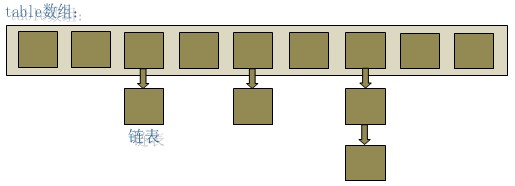
<http://wiki.jikexueyuan.com/project/java-collection/>

[HashMap的实现原理](http://wiki.jikexueyuan.com/project/java-collection/hashmap.html)

* 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



* 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。
* Indexing:

static int indexFor(int h, int length) {

return h & (length-1);

}

这个方法非常巧妙，它通过h & (table.length -1)来得到该对象的保存位，而HashMap底层数组的长度总是 2 的 n 次方，这是 HashMap 在速度上的优化。在HashMap构造器中有如下代码：

// Find a power of 2 >= initialCapacity

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

当length总是2的n次方时，h & (length-1)运算等价于对 length 取模，也就是h % length，但是&比%具有更高的效率

* HashMap包含如下几个构造器：
  + 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。
* Iterator Fail-Fast 机制

HashIterator() {

expectedModCount = modCount;

…

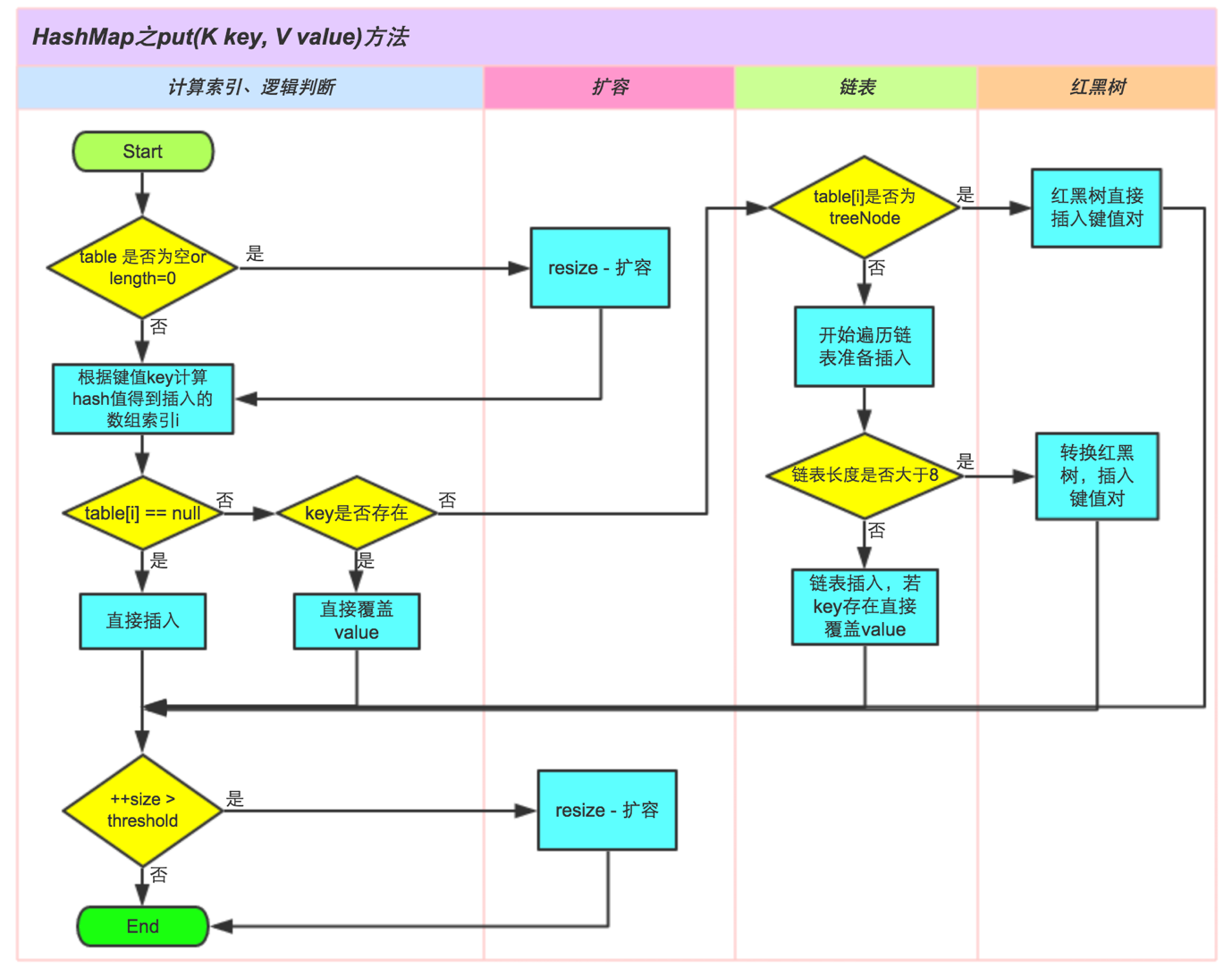
}

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

* java 1.8改进：链表长度大于8时转化为红黑树



[HashSet的实现原理](http://wiki.jikexueyuan.com/project/java-collection/hashset.html)

* 基于HashMap实现的，底层采用 HashMap来保存元素。

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

public boolean contains(Object o) {

return map.containsKey(o);

}

public boolean remove(Object o) {

return map.remove(o)==PRESENT;

}

[Hashtable的实现原理](http://wiki.jikexueyuan.com/project/java-collection/hashtable.html)

public class Hashtable<K,V>

extends Dictionary<K,V>

implements Map<K,V>, Cloneable, java.io.Serializable{}

* HashTable基于Dictionary类，而HashMap是基于AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。
* HashMap 的 key 和 value 都允许为 null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable遇到 null，直接返回 NullPointerException。
* Hashtable 方法是同步，而HashMap则不是。HashMap可以通过下面的语句进行同步：  
  Map m = Collections.synchronizeMap(hashMap);
* HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。

