HashMap、LinkedHashMap、TreeMap、Hashtable、HashSet和ConcurrentHashMap区别

# HashMap

HashMap是基于哈希表的Map接口的非同步实现，

允许使用null值和null键(HashMap最多只允许一条记录的键为null，允许多条记录的值为null。)。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap中不允许出现重复的键（Key）

Hashmap是非线程安全的，

其迭代器是fail-fast的

HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体，（JDK1.8增加了红黑树部分,会将时间复杂度从O(n)降为O(logn)）。

数据存储：先根据key的hashCode（使用key的hashCode()方法获取）重新计算hash值，根据hash值算出这个元素在数组中的位置（即下标）， 如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

数据读取：首先根据key的hashCode，找到其数组中对应位置的数据(可能只有一个数据，也可能是多个数据，其表现形式是一个链表)，然后通过key的equals方法在对应位置的链表中找到需要的元素。

hashMap的默认初始容量是16个，其会有一个负载因子，用于当hashMap中的数据量等于容量\*负载因子时，hashMap会进行扩容，扩大的容量是原本的2倍。负载因子的默认初始值为0.75

HashSet

它是基于HashMap实现的，底层采用HashMap来保存元素，而且只使用了HashMap的key来实现各种特性。HashSet实现了Set接口

HashSet较HashMap来说比较慢

HashSet中的数据不是key-value键值对，其只是单值，虽然其借助与HashMap来实现，但是其只是将值作为key来存入HashMap中，因为HashMap中的值是key-value键值对的，所以每个HashSet存储到HashMap的数据对应的value值只是一个new Object()对象

当添加数据时，如果set中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))的元素e2，则向此set 添加指定的元素e。如果此set已包含该元素，则该调用不更改set并返回false。但底层实际将将该元素作为key放入HashMap。

Hashtable

Hashtable也是一个散列表，它存储的内容是键值对。基于Dictionary类

存储数据: 首先判断value是否为空，为空则抛出异常；计算key的hash值，并根据hash值获得key在table数组中的位置index，如果table[index]元素不为空，则进行迭代，如果遇到相同的key，则直接替换，并返回旧value；否则，我们可以将其插入到table[index]位置。

key和value都不允许为null，Hashtable遇到null，直接返回NullPointerException。

线程安全,几乎所有的public的方法都是synchronized的

较HashMap速度慢

LinkedHashMap

LinkedHashMap是HashMap的一个子类，它保留插入顺序，帮助我们实现了有序的HashMap。

其维护一个双向链表，并不是说其除了维护存入的数据，另外维护了一个双向链表对象，而是说其根据重写HashMap的实体类Entry，来实现能够将HashMap的数据组成一个双向列表，其存储的结构还是数组+链表的形式，

LinkedHashMap能够做到按照插入顺序或者访问顺序进行迭代顺序。

修改Entry对象，Entry新增了其上一个元素before和下一个元素after的引用

根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用get方法)的链表。默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表

LinkedHashMap并未重写父类HashMap的put方法，只是重写了put方法里面的recordAccess、addEntry、createEntry等方法，添加了特有的双向链接列表

LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

这个虽然是访问元素，但是当设置以访问排序时，其仍然会先将元素从原本位置remove掉，然后在将该元素以新元素插入到链头，哪怕其新插入的位置还在原位置(所以如果以访问排序，其过程会涉及到删除数据和增添数据)。

读取速度与容量无关

TreeMap

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。 该映射根据其键的自然顺序(字母排序)进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

TreeMap是非线程安全的。 它的iterator 方法返回的迭代器是fail-fast的。

ConcurrentHashMap

ConcurrentHashMap是弱一致性，也就是说遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据

ConcurrentHashMap是基于分段锁设计来实现线程安全性，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。

并发度就是ConcurrentHashMap中的分段锁个数，默认的并发度为16，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）

通过将key的高n位（n = 32 – segmentShift）和并发度减1（segmentMask）做位与运算定位到所在的Segment(分段)

HashMap

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键(HashMap最多只允许一条记录的键为null，允许多条记录的值为null。)。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap中不允许出现重复的键（Key）

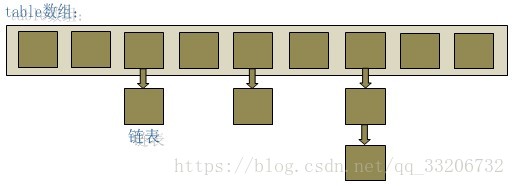
Hashmap是非线程安全的，如果多个线程同时访问一个HashMap，可能会导致数据不一致，所以当其中至少一个线程从结构上（指添加或者删除一个或多个映射关系的任何操作）修改了，则必须保持外部同步，以防止对映射进行意外的非同步访问。

