芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-12-10
JDK

精尽 JDK 源码解析 —— 集合（四）哈希表 LinkedHashMap

1. 概述

众所周知，HashMap 提供的访问，是无序的。而在一些业务场景下，我们希望能够提供有序访问的 HashMap 。那么此时，我们就有两种选择：

> TreeMap ：按照 key 的顺序。
> LinkedHashMap ：按照 key 的插入和访问的顺序。

LinkedHashMap ，在 HashMap 的基础之上，提供了顺序访问的特性。而这里的顺序，包括两种：

> 按照 key-value 的插入顺序进行访问。关于这一点，相信大多数人都知道。
>
> > 芳芳：如果不知道，那就赶紧知道。这不找抽么，哈哈哈。
>
> 按照 key-value 的访问顺序进行访问。通过这个特性，我们实现基于 LRU 算法的缓存。相信这一点，可能还是有部分胖友不知道的噢，下文我们也会提供一个示例。
>
> > 芳芳：面试中，有些面试官会喜欢问你，如何实现一个 LRU 的缓存。

实际上，LinkedHashMap 可以理解成是 LinkedList + HashMap 的组合。为什么这么说呢？让我们带着这样的疑问，一起往下看。

2. 类图

LinkedHashMap 实现的接口、继承的类，如下图所示：

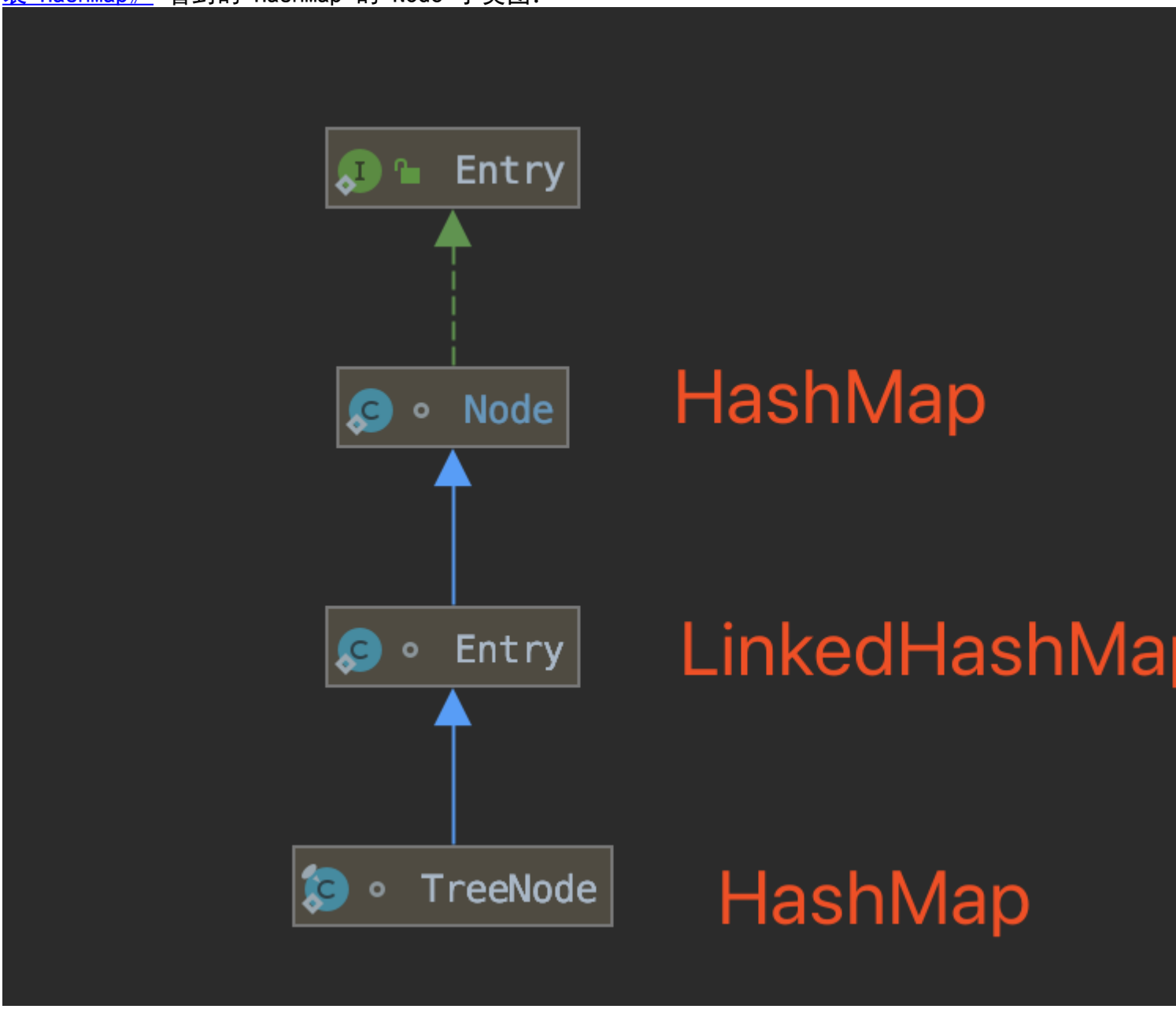
实现 Map 接口。
继承 HashMap 类。

很简单，很粗暴。嘿嘿~

芳芳：因为 LinkedHashMap 继承自 HashMap 类，所以它的代码并不多，不到 500 行。

3. 属性

在开始看 LinkedHashMap 的属性之前，我们先来看在 《精尽 JDK 源码解析 —— 集合（三）哈希表 HashMap》 看到的 HashMap 的 Node 子类图：



在图中，我们可以看到 LinkedHashMap 实现了自定义的节点 Entry ，一个支持指向前后节点的 Node 子类。代码如下：

```
// LinkedHashMap.java
```

```
static class Entry<K,V> extends HashMap.Node<K,V> {

    Entry<K,V> before, // 前一个节点
            after; // 后一个节点

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

}
```

before 属性，指向前一个节点。after 属性，指向后一个节点。
通过 before + after 属性，我们就可以形成一个以 Entry 为节点的链表。 胖友，发挥下你的想象力。

既然 LinkedHashMap 是 LinkedList + HashMap 的组合，那么必然就会有头尾节点两兄弟。所以属性如下：

```
// LinkedHashMap.java

/**
 * 头节点。
 *