[LinkedHashMap 的实现原理](http://wiki.jikexueyuan.com/project/java-collection/linkedhashmap.html)

* LinkedHashMap是HashMap的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用LinkedHashMap。
* 定义了一个 Entry<K,V> header，这个header不是放在Table里，它是额外独立出来的。LinkedHashMap通过继承hashMap中的Entry<K,V>,并添加两个属性 Entry<K,V> before,after,和header结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。
* protected boolean removeEldestEntry([Map.Entry](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html" \o "interface in java.util)<[K](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html),[V](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html)> eldest)

Returns true if this map should remove its eldest entry.

[LinkedHashSet 的实现原理](http://wiki.jikexueyuan.com/project/java-collection/linkedhashset.html)

* 对于 LinkedHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。
* 在父类 HashSet 中，专为 LinkedHashSet 提供的构造方法如下，该方法为包访问权限，并未对外公开。

HashSet(int initialCapacity, float loadFactor, boolean dummy) {

map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);

}

[ArrayList 的实现原理](http://wiki.jikexueyuan.com/project/java-collection/arraylist.html)

* 数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍
* fail-fast

CopyOnWriteArrayList

* CopyOnWriteArrayList是ArrayList 的一个线程安全的变体，其中所有可变操作（add、set等等）都是通过对底层数组进行一次新的复制来实现的。

public boolean add(E e) {

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

Object[] newElements = Arrays.copyOf(elements, len + 1);

newElements[len] = e;

setArray(newElements);

return true;

} finally {

lock.unlock();

}

}

* 适合使用在读操作远远大于写操作的场景里，比如缓存。发生修改时候做copy，新老版本分离，保证读的高性能，适用于以读为主的情况。

LinkedList 的实现原理

* 基于链表实现的，所以它的插入和删除操作比ArrayList更加高效。但也是由于其为基于链表的，所以随机访问的效率要比 ArrayList 差。

[ConcurrentHashMap的实现原理](https://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/)

## Java内存模型

* 重排序
  + 编译器生成指令的次序，可以不同于源代码所暗示的“显然”版本。
  + 处理器可以乱序或者并行的执行指令。
  + 缓存会改变写入提交到主内存的变量的次序。
* 内存可见性