其迭代器是fail-fast的

数据结构

HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体，（JDK1.8增加了红黑树部分,会将时间复杂度从O(n)降为O(logn)）。

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个Entry数组，其大小为capacity。如下图：

Entry就是数组中的元素，每个Entry其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

1.static class Entry<K,V> implements Map.Entry<K,V> {

1. final K key;
2. V value;
3. Entry<K,V> next;
4. final int hash;
5. ……
6. }

数据存储

当我们往HashMap中put元素的时候，先根据key的hashCode（使用key的hashCode()方法获取）重新计算hash值(具体的算法这里就不讲解了，hash的算法中包含了很多优化的点，是存储的数据能够数组的每一位尽量只有一个值，而不是一个链表，这样能够提高查询效率)，根据hash值算出这个元素在数组中的位置（即下标）， 如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

数据读取

hashMap的数据读取相对简单，首先根据key的hashCode，找到其数组中对应位置的数据(可能只有一个数据，也可能是多个数据，其表现形式是一个链表)，然后通过key的equals方法在对应位置的链表中找到需要的元素。

扩容

hashMap的默认初始容量是16个，其会有一个负载因子，用于当hashMap中的数据量等于容量\*负载因子时，hashMap会进行扩容，扩大的容量是原本的2倍。负载因子的默认初始值为0.75，这个值是经过折中的取值，其也是合理的，所以如非特殊需求，不建议修改该因子，

对于初始容量和负载因子的设置，我们可以在创建HashMap对象时指定(也可以大于1)，HashMap提供了三个构造方法用于我们使用：

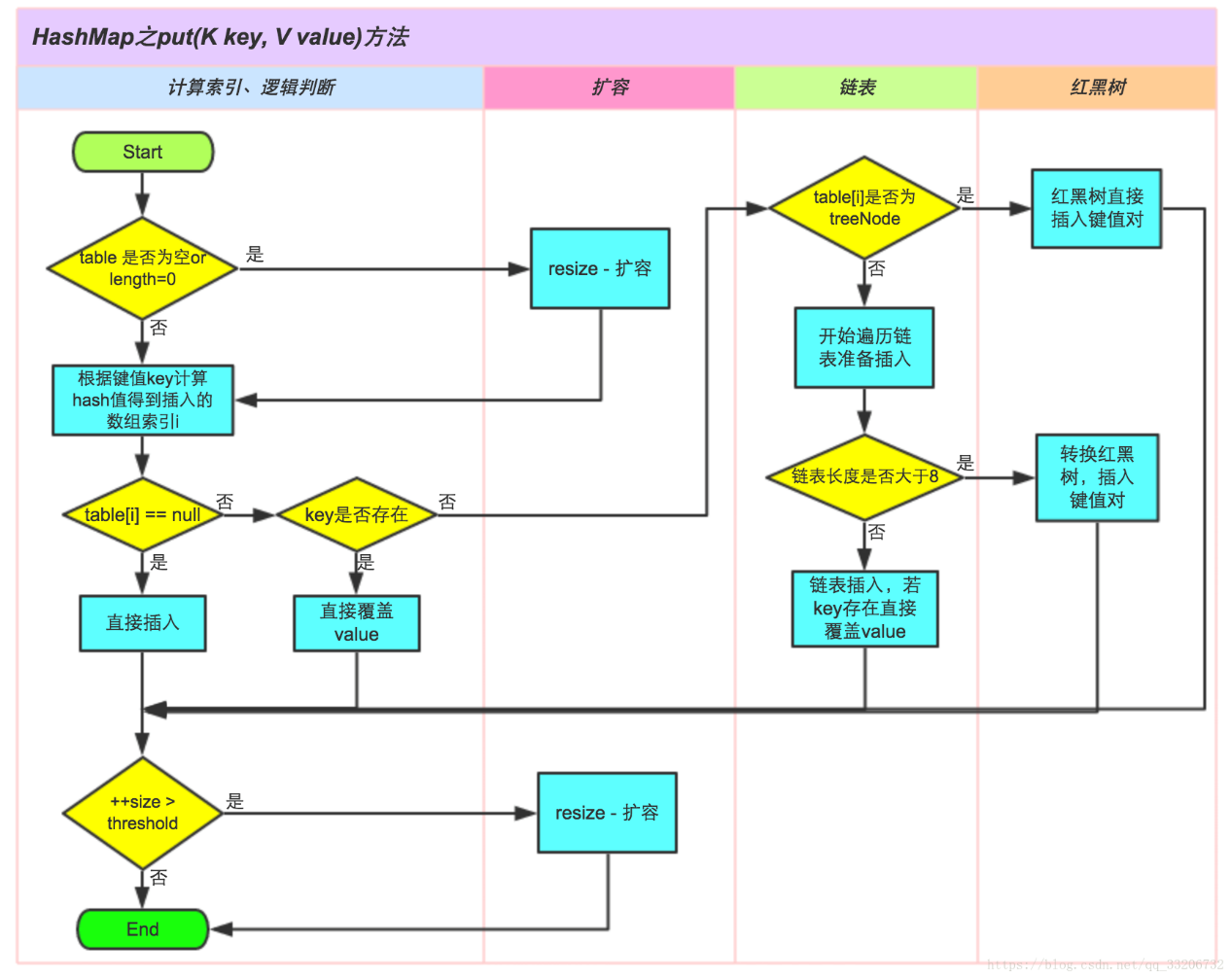
HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。  
HashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。  
HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