 * 越老的节点，放在越前面。所以头节点，指向链表的开头
 *
 * The head (eldest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> head;

/**
 * 尾节点
 *
 * 越新的节点，放在越后面。所以尾节点，指向链表的结尾
 *
 * The tail (youngest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> tail;

/**
 * 是否按照访问的顺序。
 *
 * true ：按照 key-value 的访问顺序进行访问。
 * false ：按照 key-value 的插入顺序进行访问。
 *
 * The iteration ordering method for this linked hash map: {@code true}
 * for access-order, {@code false} for insertion-order.
 *
 * @serial
 */
final boolean accessOrder;
```

仔细看下每个属性的注释。
head + tail 属性，形成 LinkedHashMap 的双向链表。而访问的顺序，就是 head => tail 的过程。
accessOrder 属性，决定了 LinkedHashMap 的顺序。也就是说：
- true 时，当 Entry 节点被访问时，放置到链表的结尾，被 tail 指向。

- false 时，当 Entry 节点被添加时，放置到链表的结尾，被 tail 指向。如果插入的 key 对应的 Entry 节点已经存在，也会被放到结尾。

总结来说，就是如下一张图：

FROM 《Working of LinkedHashMap》

4. 构造方法

LinkedHashMap 一共有 5 个构造方法，其中四个和 HashMap 相同，只是多初始化 accessOrder = false 。所以，默认使用插入顺序进行访问。

另外一个 #LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) 构造方法，允许自定义 accessOrder 属性。代码如下：

```
// LinkedHashMap.java

public LinkedHashMap(int initialCapacity,
                     float loadFactor,
                     boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

5. 创建节点

在插入 key-value 键值对时，例如说 #put(K key, V value) 方法，如果不存在对应的节点，则会调用 #newNode(int hash, K key, V value, Node<K,V> e) 方法，创建节点。

因为 LinkedHashMap 自定义了 Entry 节点，所以必然需要重写该方法。代码如下：

