### **putHashTable和ConcurrentHashMap**

### ****HashTable****

* 底层数组+链表实现，无论key还是value都**不能为null**，线程**安全**，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化。

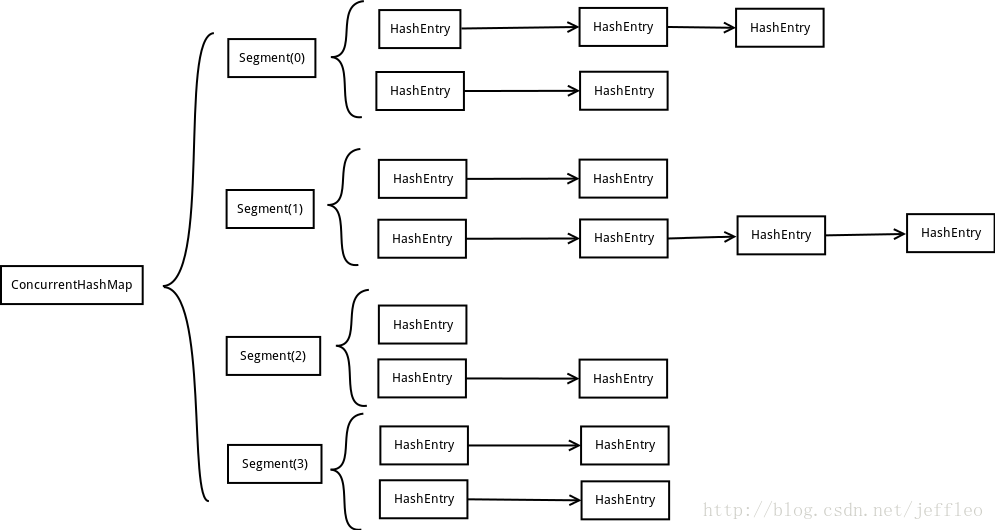
## **效率低下的HashTable容器**

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

**ConcurrentHashMap**

### 

**锁分段技术**：首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。



* 底层采用分段的数组+链表实现，线程**安全**
* 通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)
* Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术
* 有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁
* **扩容：**段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

ConcurrentHashMap提供了与Hashtable和SynchronizedMap不同的锁机制。Hashtable中采用的锁机制是一次锁住整个hash表，从而在同一时刻只能由一个线程对其进行操作；而ConcurrentHashMap中则是一次锁住一个桶。

ConcurrentHashMap默认将hash表分为16个桶，诸如get、put、remove等常用操作只锁住当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

**ConcurrentHashMap扩容机制**

当ConcurrentHashMap扩容之后，一个元素的位置要么不发生变化，要么变化为原桶的序号+(2^k-1)。

理由：

int idx = e.hash & sizeMask;

而sizeMask = newTable.length - 1  即sizeMask = 11...1，即全是1，共k个1。获取序号的算法是用元素的hash值与sizeMask做与的操作。这样得到的idx实际上就是元素的hashcode值的低k位的值。而原table的sizeMask也全是1的二进制，不过总共是k-1位。那么原table的idx就是元素的hashcode的低k-1位的值。所以说如果元素的hashcode的第k为如果是0，那么元素在新桶的序号就是和原桶的序号是相等的。如果第k位的值是1，那么元素在新桶的序号就是原桶的序号+(2^k-1)。所以说只可能是这两个值。那么上面的那个newTable[lastIdx] = lastRun;就没问题了，newTable中新序号处此时肯定是空的。