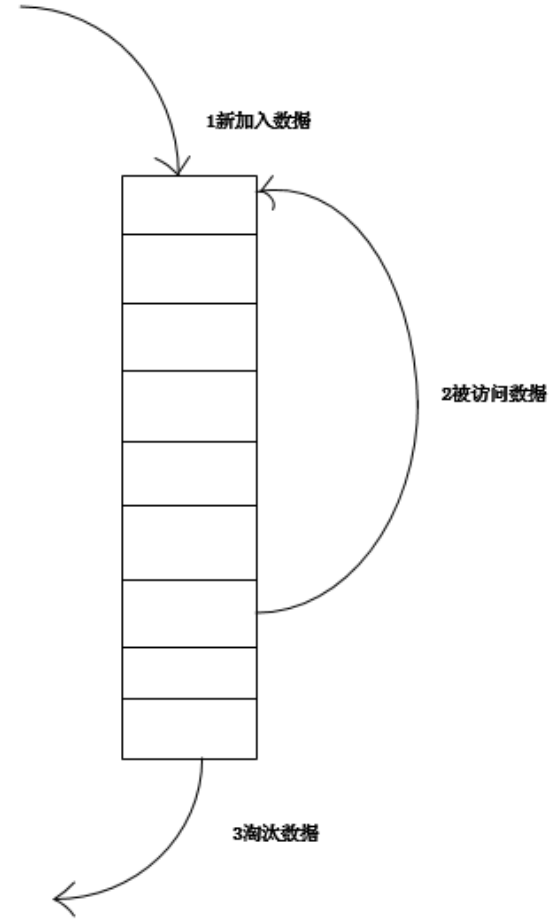
LRU

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”。

实现1

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。

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。

```
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();
        }
    }
```

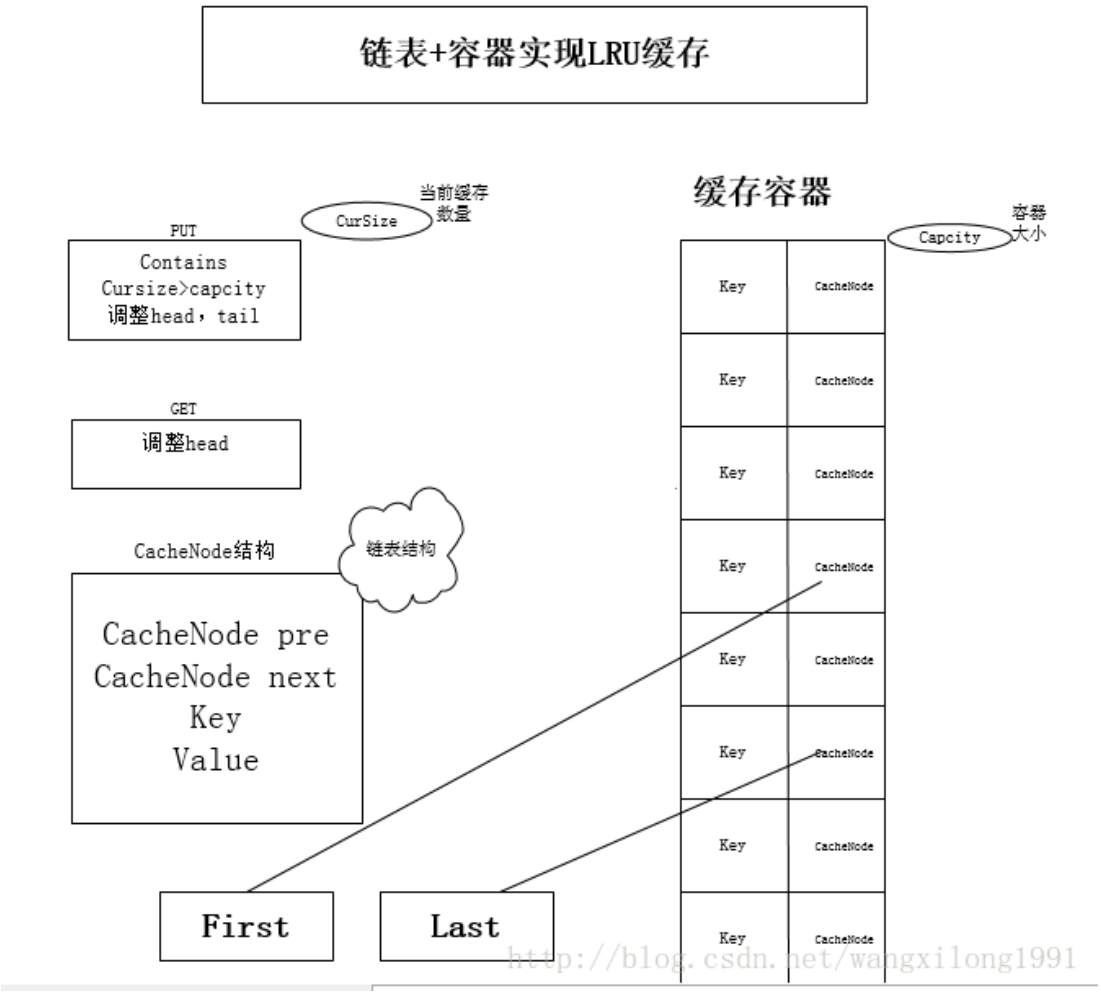
```
    @Override
    public V get(Object key) {
        try {
            lock.lock();
            return super.get(key);
        } finally {
            lock.unlock();
        }
    }

    @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();
            }
        }
    }
```

