

如果你去看LinkedHashMap的源码可知，LRU算法是通过双向链表来实现，**当某个位置被命中，通过调整链表的指向将该位置调整到头位置，新加入的内容直接放在链表头，如此一来，最近被命中的内容就向链表头移动，需要替换时，链表最后的位置就是最近最少使用的位置。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | import java.util.ArrayList;  import java.util.Collection;  import java.util.LinkedHashMap;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  import java.util.Map;      /\*\*   \* 类说明：利用LinkedHashMap实现简单的缓存， 必须实现removeEldestEntry方法，具体参见JDK文档   \*   \* @author dennis   \*   \* @param <K>   \* @param <V>   \*/  public class LRULinkedHashMap<K, V> extends LinkedHashMap<K, V> {      private final int maxCapacity;        private static final float DEFAULT\_LOAD\_FACTOR = 0.75f;        private final Lock lock = new ReentrantLock();        public LRULinkedHashMap(int maxCapacity) {          super(maxCapacity, DEFAULT\_LOAD\_FACTOR, true);          this.maxCapacity = maxCapacity;      }        @Override      protected boolean removeEldestEntry(java.util.Map.Entry<K, V> eldest) {          return size() > maxCapacity;      }      @Override      public boolean containsKey(Object key) {          try {              lock.lock();              return super.containsKey(key);          } finally {              lock.unlock();          }      }          @Override      public V get(Object key) {          try {              lock.lock();              return super.get(key);          } finally {              lock.unlock();          }      }        @Override      public V put(K key, V value) {          try {              lock.lock();              return super.put(key, value);          } finally {              lock.unlock();          }      }        public int size() {          try {              lock.lock();              return super.size();          } finally {              lock.unlock();          }      }        public void clear() {          try {              lock.lock();              super.clear();          } finally {              lock.unlock();          }      }        public Collection<Map.Entry<K, V>> getAll() {          try {              lock.lock();              return new ArrayList<Map.Entry<K, V>>(super.entrySet());          } finally {              lock.unlock();          }      }  } |

基于双链表 的LRU实现:

　　传统意义的LRU算法是为每一个Cache对象设置一个计数器，每次Cache命中则给计数器+1，而Cache用完，需要淘汰旧内容，放置新内容时，就查看所有的计数器，并将最少使用的内容替换掉。  
  
   它的弊端很明显，如果Cache的数量少，问题不会很大， 但是如果Cache的空间过大，达到10W或者100W以上，一旦需要淘汰，则需要遍历所有计算器，其性能与资源消耗是巨大的。效率也就非常的慢了。

    它的原理： 将Cache的所有位置都用双连表连接起来，当一个位置被命中之后，就将通过调整链表的指向，将该位置调整到链表头的位置，新加入的Cache直接加到链表头中。

     这样，在多次进行Cache操作后，最近被命中的，就会被向链表头方向移动，而没有命中的，而想链表后面移动，链表尾则表示最近最少使用的Cache。

     当需要替换内容时候，链表的最后位置就是最少被命中的位置，我们只需要淘汰链表最后的部分即可。

  上面说了这么多的理论， 下面用代码来实现一个LRU策略的缓存。

    我们用一个对象来表示Cache，并实现双链表，

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** LRUCache {
2. /\*\*
3. \* 链表节点
4. \* @author Administrator
5. \*
6. \*/
7. **class** CacheNode {
8. ……
9. }
10. **private** **int** cacheSize;//缓存大小
11. **private** Hashtable nodes;//缓存容器
12. **private** **int** currentSize;//当前缓存对象数量
13. **private** CacheNode first;//(实现双链表)链表头
14. **private** CacheNode last;//(实现双链表)链表尾
15. }

 下面给出完整的实现，这个类也被Tomcat所使用（ org.apache.tomcat.util.collections.LRUCache），但是在tomcat6.x版本中，已经被弃用，使用另外其他的缓存类来替代它。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** LRUCache {
2. /\*\*
3. \* 链表节点
4. \* @author Administrator
5. \*
6. \*/
7. **class** CacheNode {
8. CacheNode prev;//前一节点
9. CacheNode next;//后一节点
10. Object value;//值
11. Object key;//键
12. CacheNode() {
13. }
14. }
15. **public** LRUCache(**int** i) {
16. currentSize = 0;
17. cacheSize = i;
18. nodes = **new** Hashtable(i);//缓存容器
19. }
21. /\*\*
22. \* 获取缓存中对象
23. \* @param key
24. \* @return
25. \*/
26. **public** Object get(Object key) {
27. CacheNode node = (CacheNode) nodes.get(key);
28. **if** (node != **null**) {
29. moveToHead(node);
30. **return** node.value;
31. } **else** {
32. **return** **null**;
33. }
34. }
36. /\*\*
37. \* 添加缓存
38. \* @param key
39. \* @param value
40. \*/
41. **public** **void** put(Object key, Object value) {
42. CacheNode node = (CacheNode) nodes.get(key);
44. **if** (node == **null**) {
45. //缓存容器是否已经超过大小.
46. **if** (currentSize >= cacheSize) {
47. **if** (last != **null**)//将最少使用的删除
48. nodes.remove(last.key);
49. removeLast();
50. } **else** {
51. currentSize++;
52. }
54. node = **new** CacheNode();
55. }
56. node.value = value;
57. node.key = key;
58. //将最新使用的节点放到链表头，表示最新使用的.
59. moveToHead(node);
60. nodes.put(key, node);
61. }
62. /\*\*
63. \* 将缓存删除
64. \* @param key
65. \* @return
66. \*/
67. **public** Object remove(Object key) {
68. CacheNode node = (CacheNode) nodes.get(key);
69. **if** (node != **null**) {
70. **if** (node.prev != **null**) {
71. node.prev.next = node.next;
72. }
73. **if** (node.next != **null**) {
74. node.next.prev = node.prev;
75. }
76. **if** (last == node)
77. last = node.prev;
78. **if** (first == node)
79. first = node.next;
80. }
81. **return** node;
82. }
83. **public** **void** clear() {
84. first = **null**;
85. last = **null**;
86. }
87. /\*\*
88. \* 删除链表尾部节点
89. \*  表示 删除最少使用的缓存对象
90. \*/
91. **private** **void** removeLast() {
92. //链表尾不为空,则将链表尾指向null. 删除连表尾（删除最少使用的缓存对象）
93. **if** (last != **null**) {
94. **if** (last.prev != **null**)
95. last.prev.next = **null**;
96. **else**
97. first = **null**;
98. last = last.prev;
99. }
100. }
102. /\*\*
103. \* 移动到链表头，表示这个节点是最新使用过的
104. \* @param node
105. \*/
106. **private** **void** moveToHead(CacheNode node) {
107. **if** (node == first)
108. **return**;
109. **if** (node.prev != **null**)
110. node.prev.next = node.next;
111. **if** (node.next != **null**)
112. node.next.prev = node.prev;
113. **if** (last == node)
114. last = node.prev;
115. **if** (first != **null**) {
116. node.next = first;
117. first.prev = node;
118. }
119. first = node;
120. node.prev = **null**;
121. **if** (last == **null**)
122. last = first;
123. }
124. **private** **int** cacheSize;
125. **private** Hashtable nodes;//缓存容器
126. **private** **int** currentSize;
127. **private** CacheNode first;//链表头
128. **private** CacheNode last;//链表尾
129. }<br style="margin: 0px; padding: 0px;"><br style="margin: 0px; padding: 0px;">