```
// LinkedHashMap.java

Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {
    // <1> 创建 Entry 节点
    LinkedHashMap.Entry<K,V> p =
        new LinkedHashMap.Entry<>(hash, key, value, e);
    // <2> 添加到结尾
    linkNodeLast(p);
    // 返回
    return p;
}
```

<1> 处，创建 Entry 节点。虽然此处传入 e 作为 Entry.next 属性，指向下一个节点。但是实际上，#put(K key, V value) 方法中，传入的 e = null 。

<2> 处，调用 #linkNodeLast(LinkedHashMap.Entry<K,V> p) 方法，添加到结尾。代码如下：

```
// LinkedHashMap.java
```

```
private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {
    // 记录原尾节点到 last 中
    LinkedHashMap.Entry<K,V> last = tail;
    // 设置 tail 指向 p ，变更新的尾节点
    tail = p;
    // 如果原尾节点 last 为空，说明 head 也为空，所以 head 也指向 p
    if (last == null)
        head = p;
    // last <=> p ，相互指向
    else {
        p.before = last;
        last.after = p;
    }
}
```

- 这样，符合越新的节点，放到链表的越后面。

6. 节点操作回调

在 HashMap 的读取、添加、删除时，分别提供了 `#afterNodeAccess(Node<K,V> e)`、`#afterNodeInsertion(boolean evict)`、`#afterNodeRemoval(Node<K,V> e)` 回调方法。这样，LinkedHashMap 可以通过它们实现自定义拓展逻辑。

6.1 afterNodeAccess

在 `accessOrder` 属性为 `true` 时，当 Entry 节点被访问时，放置到链表的结尾，被 `tail` 指向。所以 `#afterNodeAccess(Node<K,V> e)` 方法的代码如下：

```
// LinkedHashMap.java

void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    // accessOrder 判断必须是满足按访问顺序。
    // (last = tail) != e 将 tail 赋值给 last ，并且判断是否 e 已经是队尾。如果是队尾，就不用处理了。
    if (accessOrder && (last = tail) != e) {
        // 将 e 赋值给 p 【因为要 Node 类型转换成 Entry 类型】
        // 同时 b、a 分别是 e 的前后节点
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        // 第一步，将 p 从链表中移除
        p.after = null;
        // 处理 b 的下一个节点指向 a
        if (b == null)
            head = a;
        else
            b.after = a;
        // 处理 a 的前一个节点指向 b
        if (a != null)
            a.before = b;
        else
            last = b;
        // 第二步，将 p 添加到链表的尾巴。实际这里的代码，和 linkNodeLast 是一致的。
        if (last == null)
            head = p;
```

```
            else {
                p.before = last;
                last.after = p;
            }
            // tail 指向 p ，实际就是 e 。
            tail = p;
            // 增加修改次数
            ++modCount;
        }
    }
```

代码已经添加详细的注释，胖友认真看看噢。
链表的操作看起来比较繁琐，实际一共分成两步：1）第一步，将 p 从链表中移除；2）将 p 添加到链表的尾巴。

因为 HashMap 提供的 #get(Object key) 和 #getOrDefault(Object key, V defaultValue) 方法，并未调用 #afterNodeAccess(Node<K,V> e) 方法，这在按照读取顺序访问显然不行，所以 LinkedHashMap 重写这两方法的代码，如下：

```
// LinkedHashMap.java

public V get(Object key) {
    // 获得 key 对应的 Node
    Node<K,V> e;
    if ((e = getNode(hash(key), key)) == null)
        return null;
    // 如果访问到，回调节点被访问
    if (accessOrder)
        afterNodeAccess(e);
    return e.value;
}

public V getOrDefault(Object key, V defaultValue) {
    // 获得 key 对应的 Node
    Node<K,V> e;
   if ((e = getNode(hash(key), key)) == null)
       return defaultValue;
    // 如果访问到，回调节点被访问
    if (accessOrder)
       afterNodeAccess(e);
   return e.value;
}
```

6.2 afterNodeInsertion

在开始看 #afterNodeInsertion(boolean evict) 方法之前，我们先来看看如何基于 LinkedHashMap 实现 LRU 算法的缓存。代码如下：

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

    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
```

```
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 LinkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map 中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }

}
```