实现2

LRUCache的链表+HashMap实现

Southeast



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。
非线程安全，若实现安全，则在响应的方法加锁。

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;


public class LRUCache<K, V> {

    private int currentCacheSize;
    private int CacheCapcity;
    private HashMap<K,CacheNode> caches;
    private CacheNode first;
    private CacheNode last;

    public LRUCache(int size){
        currentCacheSize = 0;
        this.CacheCapcity = size;
        caches = new HashMap<K,CacheNode>(size);
    }

    public void put(K k,V v){
        CacheNode node = caches.get(k);
        if(node == null){
            if(caches.size() >= CacheCapcity){

                caches.remove(last.key);
                removeLast();
            }
            node = new CacheNode();
            node.key = k;
        }
        node.value = v;
        moveToFirst(node);
        caches.put(k, node);
    }

    public Object  get(K k){
        CacheNode node = caches.get(k);
        if(node == null){
```

```
            return null;
        }
        moveToFirst(node);
        return node.value;
    }

    public Object remove(K k){
        CacheNode node = caches.get(k);
        if(node != null){
            if(node.pre != null){
                node.pre.next=node.next;
            }
            if(node.next != null){
                node.next.pre=node.pre;
            }
            if(node == first){
                first = node.next;
            }
            if(node == last){
                last = node.pre;
            }
        }

        return caches.remove(k);
    }

    public void clear(){
        first = null;
        last = null;
        caches.clear();
    }



    private void moveToFirst(CacheNode node){
        if(first == node){
            return;
        }
        if(node.next != null){
            node.next.pre = node.pre;
        }
        if(node.pre != null){
            node.pre.next = node.next;
        }
        if(node == last){
```

```
            last= last.pre;
        }
        if(first == null || last == null){
            first = last = node;
            return;
        }

        node.next=first;
        first.pre = node;
        first = node;
        first.pre=null;

    }

    private void removeLast(){
        if(last != null){
            last = last.pre;
            if(last == null){
                first = null;
            }else{
                last.next = null;
            }
        }
    }
    @Override
    public String toString(){
        StringBuilder sb = new StringBuilder();
        CacheNode node = first;
        while(node != null){
            sb.append(String.format("%s:%s ", node.key,node.value));
            node = node.next;
        }

        return sb.toString();
    }

    class CacheNode{
        CacheNode pre;
        CacheNode next;
        Object key;
        Object value;
        public CacheNode(){

        }
    }
```

```
    public static void main(String[] args) {

        LRUCache<Integer,String> lru = new LRUCache<Integer,String>(3);

        lru.put(1, "a");    // 1:a
        System.out.println(lru.toString());
        lru.put(2, "b");    // 2:b 1:a
        System.out.println(lru.toString());
        lru.put(3, "c");    // 3:c 2:b 1:a
        System.out.println(lru.toString());
        lru.put(4, "d");    // 4:d 3:c 2:b
        System.out.println(lru.toString());
        lru.put(1, "aa");   // 1:aa 4:d 3:c
        System.out.println(lru.toString());
        lru.put(2, "bb");   // 2:bb 1:aa 4:d
        System.out.println(lru.toString());
        lru.put(5, "e");    // 5:e 2:bb 1:aa
        System.out.println(lru.toString());
        lru.get(1);         // 1:aa 5:e 2:bb
        System.out.println(lru.toString());
        lru.remove(11);     // 1:aa 5:e 2:bb
        System.out.println(lru.toString());
        lru.remove(1);      //5:e 2:bb
        System.out.println(lru.toString());
        lru.put(1, "aaa");  //1:aaa 5:e 2:bb
        System.out.println(lru.toString());
    }

}
```