1：HashMap集合等Map集合底层数据结构有所改变

从jdk1.8开始HashMap底层数据结构是数组+链表+红黑树

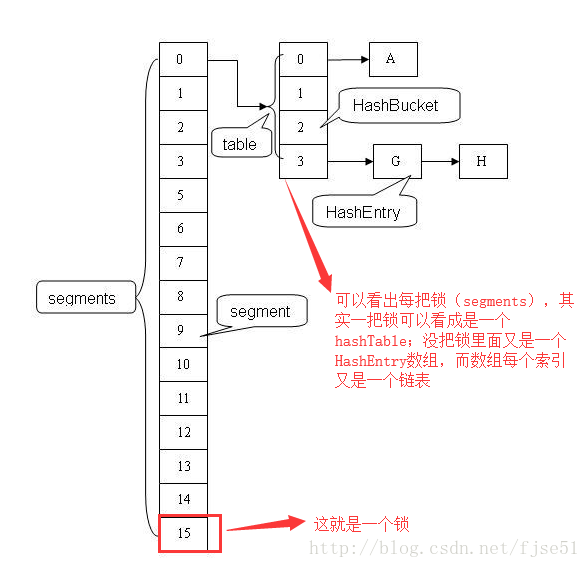
（只有当集合中如果某节点的链表的长度大于8，并且总长度大于64时，链表的结构变为红黑树结构），红黑树的效率比链表查询高（但是除了添加元素，添加元素时效率可能比链表慢一点，但是其他对元素的操作效率明显提高，时间复杂度最高为O(long n)）

同理HashSet，HashTable等也都变了

2：ConcurrentHashMap

原来：并发级别（并发度）为16，即concurretnLevel-16, 采用锁分段机制，也就是说该容器初始值时锁的个数为16（即Segment[]长度），用户也可以自定义并发级别，之所以它比hashTable的效率高，是因为，每把锁只锁住容器中一段数据，这样当一个线程在访问某个元素时，它首先会找到对应的Segment，也就是对应的该段的锁，进行加锁操作，这样其他段的数据此时可以被其他线程访问，因为当前只对该段数据加速，其他段如果没有加锁，就可以并发访问其他段，（注意咋get的时候不管你加没加锁，都可以并发访问的），而hashTable并发情况下效率极低，因为每当访问容器时，它都对整个容器加锁，时整个容器，包括get方法，也加锁。（详细了解请点击: <https://www.cnblogs.com/slwenyi/p/6393808.html>）:

原来：



源码：

|  |
| --- |
| public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>  implements ConcurrentMap<K, V>, Serializable {  // 将整个hashmap分成几个小的map，每个segment都是一个锁；与hashtable相比，这么设计的目的是对于put, remove等操作，可以减少并发冲突，对  // 不属于同一个片段的节点可以并发操作，大大提高了性能  final Segment<K,V>[] segments;  // 本质上Segment类就是一个小的hashmap，里面table数组存储了各个节点的数据，继承了ReentrantLock, 可以作为互拆锁使用  static final class Segment<K,V> extends ReentrantLock implements Serializable {  transient volatile HashEntry<K,V>[] table;  transient int count;  }  // 基本节点，存储Key， Value值  static final class HashEntry<K,V> {  final int hash;  final K key;  volatile V value;  volatile HashEntry<K,V> next;  }  } |