因为每次扩容都会对原本的数据进行重新的hash计算其在新数组中的位置，所以扩容是非常消耗性能的，所以为了尽量少的避免多次扩容，我们如果知道数据的大概量，我们可以在创建时进行指定初始容量大小。

红黑树  
即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，可以参考[这篇博文](https://blog.csdn.net/v_july_v/article/details/6105630)来详细认识红黑树

归纳

简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，  
也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

网上找到了一个HashMap的存储流程图  


HashSet

HashSet实现了Set接口，它不允许集合中出现重复元素。

对于HashSet而言，它是基于HashMap实现的，底层采用HashMap来保存元素，而且只使用了HashMap的key来实现各种特性。相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成，我们应该为保存到HashSet中的对象覆盖hashCode()和equals()，以保证放入的对象的唯一性。

HashSet较HashMap来说比较慢

其迭代器是fail-fast的

存储数据

HashSet中的数据不是key-value键值对，其只是单值，虽然其借助与HashMap来实现，但是其只是将值作为key来存入HashMap中，因为HashMap中的值是key-value键值对的，所以每个HashSet存储到HashMap的数据对应的value值只是一个new Object()对象，

当添加数据时，如果set中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))的元素e2，则向此set 添加指定的元素e。如果此set已包含该元素，则该调用不更改set并返回false。但底层实际将将该元素作为key放入HashMap。

这是因为HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），新添加的Entry的value会将覆盖原来Entry的value（HashSet中的value都是PRESENT），但key不会有任何改变，因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性

| HashMap | HashSet |
| --- | --- |
| HashMap实现了Map接口 | HashSet实现了Set接口 |
| HashMap储存键值对 | HashSet仅仅存储对象 |
| 使用put()方法将元素放入map中 | 使用add()方法将元素放入set中 |
| HashMap中使用键对象来计算hashcode值 | HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false |
| HashMap比较快，因为是使用唯一的键来获取对象 | HashSet较HashMap慢 |

Hashtable

和HashMap一样，Hashtable也是一个散列表，它存储的内容是键值对。

成员变量

Hashtable是通过”拉链法”实现的哈希表。它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount。

table：是一个Entry[]数组类型，而Entry实际上就是一个单向链表。哈希表的”key-value键值对”都是存储在Entry数组中的。  
count：是Hashtable的大小，它是Hashtable保存的键值对的数量。  
threshold：是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值=”容量\*加载因子”。  
loadFactor：就是加载因子。  
modCount：是用来实现fail-fast机制的(用于防止在读取过程中，有更新操作，如果有更新操，该参数会修改，然后读取操作会报错)。

构造方法  
Hashtable一共提供了4个构造方法：

public Hashtable(int initialCapacity, float loadFactor)： 用指定初始容量和指定加载因子构造一个新的空哈希表。useAltHashing为boolean，其如果为真，则执行另一散列的字符串键，以减少由于弱哈希计算导致的哈希冲突的发生。  
public Hashtable(int initialCapacity)：用指定初始容量和默认的加载因子 (0.75) 构造一个新的空哈希表。  
public Hashtable()：默认构造函数，容量为11，加载因子为0.75。  
public Hashtable(Map<? extends K, ? extends V> t)：构造一个与给定的 Map 具有相同映射关系的新哈希表。

