JDK1.8版本以前的HashMap和 ConcurrentHashMap

HashMap 底层 ： 数组、存一个个entry，初始为null

初始长度：2的幂 ，因为索引从0开始 ，所以相与的是 2的幂-1 。**原因**

单机下的 put(index重复时该位置变成 ？) \ get

单机下的rehash\resize ： HashMap 扩容 ，元素越来越多，逐渐饱和

Resize会将相同key的顺序相反

Resize在单线程情况是不安全的。因为resize不安全，所以当并发插入数据导致resize后，可能导致下次读操作死循环，**链表内死循环**

并发下，借助 ： ConcurrentHashMap

ConcurrentHashMap

底层：Segment ，一个ConcurrentHashMap，对应多个Segment,Segment中含多个Entry

Segment自制，实现并发的读写，只有在同一个Segment中写才需要加锁

计算Size ： 通过乐观锁思想不断重试（比对修改次数和上次是否一样），到一定阈值就悲观锁思想，对每个Segment都加锁

关于定位：ConcurrentHashMap每次都进行两次hash，一次定位Segment，一次定位Segment中具体位置。

HashMap 存储一个个键值对。每个键值对称 Entry , 这些Entry分散存储在一个数组中，这个数组就是HashMap的主体。

**关键字： 底层一个个Entry 构成的 数组，Entry实体存的是键值对，为HashMap主体。HashMap数组初始值都为null .**

**HashMap 长度**

默认初始数组长度 ：**16**

每次扩容或初始化，**长度**必须是 **2 的 幂 ， 因为2的幂 的二进制 减1 （因为索引都是从0开始的） 都是 1 、 11 、 111 、 1111…. 全 1 。选这类型的数字长度主要和 Key（键值）映射到 索引（index）的Hash算法有关。**

**Key 映射到HashMap数组上的索引 是通过计算Key 的Hash 值 并于当前长度的二进制进行相与（&）运算，达到均匀分布在数组上，当然 HashCode 对长度取模也可达到相同效果，不过效率太低。 所以，为了均匀分布，相与的长度必须是每位全为1 的二进制，保证映射的位置完全由Key的HashCode决定并且是均匀的。因为1 与任何数相与都等于他本身。而如果有 0 ，0和任何数相与都等于 0 ，导致存在很多重复的可能 。**

**单线程下**

HashMap 的 put :

put(key，value)

即使是通过2的幂确定索引位置，仍然存在计算出的index 重复的情况。所以，entry并不只是一个数组的实体，也是一个链表的头。即 如果相同的 index, 采用头插法。新entry指向就的entry 存于数组中。因为设计HashMap时认为后插入的节点更可能被访问。

get()

通过计算key的hashcode 并得出index 后，找到对应位置。如果是个链表（即由多个值，仍需一个个遍历比对value查找。）

高并发下 ： **注意，HashMap并非线程安全的。**

当元素越来越多，原HashMap数组逐渐饱和，需要进行Resize

Capacity ：HashMap 的最大长度，必须是2 的幂

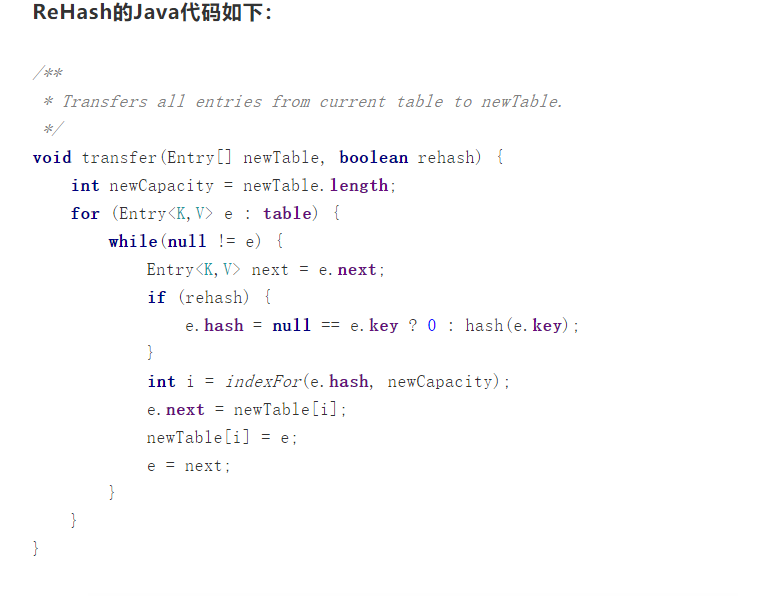
LoadFactor : 负载因子， 默认 0.75f

ReSize 条件 ： HashMap.Size >= Capacity \* LoadFactor **当前长度达到总长的3/4**

**ReSize 步骤 ：**

1. **创建一个新的空的Entry数组，长度为原来2倍**
2. **ReHash ：遍历原来的Entry数组，重新计算每个Entry 的Key对应映射到的新的index 。（长度变后，映射规则也改变）**