在现代可共享内存的多处理器体系结构中每个处理器都有自己的缓存，并周期性的与主内存协调一致。

假设线程 A 写入一个变量值 V，随后另一个线程 B 读取变量 V 的值，在下列情况下，线程 B 读取的值可能不是线程 A 写入的最新值：

* + 执行线程 A 的处理器把变量 V 缓存到寄存器中。
  + 执行线程 A 的处理器把变量 V 缓存到自己的缓存中，但还没有同步刷新到主内存中去。
  + 执行线程 B 的处理器的缓存中有变量 V 的旧值。
* Happens-before 关系

如果线程A与线程B满足happens-before关系，则线程A执行动作的结果对于线程B是可见的

* + 程序次序法则：如果在程序中，所有动作 A 出现在动作 B 之前，则线程中的每动作 A 都 happens-before 于该线程中的每一个动作 B。
  + 监视器锁法则：对一个监视器的解锁 happens-before 于每个后续对同一监视器的加锁。
  + Volatile 变量法则：对 Volatile 域的写入操作 happens-before 于每个后续对同一 Volatile 的读操作。
  + 传递性：如果 A happens-before 于 B，且 B happens-before C，则 A happens-before C。

## ConcurrentHashMap的结构分析

* ConcurrentHashMap类中包含两个静态内部类HashEntry和Segment
* HashEntry用来封装散列映射表中的键值对

static final class HashEntry<K,V> {

       final K key;

       final int hash;

       volatile V value;

final HashEntry<K,V> next;

}

在 ConcurrentHashMap 中，在散列时如果产生“碰撞”，将采用“分离链接法”来处理“碰撞”：把“碰撞”的 HashEntry 对象链接成一个链表。由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。

* Segment类

继承于ReentrantLock类，从而使得Segment对象能充当锁的角色

static final class Segment<K,V>

extends ReentrantLock implements Serializable {

       /\*\* 包含的HashEntry元素的个数 \*/

       transient volatile int count;

       /\*\* table 被更新的次数 \*/

       transient int modCount;

       /\*\* 当table中包含的HashEntry元素的个数超过本变量值时，触发table的再散列 \*/

       transient int threshold;

/\*\* table是由HashEntry对象组成的数组。

如果散列时发生碰撞，碰撞的 HashEntry对象就以链表的形式链接成一个链表。

table 数组的数组成员代表散列映射表的一个桶。

       每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分。

       如果并发级别为16,table则守护ConcurrentHashMap包含的桶总数的1/16\*/

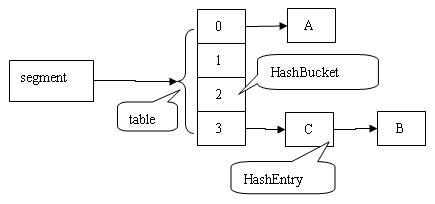
       transient volatile HashEntry<K,V>[] table;

       /\*\* 装载因子 \*/

       final float loadFactor;

}

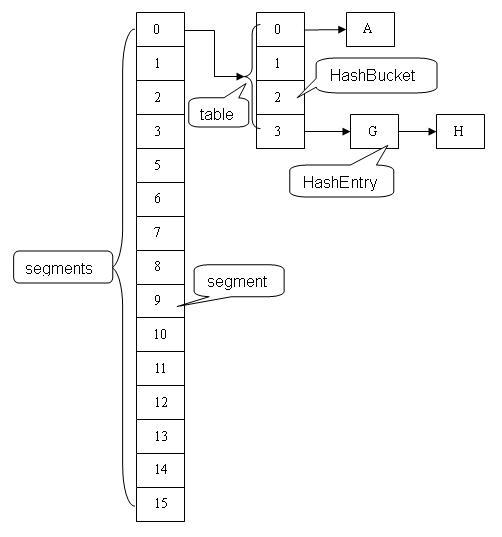
下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图。



* ConcurrentHashMap类

ConcurrentHashMap 在默认并发级别会创建包含 16 个 Segment 对象的数组。每个 Segment 的成员对象 table 包含若干个散列表的桶。每个桶是由 HashEntry 链接起来的一个链表。如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