为什么能够这么实现呢？我们在 `#afterNodeInsertion(boolean evict)` 方法中来理解。代码如下：

```
// LinkedHashMap.java

// evict 翻译为驱逐，表示是否允许移除元素
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    // first = head 记录当前头节点。因为移除从头开始，最老
    // <1> removeEldestEntry(first) 判断是否满足移除最老节点
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        // <2> 移除指定节点
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}
```

<1> 处，调用 `#removeEldestEntry(Map.Entry<K,V> eldest)` 方法，判断是否移除最老节点。代码如下：

```
// LinkedHashMap.java

protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return false;
}
```

- 默认情况下，都不移除最老的节点。所以在上述的 LRU 缓存的示例，重写了该方法，判断 LinkedHashMap 是否超过缓存最大大小。如果是，则移除最老的节点。

<2> 处，如果满足条件，则调用 `#removeNode(...)` 方法，移除最老的节点。

这样，是不是很容易理解基于 LinkedHashMap 实现 LRU 算法的缓存。

6.3 afterNodeRemoval

在节点被移除时，LinkedHashMap 需要将节点也从链表中移除，所以重写 `#afterNodeRemoval(Node<K,V> e)` 方法，实现该逻辑。代码如下：

```
// LinkedHashMap.java

void afterNodeRemoval(Node<K,V> e) { // unlink
    // 将 e 赋值给 p 【因为要 Node 类型转换成 Entry 类型】
    // 同时 b、a 分别是 e 的前后节点
    LinkedHashMap.Entry<K,V> p =
        (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
    // 将 p 从链表中移除
    p.before = p.after = null;
    // 处理 b 的下一个节点指向 a
    if (b == null)
        head = a;
    else
        b.after = a;
    // 处理 a 的前一个节点指向 b
    if (a == null)
        tail = b;
    else
        a.before = b;
}
```

7. 转换成数组

因为 LinkedHashMap 需要满足按顺序访问，所以需要重写 HashMap 提供的好多方法，例如说本小节我们看到的几个。

#keysToArray(T[] a) 方法，转换出 key 数组顺序返回。代码如下：

```
// LinkedHashMap.java

@Override
final <T> T[] keysToArray(T[] a) {
    Object[] r = a;
    int idx = 0;
    // 通过 head 顺序遍历，从头到尾
    for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after) {
        r[idx++] = e.key;
    }
    return a;
}
```

要小心噢，必须保证 a 放得下 LinkedHashMap 所有的元素。

#valuesToArray(T[] a) 方法，转换出 value 数组顺序返回。代码如下：

```
// LinkedHashMap.java

@Override
final <T> T[] valuesToArray(T[] a) {
    Object[] r = a;
    int idx = 0;
    // 通过 head 顺序遍历，从头到尾
    for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after) {
```

```
            r[idx++] = e.value;
        }
        return a;
    }
```

芮芮：看到此处，胖友基本可以结束本文落。

8. 转换成 Set/Collection

#keySet() 方法，获得 key Set 。代码如下：

```
// LinkedHashMap.java

public Set<K> keySet() {
    // 获得 keySet 缓存
    Set<K> ks = keySet;
    // 如果不存在，则进行创建
    if (ks == null) {
        ks = new LinkedKeySet(); // LinkedKeySet 是 LinkedHashMap 自定义的
        keySet = ks;
    }
    return ks;
}
```

其中， LinkedKeySet 是 LinkedHashMap 自定义的 Set 实现类。代码如下：

```
// LinkedHashMap.java

final class LinkedKeySet extends AbstractSet<K> {

    public final int size()                 { return size; }
    public final void clear()               { LinkedHashMap.this.clear(); }
    public final Iterator<K> iterator() {
        return new LinkedKeyIterator(); // <K>
    }
    public final boolean contains(Object o) { return containsKey(o); }
    public final boolean remove(Object key) {
        return removeNode(hash(key), key, null, false, true) != null;
    }
    public final Spliterator<K> spliterator()  {
        return Spliterators.spliterator(this, Spliterator.SIZED |
                                        Spliterator.ORDERED |
                                        Spliterator.DISTINCT);
    }

    public Object[] toArray() {
        return keysToArray(new Object[size]);
    }

    public <T> T[] toArray(T[] a) {
        return keysToArray(prepareArray(a));
    }

    public final void forEach(Consumer<? super K> action) {
```

```
                if (action == null)
                    throw new NullPointerException();
                int mc = modCount;
                for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after)
                    action.accept(e.key);
                if (modCount != mc)
                    throw new ConcurrentModificationException();
            }
        }
```