存储数据

其存储数据流程如下：  
1.判断value是否为空，为空则抛出异常；  
2.计算key的hash值，并根据hash值获得key在table数组中的位置index，如果table[index]元素不为空，则进行迭代，如果遇到相同的key，则直接替换，并返回旧value；  
3.否则，我们可以将其插入到table[index]位置。

从上面可以看出，其存储流程和HashMap相似。

获取数据

相比较于put方法，get方法则简单很多。其过程就是首先通过hash()方法求得key的哈希值，然后根据hash值得到index索引（上述两步所用的算法与put方法都相同）。然后迭代链表，返回匹配的key的对应的value；找不到则返回null。

HashTable与HashMap比较

| HashTable | HashMap |
| --- | --- |
| 基于Dictionary类 | 基于AbstractMap类 |
| key和value都不允许为null，Hashtable遇到null，直接返回NullPointerException。 | key和value都允许为null，HashMap遇到key为null的时候，调用putForNullKey方法进行处理。 |
| 线程安全,几乎所有的public的方法都是synchronized的 | 非线程安全 |
| 速度慢 | 1. 速度快 |

LinkedHashMap

概述

我们知道HashMap是无序的存储，即使我们有序的将数据存储到HashMap中，但是我们读取出来的数据的顺序，很大可能与存储的顺序不同。

而LinkedHashMap是HashMap的一个子类，它保留插入顺序，帮助我们实现了有序的HashMap。LinkedHashMap维护着一个运行于所有条目的双重链接列表，在每次插入数据，或者访问、修改数据时，会增加节点、或调整链表的节点顺序,用于定义迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

注：其维护一个双向链表，并不是说其除了维护存入的数据，另外维护了一个双向链表对象，而是说其根据重写HashMap的实体类Entry，来实现能够将HashMap的数据组成一个双向列表，其存储的结构还是数组+链表的形式，看下面我自己理解的存储和遍历的流程图大家可以看下

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代,LinkedHashMap能够做到按照插入顺序或者访问顺序进行迭代顺序。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

和HashMap一样，其不保证线程安全。

成员变量

LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了数组中保存的元素Entry，该Entry除了保存当前对象的引用外，还新增了其上一个元素before和下一个元素after的引用，从而在哈希表的基础上又构成了双向链接列表。

初始化

在LinkedHashMap的构造方法中，实际调用了父类HashMap的相关构造方法来构造一个底层存放的table数组，但额外可以增加accessOrder这个参数，如果不设置，默认为false，代表按照插入顺序进行迭代；当然可以显式设置为true，代表以访问顺序进行迭代(根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用get方法)的链表。默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。)。

public LinkedHashMap(int initialCapacity, float loadFactor,boolean accessOrder) {

super(initialCapacity, loadFactor);

this.accessOrder = accessOrder;

}

我们已经知道LinkedHashMap的Entry元素继承HashMap的Entry，提供了双向链表的功能。在上述HashMap的构造器中，最后会调用init()方法，进行相关的初始化，这个方法在HashMap的实现中并无意义，只是提供给子类实现相关的初始化调用。

1. 但在LinkedHashMap重写了init()方法，在调用父类的构造方法完成构造后，进一步实现了对其元素Entry的初始化操作。

void init() {

header = new Entry<>(-1, null, null, null);

header.before = header.after = header;

}

数据存储

LinkedHashMap并未重写父类HashMap的put方法(所以其调用的方法逻辑还是HashMap的put方法的存储逻辑)，而是重写了父类HashMap的put方法调用的子方法void recordAccess(HashMap m) ，void addEntry(int hash, K key, V value, int bucketIndex) 和void createEntry(int hash, K key, V value, int bucketIndex)，提供了自己特有的双向链接列表的实现。  
HashMap.put:

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key);

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

重写方法：

void recordAccess(HashMap<K,V> m) {

LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;

if (lm.accessOrder) {

lm.modCount++;

remove();

addBefore(lm.header);

}

}

void addEntry(int hash, K key, V value, int bucketIndex) {

// 调用create方法，将新元素以双向链表的的形式加入到映射中。

createEntry(hash, key, value, bucketIndex);

// 删除最近最少使用元素的策略定义

Entry<K,V> eldest = header.after;

if (removeEldestEntry(eldest)) {

removeEntryForKey(eldest.key);

} else {

if (size >= threshold)

resize(2 \* table.length);

}

}

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;

// 调用元素的addBrefore方法，将元素加入到哈希、双向链接列表。

e.addBefore(header);

size++;

