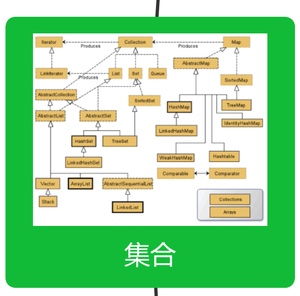
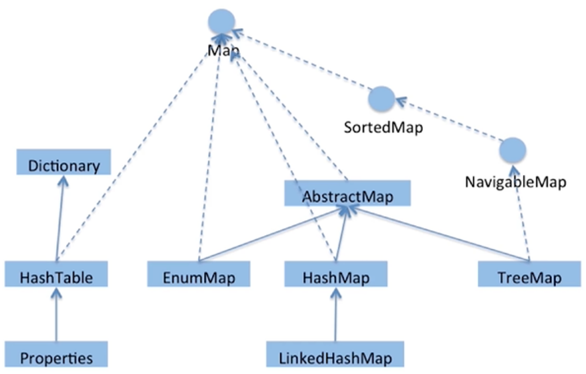
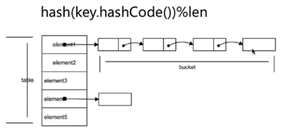




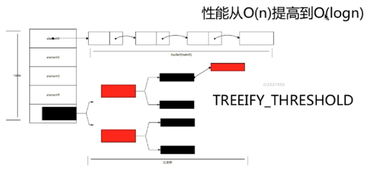
Map

HashMap

HashMap（Java8 以前）：数组+链表



HashMap（Java8 及以后）：数组+链表+红黑树

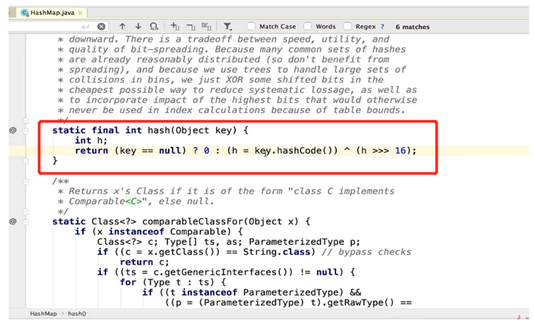


HashMap：put方法的逻辑 TREEIFY_THRESHOLD=8，MIN_TREEIFY_CAPACITY=64

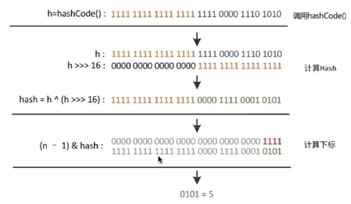
1、若HashMap未被初始化，则进行初始化操作；
2、对Key求Hash值，依据Hash值计算下标；
3、若未发生碰撞，则直接放入桶中；
4、若发生碰撞，则以链表的方式链接到后面；
5、若链表长度超过阈值，且HashMap元素超过最低树化容量，则将链表转成红黑树；
6、若节点已经存在，则用新值替换旧值；
7、若桶满了(默认容量16*扩容因子0.75)，就需要resize(扩容2倍后重排)；

HashMap：如何有效减少碰撞

- 扰动函数：促使元素位置分布均匀，减少碰撞机率
- 使用final对象，并采用合适的equals()和hashCode()方法



HashMap：从获取hash到散列的过程 int范围：-2147483648到2147483647



HashMap：扩容的问题

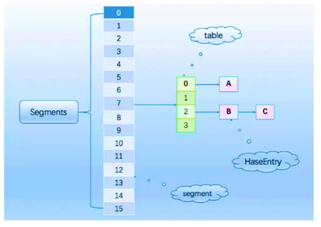
- 多线程环境下，调整大小会存在条件竞争，容易造成死锁
- rehashing是一个比较耗时的过程

HashTable

Hashtable

- 早期Java类库提供的哈希表的实现
- 线程安全：涉及到修改Hashtable的方法，使用synchronized修饰
- 串行化的方式运行，性能较差

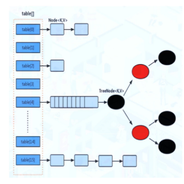
早期的ConcurrentHashMap：通过分段锁Segment来实现



数组+链表

CurrentHashMap

当前的ConcurrentHashMap：CAS+synchronized使锁更细化



数组+链表+红黑树

ConcurrentHashMap：put方法的逻辑

1. 判断Node[]数组是否初始化，没有则进行初始化操作
2. 通过hash定位数组的索引坐标，是否有Node节点，如果没有则使用CAS进行添加（链表的头节点），添加失败则进入下次循环。
3. 检查到内部正在扩容，就帮助它一块扩容。
4. 如果f!=null，则使用synchronized锁住f元素（链表/红黑二叉树的头元素）
   4.1 如果是Node(链表结构)则执行链表的添加操作。
   4.2 如果是TreeNode(树型结构)则执行树添加操作。
5. 判断链表长度已经达到临界值8，当然这个8是默认值，大家也可以去做调整，当节点数超过这个值就需要把链表转换为树结构。

三者的区别

- HashMap线程不安全，数组+链表+红黑树
- Hashtable线程安全，锁住整个对象，数组+链表
- ConccurentHashMap线程安全，CAS+同步锁，数组+链表+红黑树
- HashMap的key、value均可为null，而其他的两个类不支持