- 其内部，调用的都是 LinkedHashMap 提供的方法。

#values() 方法，获得 value Collection 。代码如下：

```
// LinkedHashMap.java

public Collection<V> values() {
    // 获得 values 缓存
    Collection<V> vs = values;
    // 如果不存在，则进行创建
    if (vs == null) {
        vs = new LinkedValues(); // LinkedValues 是 LinkedHashMap 自定义的
        values = vs;
    }
    return vs;
}
```

其中， LinkedValues 是 LinkedHashMap 自定义的 Collection 实现类。代码如下：

```
// LinkedHashMap.java

final class LinkedValues extends AbstractCollection<V> {
    public final int size()                 { return size; }
    public final void clear()               { LinkedHashMap.this.clear(); }
    public final Iterator<V> iterator() {
        return new LinkedValueIterator(); // <X>
    }
    public final boolean contains(Object o) { return containsValue(o); }
    public final Spliterator<V> spliterator() {
        return Spliterators.spliterator(this, Spliterator.SIZED |
                                        Spliterator.ORDERED);
    }

    public Object[] toArray() {
        return valuesToArray(new Object[size]);
    }

    public <T> T[] toArray(T[] a) {
        return valuesToArray(prepareArray(a));
    }

    public final void forEach(Consumer<? super V> action) {
        if (action == null)
            throw new NullPointerException();
        int mc = modCount;
        for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after)
```

```
                action.accept(e.value);
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
```

- 其内部，调用的都是 LinkedHashMap 提供的方法。

#entrySet() 方法，获得 key-value Set 。代码如下：

```
// LinkedHashMap.java

public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    // LinkedEntrySet 是 LinkedHashMap 自定义的
    return (es = entrySet) == null ? (entrySet = new LinkedEntrySet()) : es;
}
```

其中， LinkedEntrySet 是 LinkedHashMap 自定义的 Set 实现类。代码如下：

```
// LinkedHashMap.java

final class LinkedEntrySet extends AbstractSet<Map.Entry<K,V>> {
    public final int size()                 { return size; }
    public final void clear()               { LinkedHashMap.this.clear(); }
    public final Iterator<Map.Entry<K,V>> iterator() {
        return new LinkedEntryIterator(); // <X>
    }
    public final boolean contains(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<?,?> e = (Map.Entry<?,?>) o;
        Object key = e.getKey();
        Node<K,V> candidate = getNode(hash(key), key);
        return candidate != null && candidate.equals(e);
    }
    public final boolean remove(Object o) {
        if (o instanceof Map.Entry) {
            Map.Entry<?,?> e = (Map.Entry<?,?>) o;
            Object key = e.getKey();
            Object value = e.getValue();
            return removeNode(hash(key), key, value, true, true) != null;
        }
        return false;
    }
    public final Spliterator<Map.Entry<K,V>> spliterator() {
        return Spliterators.spliterator(this, Spliterator.SIZED |
                                        Spliterator.ORDERED |
                                        Spliterator.DISTINCT);
    }
    public final void forEach(Consumer<? super Map.Entry<K,V>> action) {
        if (action == null)
            throw new NullPointerException();
        int mc = modCount;
        for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after)
            action.accept(e);
```

```
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
```

