LinkedHashMap源码解析

```
public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
{

    private static final long serialVersionUID = 3801124242820219131L;

    /**
     * 双向链表头结点.
     */
    private transient Entry<K,V> header;

    /**
     * 排序标志 accessOrder为true时表示按最近访问排序(最近访问的在链表尾部)，为false表示按插入顺序排序（默认方式）。
     */
    private final boolean accessOrder;

    /**
     * 构造函数 可设置初始容量 负载因子
     */
    public LinkedHashMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
        accessOrder = false;
    }

    /**
     * 构造函数可设置初始容量
     */
    public LinkedHashMap(int initialCapacity) {
 super(initialCapacity);
        accessOrder = false;
    }

    /**
     * 无参构造函数
     */
    public LinkedHashMap() {
 super();
        accessOrder = false;
    }

    /**
     * 构造函数 根据已有集体初始化
     */
    public LinkedHashMap(Map<? extends K, ? extends V> m) {
        super(m);
        accessOrder = false;
    }

    /**
     * 构造函数 可设置初始容量 负载因子 排序方式
     */
    public LinkedHashMap(int initialCapacity,
    float loadFactor,
                         boolean accessOrder) {
        super(initialCapacity, loadFactor);
        this.accessOrder = accessOrder;
    }

    /**
     * 初始化链表
     */
    void init() {
        header = new Entry<K,V>(-1, null, null, null);
        header.before = header.after = header;
    }

    /**
     * 转移链表元素到新的链表，扩容时使用
     */
    void transfer(HashMap.Entry[] newTable) {
        int newCapacity = newTable.length;
        for (Entry<K,V> e = header.after; e != header; e = e.after) {
            int index = indexFor(e.hash, newCapacity);
            e.next = newTable[index];
            newTable[index] = e;
        }
    }


    /**
     * 判断是否含有某个值
     */
    public boolean containsValue(Object value) {
        // Overridden to take advantage of faster iterator
        if (value==null) {
            for (Entry e = header.after; e != header; e = e.after)
                if (e.value==null)
                    return true;
        } else {
            for (Entry e = header.after; e != header; e = e.after)
                if (value.equals(e.value))
                    return true;
        }
        return false;
    }

    /**
     * 获取Value
     */
    public V get(Object key) {
        Entry<K,V> e = (Entry<K,V>)getEntry(key);
        if (e == null)
            return null;
        e.recordAccess(this);
        return e.value;
    }

    /**
     * Removes all of the mappings from this map.
     * The map will be empty after this call returns.
     */
    public void clear() {
        super.clear();
        header.before = header.after = header;
    }

    /**
     * LinkedHashMap entry.
     */
    private static class Entry<K,V> extends HashMap.Entry<K,V> {
        // These fields comprise the doubly linked list used for iteration.
        Entry<K,V> before, after;

 Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {
            super(hash, key, value, next);
        }

        /**
         * 从链表中移除
         */
    private void remove() {
      before.after = after;
      after.before = before;
    }

    /**
     * 在existingEntry结点之前插入结点
     */
    private void addBefore(Entry<K,V> existingEntry) {
      after  = existingEntry;
      before = existingEntry.before;
      before.after = this;
      after.before = this;
    }

    /**
     * 当accessOrder为true且结点被访问或修改时将其移到链表的尾部
     */
    void recordAccess(HashMap<K,V> m) {
      LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
      if (lm.accessOrder) {
        lm.modCount++;
        remove();
        addBefore(lm.header);
      }
    }

    void recordRemoval(HashMap<K,V> m) {
      remove();
    }
  }

  private abstract class LinkedHashIterator<T> implements Iterator<T> {
Entry<K,V> nextEntry    = header.after;
Entry<K,V> lastReturned = null;

/**
 * The modCount value that the iterator believes that the backing
 * List should have.  If this expectation is violated, the iterator
 * has detected concurrent modification.
 */
int expectedModCount = modCount;

public boolean hasNext() {
      return nextEntry != header;
}

public void remove() {
  if (lastReturned == null)
 throw new IllegalStateException();
  if (modCount != expectedModCount)
 throw new ConcurrentModificationException();

      LinkedHashMap.this.remove(lastReturned.key);
      lastReturned = null;
      expectedModCount = modCount;
}

Entry<K,V> nextEntry() {
  if (modCount != expectedModCount)
 throw new ConcurrentModificationException();
      if (nextEntry == header)
        throw new NoSuchElementException();

      Entry<K,V> e = lastReturned = nextEntry;
      nextEntry = e.after;
      return e;
}
```

```
    }

    private class KeyIterator extends LinkedHashIterator<K> {
 public K next() { return nextEntry().getKey(); }
    }

    private class ValueIterator extends LinkedHashIterator<V> {
 public V next() { return nextEntry().value; }
    }

    private class EntryIterator extends LinkedHashIterator<Map.Entry<K,V>> {
 public Map.Entry<K,V> next() { return nextEntry(); }
    }

    // These Overrides alter the behavior of superclass view iterator() methods
    Iterator<K> newKeyIterator()   { return new KeyIterator();   }
    Iterator<V> newValueIterator() { return new ValueIterator(); }
    Iterator<Map.Entry<K,V>> newEntryIterator() { return new EntryIterator(); }

    /**
     * 增加结点
     */
    void addEntry(int hash, K key, V value, int bucketIndex) {
        createEntry(hash, key, value, bucketIndex);

        // Remove eldest entry if instructed, else grow capacity if appropriate
        Entry<K,V> eldest = header.after;
        if (removeEldestEntry(eldest)) {
            removeEntryForKey(eldest.key);
        } else {
            if (size >= threshold)
                resize(2 * table.length);
        }
    }

    /**
     * 创建结点
     */
    void createEntry(int hash, K key, V value, int bucketIndex) {
        HashMap.Entry<K,V> old = table[bucketIndex];
 Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
        table[bucketIndex] = e;
        e.addBefore(header);
        size++;
    }

    /**
     * 判断是否需要删除最老元素
     */
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return false;
    }
}
```

LinkedHashMap 是继承自HashMap 所以内容 比较少 相比于它的父类 它的个性就是底层结点的数据结构采用带头结点的双向链表 而它的父类是采用单向链表 这样就导致它有一个非常牛B的特性，支持两种非常实用的排序：
1、FIFO 先进先出 2、LRU 最近最少使用排序
通过参数accessOrder来进行配置 为true时 表示按LRU排序 为false时表示按FIFO排序 默认按FIFO
如何用LinkedHashMap实现自己的缓存控制 ，下篇文章分解。