}

private void addBefore(Entry<K,V> existingEntry) {

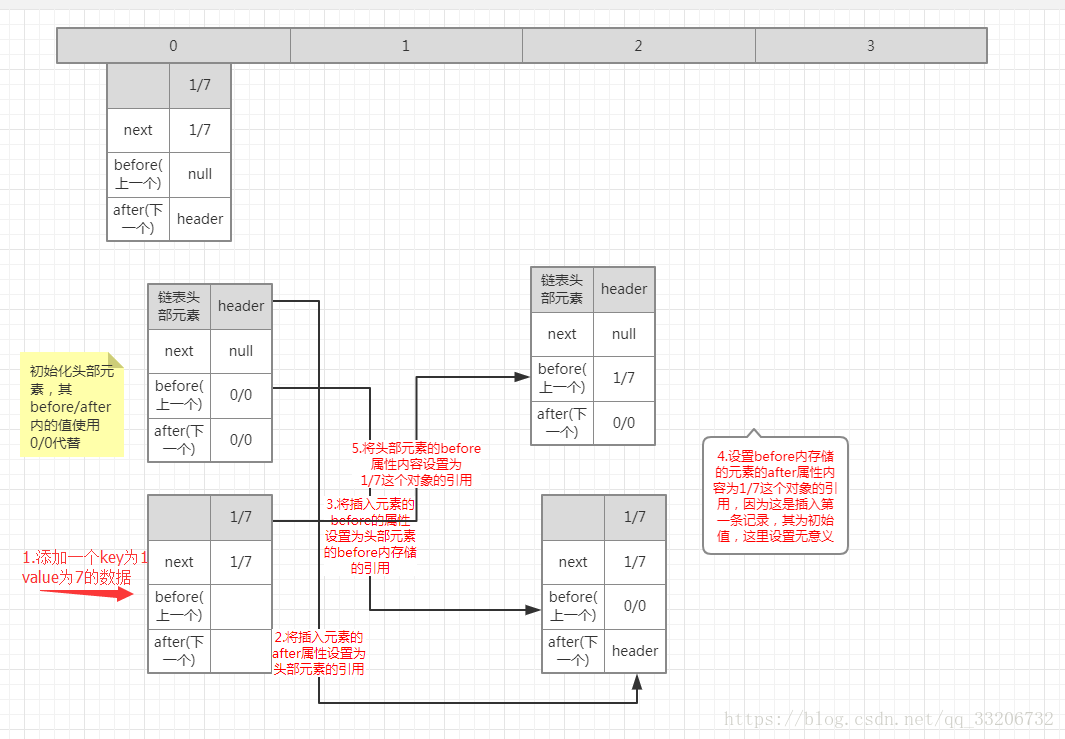
after = existingEntry;

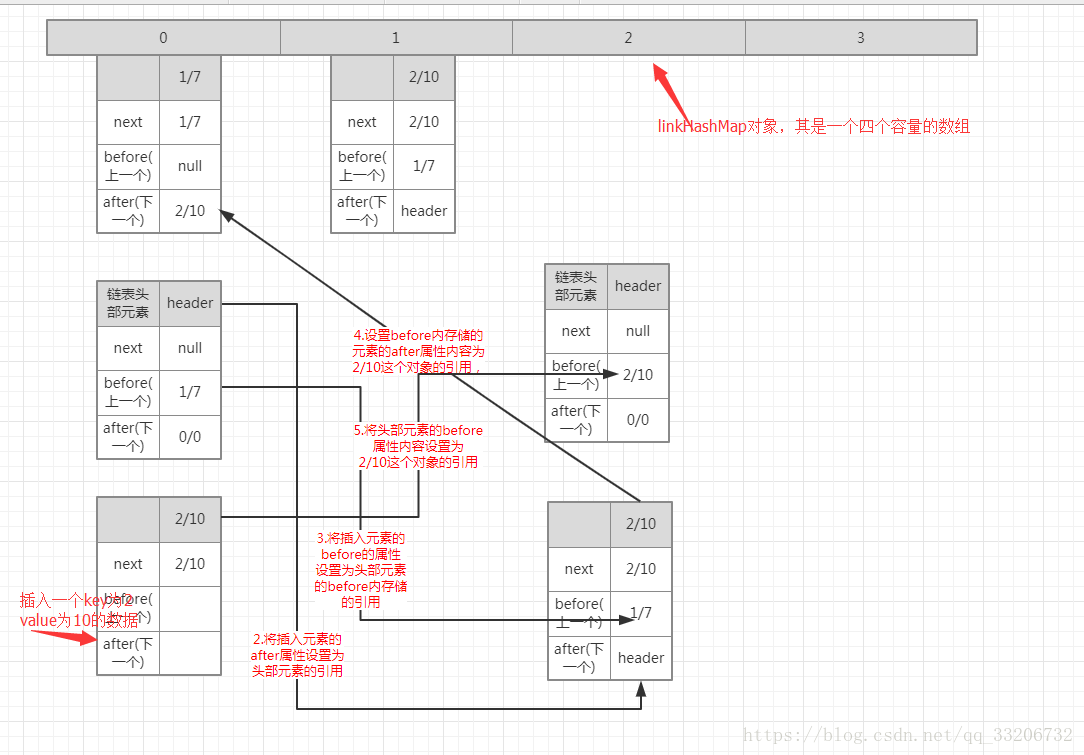
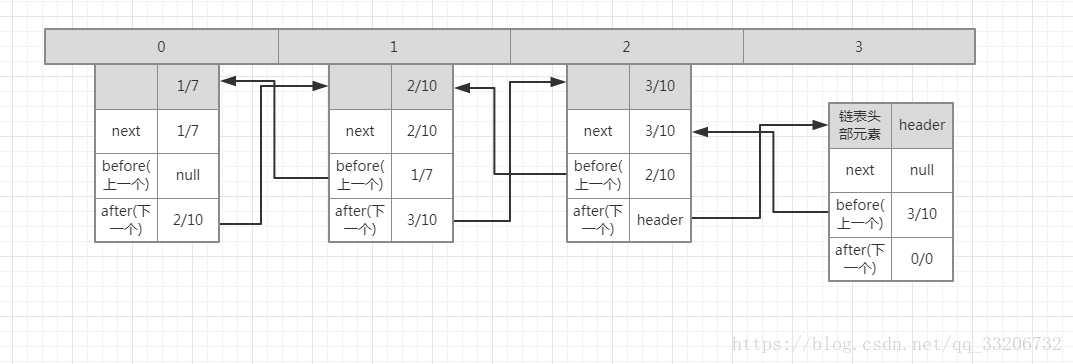
before = existingEntry.before;