现在：

**同步机制：无锁算法（CAS算法）+ synchronized保证并发更新**

**数据结构：table数组+链表+红黑树**

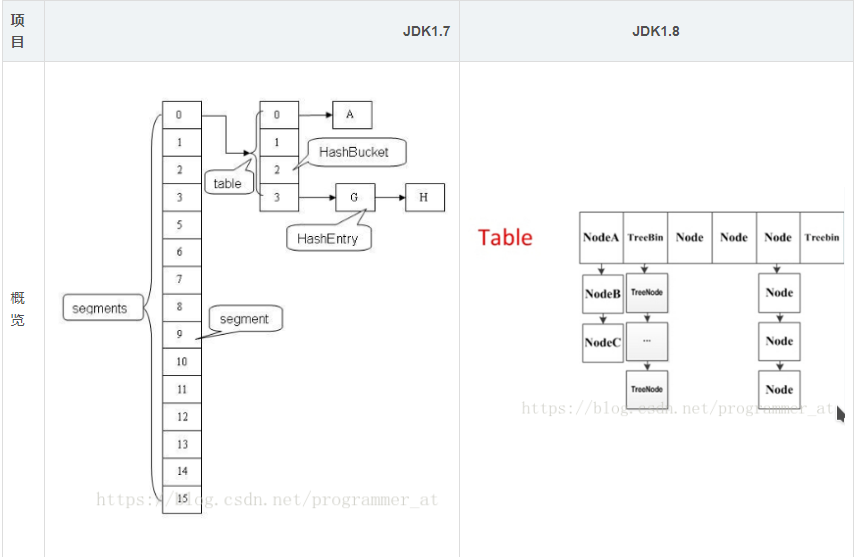
1:取消原来的**segments字段，直接采用transient volatile hashEnty<K,V>[] table保存数据，采用table数组元素作为锁，这样可以对每一个元素进行加锁，进一步减少了并发冲突的概率**

**2：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。（table的扩容采用CAS算法）**

**源码：**

|  |
| --- |
| final V putVal(K key, V value, boolean onlyIfAbsent) {  if (key == null || value == null) throw new NullPointerException();  int hash = spread(key.hashCode());  int binCount = 0;  for (Node<K,V>[] tab = table;;) {  Node<K,V> f; int n, i, fh;  // 如果table为空，初始化；否则，根据hash值计算得到数组索引i，如果tab[i]为空，直接新建节点Node即可。注：tab[i]实质为链表或者红黑树的首节点。  if (tab == null || (n = tab.length) == 0)  tab = initTable();  else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {  if (casTabAt(tab, i, null,  new Node<K,V>(hash, key, value, null)))  break; // no lock when adding to empty bin  }  **// 如果tab[i]不为空并且hash值为MOVED，说明该链表正在进行transfer操作，返回扩容完成后的table。CAS**  else if ((fh = f.hash) == MOVED)  tab = helpTransfer(tab, f);  else {  V oldVal = null;  **// 针对首个节点进行加锁操作，而不是segment，进一步减少线程冲突**  synchronized (f) {  if (tabAt(tab, i) == f) {  if (fh >= 0) {  binCount = 1;  for (Node<K,V> e = f;; ++binCount) {  K ek;  // 如果在链表中找到值为key的节点e，直接设置e.val = value即可。  if (e.hash == hash &&  ((ek = e.key) == key ||  (ek != null && key.equals(ek)))) {  oldVal = e.val;  if (!onlyIfAbsent)  e.val = value;  break;  }  // 如果没有找到值为key的节点，直接新建Node并加入链表即可。  Node<K,V> pred = e;  if ((e = e.next) == null) {  pred.next = new Node<K,V>(hash, key,  value, null);  break;  }  }  }  // 如果首节点为TreeBin类型，说明为红黑树结构，执行putTreeVal操作。  else if (f instanceof TreeBin) {  Node<K,V> p;  binCount = 2;  if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,  value)) != null) {  oldVal = p.val;  if (!onlyIfAbsent)  p.val = value;  }  }  }  }  if (binCount != 0) {  // 如果节点数>＝8，那么转换链表结构为红黑树结构。  if (binCount >= TREEIFY\_THRESHOLD)  treeifyBin(tab, i);  if (oldVal != null)  return oldVal;  break;  }  }  }  // 计数增加1，有可能触发transfer操作(扩容)。  addCount(1L, binCount);  return null;  }  具体：请看：<https://blog.csdn.net/programmer_at/article/details/79715177>  <https://www.jianshu.com/p/e694f1e868ec> |

区别：





3：Lambda表达式：

代码入门：请看练习

3.1Lambda基本语法：

1：jdk操作符：”->”

2：其他函数式接口：多参数，满足不同的参数的列表

记住截图：第五集：截图其他接口

3：jdk1.8强大的Stream

数据源，集合 一系列流水线操作

数组等 产生新的流

3.1 笔记截图 第七集开始