- 其内部，调用的都是 LinkedHashMap 提供的方法。

在上面的代码中，艿艿实际标记了三处 <X> 标记，分别是 LinkedKeyIterator、LinkedValueIterator、LinkedEntryIterator ，用于迭代遍历 key、value、Entry 。而它们都继承了 LinkedHashIterator 抽象类，代码如下：

```
// LinkedHashMap.java

abstract class LinkedHashIterator {

    /**
     * 下一个节点
     */
    LinkedHashMap.Entry<K,V> next;
    /**
     * 当前节点
     */
    LinkedHashMap.Entry<K,V> current;
    /**
     * 修改次数
     */
    int expectedModCount;

    LinkedHashIterator() {
        next = head;
        expectedModCount = modCount;
        current = null;
    }

    public final boolean hasNext() {
        return next != null;
    }

    final LinkedHashMap.Entry<K,V> nextNode() {
        LinkedHashMap.Entry<K,V> e = next;
        // 如果发生了修改，抛出 ConcurrentModificationException 异常
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        // 如果 e 为空，说明没有下一个节点，则抛出 NoSuchElementException 异常
        if (e == null)
            throw new NoSuchElementException();
        // 遍历到下一个节点
        current = e;
        next = e.after;
        return e;
    }

    public final void remove() {
        // 移除当前节点
        Node<K,V> p = current;
        if (p == null)
            throw new IllegalStateException();
        // 如果发生了修改，抛出 ConcurrentModificationException 异常
```

```
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // 标记 current 为空，因为被移除了
            current = null;
            // 移除节点
            removeNode(p.hash, p.key, null, false, false);
            // 修改 expectedModCount 次数
            expectedModCount = modCount;
        }

    }

    final class LinkedKeyIterator extends LinkedHashIterator
        implements Iterator<K> {

        // key
        public final K next() { return nextNode().getKey(); }
    }

    final class LinkedValueIterator extends LinkedHashIterator
        implements Iterator<V> {

        // value
        public final V next() { return nextNode().value; }
    }

    final class LinkedEntryIterator extends LinkedHashIterator
        implements Iterator<Map.Entry<K,V>> {

        // Entry
        public final Map.Entry<K,V> next() { return nextNode(); }
    }
```

9. 清空

#clear() 方法，清空 LinkedHashMap 。代码如下：

```
    // LinkedHashMap.java

    public void clear() {
        // 清空
        super.clear();
        // 标记 head 和 tail 为 null
        head = tail = null;
    }
```

需要额外清空 head、tail 。

10. 其它方法

本小节，我们会罗列下其他 LinkedHashMap 重写的方法。当然，可以选择不看。

在序列化时，会调用到 #internalWriteEntries(java.io.ObjectOutputStream s) 方法，重写代码如下：

```
// LinkedHashMap.java

void internalWriteEntries(java.io.ObjectOutputStream s) throws IOException {
    // 通过 head 顺序遍历，从头到尾
    for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after) {
        // 写入 key
        s.writeObject(e.key);
        // 写入 value
        s.writeObject(e.value);
    }
}
```

在反序列化时，会调用 `#reinitialize()` 方法，重写代码如下：

```
// LinkedHashMap.java

void reinitialize() {
    // 调用父方法，初始化
    super.reinitialize();
    // 标记 head 和 tail 为 null
    head = tail = null;
}
```

查找值时，会调用 `#containsValue(Object value)` 方法，重写代码如下：

```
// LinkedHashMap.java

public boolean containsValue(Object value) {
    // 通过 head 顺序遍历，从头到尾
    for (LinkedHashMap.Entry<K,V> e = head; e != null; e = e.after) {
        V v = e.value;
        // 判断是否相等
        if (v == value || (value != null && value.equals(v)))
            return true;
    }
    return false;
}
```

666. 彩蛋

如下几个方法，是 LinkedHashMap 重写和红黑树相关的几个方法，胖友可以自己瞅瞅：

```
#replacementNode(Node<K,V> p, Node<K,V> next)
#newTreeNode(int hash, K key, V value, Node<K,V> next)
#replacementTreeNode(Node<K,V> p, Node<K,V> next)
#transferLinks(LinkedHashMap.Entry<K,V> src, LinkedHashMap.Entry<K,V> dst)
```

下面，我们来对 LinkedHashMap 做一个简单的小结：

LinkedHashMap 是 HashMap 的子类，增加了顺序访问的特性。

- 【默认】当 `accessOrder = false` 时，按照 key-value 的插入顺序进行访问。
- 当 `accessOrder = true` 时，按照 key-value 的读取顺序进行访问。

LinkedHashMap 的顺序特性，通过内部的双向链表实现，所以我们把它看成是 LinkedList + LinkedHashMap 的组合。
LinkedHashMap 通过重写 HashMap 提供的回调方法，从而实现其对顺序的特性的处理。同时，因为 LinkedHashMap 的顺序特性，需要重写 #keysToArray(T[] a) 等遍历相关的方法。
LinkedHashMap 可以方便实现 LRU 算法的缓存，

文章目录

回到首页