before.after = this;

after.before = this;

}

对于数据的存储，其逻辑主要在上面的addBefore这个方法，其逻辑可以看下面三个图：  
1.添加一个key为1，value为7的数据。  


1. 2.在上面的基础上插入第二个数据，key为2 value为10的数据。  
     
   3.同理插入第三个数据之后，其形成的双向链表如图所示。  
   

数据读取

LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

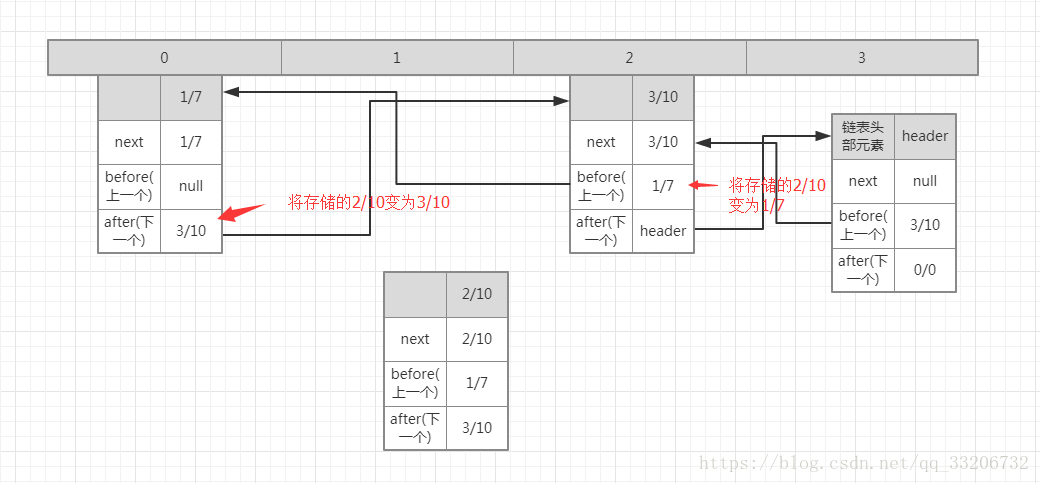
数据遍历  
从上面的流程图形成的双向链表，我们应该很容易就能有序遍历出来吧，这里就不多说了。  
说一下一种情况，就是删除一个数据，怎么维护其链表，或者以访问排序情况下，访问一个元素之后，会怎么维护链表。  
删除一个元素，维护链表主要在remove()方法的逻辑：