**此是ReHash和 ReSize的主要 代码**

****

**注意，此法得出的转移后的数组，在每个位置上，如果有重复，则顺序将会相反。即例如原来 index = 4 上元素 为 k1 → k2 → k3**

**转移后 元素为 k1 ← k2 ← k3**

**这个ReSize 步骤不是线程安全的，存在问题。当多个线程进行ReSize时，有多个相同key的位置可能出现环的情况，即 如 k1.next = k2 ，k2.next = k1 . 当进行get操作时，如果刚好在这个 key 里面找，则死循环。**

**主要原因：**

**当两个线程同时ReSize时，线程Bresize到具有多个相同key的位置时，被挂起，线程A开始Resize并执行结束，此时线程B恢复，但是仍拿着原来的 值和next开始resize已经被Aresize好的链表，且，resize后链表的值是相反的，所以和原来的顺序不一致，最终导致环。**

**判断链表是否有环 ：可用 “追逐”判断。**

**所以，HashMap不适用在多线程情况，需要借助 ConcurrentHashMap**

ConcurrentHashMap

并发安全的可采用 HashTable 或 Collections.synchronizedMap ，但是这二者性能欠佳。都是给整个集合加锁，变成了单线程操作。

而为了提高性能，ConcurrentHashMap ： 性能 + 并发安全 。

**关键概念 ： Segment**

**Segment**

**本身也是一个HashMap对象，Segment包含了一个HashEntry数组，数组中的每个HashEntry即是一个键值对，也就是一个链表的头结点**

**Segment1**

↓

{E1,E2,E3，null}

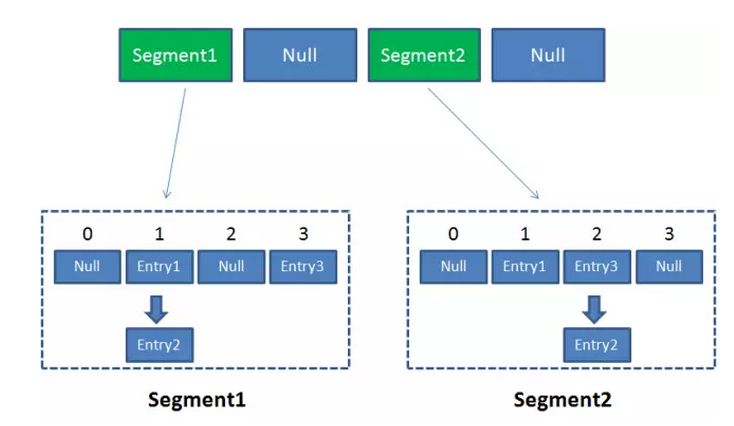
↓ ↓

E4 E6

↓

E5

**ConcurrentHashMap 中的元素，就是一个个Segment,而非一个个Entry。Segment下再包含一个个Entry**

****

这样设计的优势： **锁分段技术 。每一个Segment都是一个自治区，读写自治，Segment之间互不影响。可在不同Segment进行同步读写不加安全策略，可以做到并行处理同一个ConcurrentHashMap对象，而在同一个Segment中才需要进行并发安全的限制。**

**同一Segment写需要加锁，每个Segment各持一把锁（对象就是监视器），降低了加锁粒度，并发效率更高**

注意，实际上，为了保证均衡，ConcurrentHashMap进行hash计算是进行了两次，一次定位到Segment，一次是在Segment中定位！

ConcurrentHashMap 的 读写

Get

计算key的hash，得到hash

定位Segment

再详细定位到Segment中的具体位置

Put

计算key的hash，得到hash

定位Segment

**获得 重入锁 ！！ 重入锁 ReentrantLock，在进入Segment时必须获得锁**

再次定位详细位置

插入或读取HashEntry对象

**释放锁**

**关于 获取 Size**

因为获取size自然是所有segment所含个数相加，当出现计算过程前面的Segment进行了Put，就可能导致size错误，为此，ConcurrentHashMap进行如下步骤：

* 设置一个计数值，记录重试的次数。
* 记录每个Segment所含个数
* 记录每个Segment修改的次数
* 若，Segment的总修改次数大于上次的修改次数，说明有修改，计数值+1，重新统计，否则，统计结束。
* 若计数值大于一个阈值，直接给每个Segment加锁，再重新统计步骤，这次一定是正确的。
* 释放锁，统计结束。

**PS： 这种统计Size数 是 乐观锁悲观锁结合 的思想，不对就重新尝试，实在不行就直接锁住。**