1. 页面置换算法（RT：LRU时间有关，也就最近最久未使用；LFU是最近最少使用）

# **[缓存算法（FIFO 、LRU、LFU三种算法的区别）](https://www.cnblogs.com/hongdada/p/10406902.html)**

## **FIFO算法[#](https://www.cnblogs.com/hongdada/p/10406902.html" \l "123286832)**

FIFO 算法是一种比较容易实现的算法。它的思想是先进先出（FIFO，队列），这是最简单、最公平的一种思想，即**如果一个数据是最先进入的，那么可以认为在将来它被访问的可能性很小。空间满的时候，最先进入的数据会被最早置换（淘汰）掉**。

FIFO 算法的描述：设计一种缓存结构，该结构在构造时确定大小，假设大小为 K，并有两个功能：

1. set(key,value)：将记录(key,value)插入该结构。当缓存满时，将最先进入缓存的数据置换掉。
2. get(key)：返回key对应的value值。

实现：维护一个FIFO队列，按照时间顺序将各数据（已分配页面）链接起来组成队列，并将置换指针指向队列的队首。再进行置换时，只需把置换指针所指的数据（页面）顺次换出，并把新加入的数据插到队尾即可。

缺点：判断一个页面置换算法优劣的指标就是缺页率，而FIFO算法的一个显著的缺点是，在某些特定的时刻，缺页率反而会随着分配页面的增加而增加，这称为**Belady现象**。产生Belady现象现象的原因是，FIFO置换算法与进程访问内存的动态特征是不相容的，被置换的内存页面往往是被频繁访问的，或者没有给进程分配足够的页面，因此FIFO算法会使一些页面频繁地被替换和重新申请内存，从而导致缺页率增加。因此，**现在不再使用FIFO算法**。

## **LRU算法[#](https://www.cnblogs.com/hongdada/p/10406902.html" \l "3012820330)**

LRU（The Least Recently Used，最近最久未使用算法）是一种常见的缓存算法，在很多分布式缓存系统（如Redis, Memcached）中都有广泛使用。

LRU算法的思想是：**如果一个数据在最近一段时间没有被访问到，那么可以认为在将来它被访问的可能性也很小。因此，当空间满时，最久没有访问的数据最先被置换（淘汰）**。

LRU算法的描述： 设计一种缓存结构，该结构在构造时确定大小，假设大小为 K，并有两个功能：

1. set(key,value)：将记录(key,value)插入该结构。当缓存满时，将最久未使用的数据置换掉。
2. get(key)：返回key对应的value值。

实现：最朴素的思想就是用数组+时间戳的方式，不过这样做效率较低。因此，我们可以用双向链表（LinkedList）+哈希表（HashMap）实现（链表用来表示位置，哈希表用来存储和查找），在Java里有对应的数据结构**LinkedHashMap**。

### **LInkedHashMap[#](https://www.cnblogs.com/hongdada/p/10406902.html" \l "4156628682)**

利用Java的LinkedHashMap用非常简单的代码来实现基于LRU算法的Cache功能

Copy

import java.util.LinkedHashMap;import java.util.Map;/\*\*

\* 简单用LinkedHashMap来实现的LRU算法的缓存

\*/

public class LRUCache<K, V> extends LinkedHashMap<K, V> {

private int cacheSize;

public LRUCache(int cacheSize) {

super(16, (float) 0.75, true);

this.cacheSize = cacheSize;

}

protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {

return size() > cacheSize;

}

}

测试：

Copy

import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public class LRUCacheTest {

private static final Logger log = LoggerFactory.getLogger(LRUCacheTest.class);

private static LRUCache<String, Integer> cache = new LRUCache<>(10);

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

cache.put("k" + i, i);

}

log.info("all cache :'{}'",cache);

cache.get("k3");

log.info("get k3 :'{}'", cache);

cache.get("k4");

log.info("get k4 :'{}'", cache);

cache.get("k4");

log.info("get k4 :'{}'", cache);

cache.put("k" + 10, 10);

log.info("After running the LRU algorithm cache :'{}'", cache);

}

}

Output:

Copy

all cache :'{k0=0, k1=1, k2=2, k3=3, k4=4, k5=5, k6=6, k7=7, k8=8, k9=9}'get k3 :'{k0=0, k1=1, k2=2, k4=4, k5=5, k6=6, k7=7, k8=8, k9=9, k3=3}'get k4 :'{k0=0, k1=1, k2=2, k5=5, k6=6, k7=7, k8=8, k9=9, k3=3, k4=4}'get k4 :'{k0=0, k1=1, k2=2, k5=5, k6=6, k7=7, k8=8, k9=9, k3=3, k4=4}'After running the LRU algorithm cache :'{k1=1, k2=2, k5=5, k6=6, k7=7, k8=8, k9=9, k3=3, k4=4, k10=10}'

## **LFU算法[#](https://www.cnblogs.com/hongdada/p/10406902.html" \l "2437823013)**

LFU（Least Frequently Used ，最近最少使用算法）也是一种常见的缓存算法。

顾名思义，LFU算法的思想是：**如果一个数据在最近一段时间很少被访问到，那么可以认为在将来它被访问的可能性也很小。因此，当空间满时，最小频率访问的数据最先被淘汰**。

LFU 算法的描述：

设计一种缓存结构，该结构在构造时确定大小，假设大小为 K，并有两个功能：

1. set(key,value)：将记录(key,value)插入该结构。当缓存满时，将访问频率最低的数据置换掉。
2. get(key)：返回key对应的value值。

算法实现策略：考虑到 LFU 会淘汰访问频率最小的数据，我们需要一种合适的方法按大小顺序维护数据访问的频率。LFU 算法本质上可以看做是一个 top K 问题(K = 1)，即选出频率最小的元素，因此我们很容易想到可以用二项堆来选择频率最小的元素，这样的实现比较高效。最终实现策略为小顶堆+哈希表