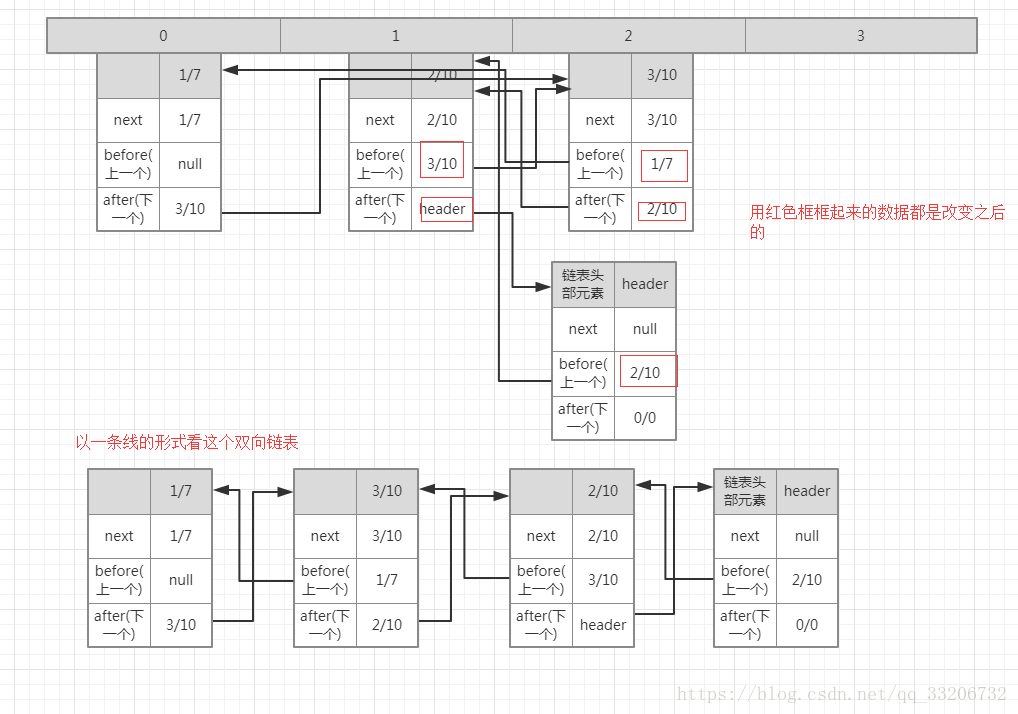
private void remove() {

before.after = after;//将删除元素的before属性存储的引用的after数据改成删除元素的after存储的引用，(本来应该是删除元素的引用)

after.before = before;//将删除元素的after属性存储的引用的before数据改成删除元素的before存储的引用，

}

如下图  


如果以访问排序情况下，访问一个元素之后，会怎么维护链表。  
这个虽然是访问元素，但是当设置以访问排序时，其仍然会先将元素从原本位置remove掉，然后在将该元素以新元素插入到链头，哪怕其新插入的位置还在原位置(所以如果以访问排序，其过程会涉及到删除数据和增添数据)。不过如果默认排序，则不会有此操作。如下图，是访问之后的双链表。  


LinkedHashMap和HashMap比较

| HashMap | LinkedHashMap |
| --- | --- |
| 无序存储 | 有序存储,以双向链表实现 |
| 读取速度与容量有关 | 读取速度与容量无关 |
| 线程不安全 | 1. 线程不安全 |
| key-value都允许null | 1. key-value都允许null |

TreeMap

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。  
TreeMap 继承于AbstractMap，所以它是一个Map，即一个key-value集合。  
TreeMap 实现了NavigableMap接口，意味着它支持一系列的导航方法。比如返回有序的key集合。  
TreeMap 实现了Cloneable接口，意味着它能被克隆。

TreeMap基于红黑树（Red-Black tree）实现。该映射根据其键的自然顺序(字母排序)进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。  
TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。  
TreeMap是非线程安全的。 它的iterator 方法返回的迭代器是fail-fast的。

这里就不详细介绍TreeMap了，我直接写其与HashMap的区别

| HashMap | TreeMap |
| --- | --- |
| 遍历出来数据无序 | 自然排序或者创建映射提供的Comparator 进行排序 |
| 基于散列表 | 红黑树 |
| 取值速度快 | 取值速度慢 |
| 适用于在Map中插入、删除和定位元素 | 适用于按自然顺序或自定义顺序遍历键(key) |

ConcurrentHashMap

如果大家想找详细的说明，可以看[这篇博文](http://www.importnew.com/22007.html)

