利用LinkedHashMap实现LRU算法

```
package com.lza.outofmemory;

import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;

public class LRUCache<K,V> extends LinkedHashMap<K,V>{
 /**
  *定义缓存数
  */
 final int MAX_CACHE_SIZE;
 static final int MAXIMUM_CAPACITY = 1 << 30;
 public LRUCache(int cacheSize){
  /**
   * 初始化LinkedHashMap的构造函数，最后一个参数为true表示按访问顺序排序 即最近访问的在最前面 最少访问的在最后面
   */
  super((int) (((cacheSize/0.75))+1),0.75f,true);
  this.MAX_CACHE_SIZE=cacheSize;
 }
 public LRUCache(){
  super(16,0.75f,true);
  this.MAX_CACHE_SIZE=16;
 }
 /**
  * 重写removeEldestEntry()方法，告诉它什么时候需要删除容器中"最老"的元素:当参数LinkedHashMap中的accessOrder参数为true
时
  * 最老元素表示最近最少使用的元素，当accessOrder参数为false时，最老元素表示最早插入的元素。
  */
 @Override
 protected boolean removeEldestEntry(Map.Entry<K, V> eldest){
  if(size()>MAX_CACHE_SIZE){
   System.out.println("Key:"+eldest.getKey()+"    Value:"+eldest.getValue()+"  has been removed.");
   return true;
  }else{
   return false;
  }
 }
 public static void main(String[]args){
  //Map<Integer,String> LRUCacheTest=Collections.synchronizedMap(new LRUCache(5)); 多线程环境下使用，确保线程安全
  Map<Integer,String> LRUCacheTest=new LRUCache(5);
  /**
   * 创建5个 占满缓存
   */
  for(Integer i=0;i<5;i++){
   LRUCacheTest.put(i, i.toString());
  }
  Iterator ite=LRUCacheTest.entrySet().iterator();
  System.out.println("原始的K-V键值对：");
  while(ite.hasNext()){
   Map.Entry<Integer, String> temp=(Map.Entry<Integer, String>)ite.next();
   System.out.println("Key:"+temp.getKey()+"    Value:"+temp.getValue());
  }
  /**
   * 刷新 0 和 2 的缓存
   */
  LRUCacheTest.put(0, "0");
  LRUCacheTest.put(2, "2");
  /**
   * 添加两个新的缓存 使最近两个最少使用的被移除
   */
  System.out.println("被移除的K-V键值对：");
  LRUCacheTest.put(5, "5");
```

```
  LRUCacheTest.put(6, "6");
  System.out.println("移除后缓存中的数据排列：");
  Iterator ite2=LRUCacheTest.entrySet().iterator();
  while(ite2.hasNext()){
   Map.Entry<Integer, String> temp=(Map.Entry<Integer, String>)ite2.next();
   System.out.println("Key:"+temp.getKey()+"    Value:"+temp.getValue());
  }
 }
}
```

运行结果：

```
原始的K-V键值对：
Key:0    Value:0
Key:1    Value:1
Key:2    Value:2
Key:3    Value:3
Key:4    Value:4
被移除的K-V键值对：
Key:1    Value:1  has been removed.
Key:3    Value:3  has been removed.
移除后缓存中的数据排列：
Key:4    Value:4
Key:0    Value:0
Key:2    Value:2
Key:5    Value:5
Key:6    Value:6
```

如果要实现FIFO算法进行缓存管理则更简单，只需要重写removeEldestEntry方法即可。