下面是ConcurrentHashMap的结构示意图。



## 用分离锁实现多个线程间的并发写操作

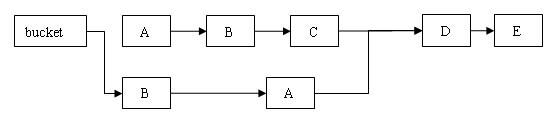
* 在ConcurrentHashMap中，线程对映射表做读操作时，*一般情况下*不需要加锁就可以完成，对容器做结构性修改的操作才需要加锁。
* put：根据key计算出对应的hash值，然后根据hash值找到对应的Segment对象，再在这个Segment中执行具体的put操作。

加锁操作是针对（key的hash值对应的）某个具体的Segment，锁定的是该Segment而不是整个ConcurrentHashMap。因为插入键/值对操作只是在这个 Segment包含的某个桶中完成，不需要锁定整ConcurrentHashMap。此时，其他写线程对另外 15 个Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。同时，所有读线程几乎不会因本线程的加锁而阻塞（除非读线程刚好读到这个 Segment 中某个 HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值）

## 用HashEntery对象的不变性来降低读操作对加锁的需求

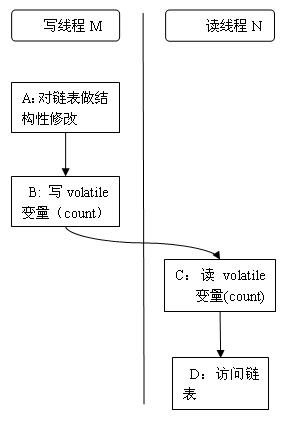
* HashEntry中的key，hash，next 都声明为final型。这意味着，不能把节点添加到链接的中间和尾部，也不能在链接的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变。这个特性可以大大降低处理链表时的复杂性。
* 同时，HashEntry类的value域被声明为volatile型，Java的内存模型可以保证：某个写线程对value域的写入马上可以被后续的某个读线程“看”到。在 ConcurrentHashMap中，不允许用null作为键和值，当读线程读到某个 HashEntry的value域的值为null时，便知道产生了冲突——发生了重排序现象，需要加锁后重新读入这个value值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。
* 写入：非结构性修改操作：只是更改某个HashEntry的value域的值。由于对 volatile变量的写入操作将与随后对这个变量的读操作进行同步。当一个写线程修改了某个HashEntry的value域后，另一个读线程读这个值域，Java内存模型能够保证读线程读取的一定是更新后的值。所以，写线程对链表的非结构性修改能够被后续不加锁的读线程“看到”。
* 写入：结构性修改，实质上是对某个桶指向的链表做结构性修改
  + clear：操作只是把所有的桶“置空”，每个桶之前引用的链表依然存在，只是桶不再引用到这些链表（所有链表的结构并没有被修改）。正在遍历某个链表的读线程依然可以正常执行对该链表的遍历。
  + put：put操作如果需要插入一个新节点到链表中时, 会在链表头部插入这个新节点。此时，链表中的原有节点的链接并没有被修改。也就是说：插入新健/值对到链表中的操作不会影响读线程正常遍历这个链表。
  + remove：首先根据散列码找到具体的链表；然后遍历这个链表找到要删除的节点；最后把待删除节点之后的所有节点原样保留在新链表中，把待删除节点之前的每个节点克隆到新链表中。下面通过图例来说明remove操作。假设写线程执行remove操作，要删除链表的C节点，另一个读线程同时正在遍历这个链表。

 4. 执行删除之前的原链表：



删除节点 C 之后的所有节点原样保留到新链表中；删除节点 C 之前的每个节点被克隆到新链表中，*注意：它们在新链表中的链接顺序被反转了*。在执行 remove 操作时，原始链表并没有被修改，也就是说：读线程不会受同时执行 remove 操作的并发写线程的干扰。

## 用Volatile变量协调读写线程间的内存可见性



V get(Object key, int hash) {

if(count != 0) {       // 首先读 count 变量

   HashEntry<K,V> e = getFirst(hash);

     while(e != null) {

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

         V v = e.value;

          if(v != null)

           return v;

            //如果读到value域为null，说明发生了重排序，加锁后重新读取

          return readValueUnderLock(e);

        }

        e = e.next;

      }

    }

    return null;

}

## ConcurrentHashMap的高并发性主要来自于三个方面：

* 用分离锁实现多个线程间的更深层次的共享访问。
* 用HashEntery对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求
* 通过对同一个Volatile变量的写/读访问，协调不同线程间读/写操作的内存可见性。