我们知道HashMap是非线程安全的，我们想要线程安全的集合只能使用使用锁的HashTable或者Collections.synchronizedMap(hashMap)，但是由于其操作都是使用锁进行锁定操作的，其性能较差，所以ConcurrentHashMap就是为了解决HashMap的非线程安全性的。

ConcurrentHashMap是弱一致性，也就是说遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据

ConcurrentHashMap究其根本，其还是一个HashMap的实现，不过是在HashMap的方法上面做了一些处理。

设计思路

ConcurrentHashMap采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。

1. ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

并发度

并发度可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap\*\*默认的并发度为16\*\*，但用户也可以在构造函数中设置并发度。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。运行时通过将key的高n位（n = 32 – segmentShift）和并发度减1（segmentMask）做位与运算定位到所在的Segment。segmentShift与segmentMask都是在构造过程中根据concurrency level被相应的计算出来。

如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。（文档的说法是根据你并发的线程数量决定，太多会导性能降低）

创建分段锁

和JDK6不同，JDK7中除了第一个Segment之外，剩余的Segments采用的是延迟初始化的机制：每次put之前都需要检查key对应的Segment是否为null，如果是则调用ensureSegment()以确保对应的Segment被创建。

ensureSegment可能在并发环境下被调用，但与想象中不同，ensureSegment并未使用锁来控制竞争，而是使用了Unsafe对象的getObjectVolatile()提供的原子读语义结合CAS来确保Segment创建的原子性。

存储数据

ConcurrentHashMap的put方法被代理到了对应的Segment中。与JDK6不同的是，JDK7版本的ConcurrentHashMap在获得Segment锁的过程中，做了一定的优化 - 在真正申请锁之前，put方法会通过tryLock()方法尝试获得锁，在尝试获得锁的过程中会对对应hashcode的链表进行遍历，如果遍历完毕仍然找不到与key相同的HashEntry节点，则为后续的put操作提前创建一个HashEntry。当tryLock一定次数后仍无法获得锁，则通过lock申请锁。

需要注意的是，由于在并发环境下，其他线程的put，rehash或者remove操作可能会导致链表头结点的变化，因此在过程中需要进行检查，如果头结点发生变化则重新对表进行遍历。而如果其他线程引起了链表中的某个节点被删除，即使该变化因为是非原子写操作可能导致当前线程无法观察到，但因为不影响遍历的正确性所以忽略不计。

之所以在获取锁的过程中对整个链表进行遍历，主要目的是希望遍历的链表被CPU cache所缓存，为后续实际put过程中的链表遍历操作提升性能。

在获得锁之后，Segment对链表进行遍历，如果某个HashEntry节点具有相同的key，则更新该HashEntry的value值，否则新建一个HashEntry节点，将它设置为链表的新head节点并将原头节点设为新head的下一个节点。新建过程中如果节点总数（含新建的HashEntry）超过threshold，则调用rehash()方法对Segment进行扩容，最后将新建HashEntry写入到数组中。

put方法中，链接新节点的下一个节点（HashEntry.setNext()）以及将链表写入到数组中（setEntryAt()）都是通过Unsafe的putOrderedObject()方法来实现，这里并未使用具有原子写语义的putObjectVolatile()的原因是：JMM会保证获得锁到释放锁之间所有对象的状态更新都会在锁被释放之后更新到主存，从而保证这些变更对其他线程是可见的。

删除数据

和put类似，remove在真正获得锁之前，也会对链表进行遍历以提高缓存命中率。

获取数据

get与containsKey两个方法几乎完全一致：他们都没有使用锁，而是通过Unsafe对象的getObjectVolatile()方法提供的原子读语义，来获得Segment以及对应的链表，然后对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。如果要求强一致性，那么必须使用Collections.synchronizedMap()方法。

Fail-Fast机制

我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

ail-fast 机制是java集合(Collection)中的一种错误机制。 当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。

例如：当某一个线程A通过 iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出 ConcurrentModificationException异常，产生 fail-fast 事件。

这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容（当然不仅仅是HashMap才会有，其他例如ArrayList也会）的修改都将增加这个值（大家可以再回头看一下其源码，在很多操作中都有modCount++这句），那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

HashIterator() {

expectedModCount = modCount;

if (size > 0) { // advance to first entry

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

}

在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

注意到modCount声明为volatile，保证线程之间修改的可见性。

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

在HashMap的API中指出：

由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

解决方案

在上文中也提到，fail-fast机制，是一种错误检测机制。它只能被用来检测错误，因为JDK并不保证fail-fast机制一定会发生。若在多线程环境下使用 fail-fast机制的集合，建议使用“java.util.concurrent包下的类”去取代“java.util包下的类”。