# jdk

## map

### hashmap

#### 红黑树

红黑树有5个原则：

1. 每个节点是红色或者黑色的
2. 根节点必须是黑色的
3. 每个叶子节点都是黑色的空节点（NIL节点），即叶子节点不存储数据
4. 红色节点的两个子节点必须都是黑色的（即路径中不能存在两个连续的红色节点）
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

定义了两个属性：节点的 黑深 以及 树的 黑高。

### concurrentHashMap

HashTable是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占，相当于所有线程进行读写时都去竞争一把锁，导致效率非常低下。ConcurrentHashMap可以做到读取数据不加锁，并且其内部的结构可以让其在进行写操作的时候能够将锁的粒度保持地尽量地小，允许多个修改操作并发进行，其关键在于使用了锁分段技术。它使用了多个锁来控制对hash表的不同部分进行的修改。对于JDK1.7版本的实现, ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）。

对于JDK1.7版本的实现，ConcurrentHashMap 为了提高本身的并发能力，在内部采用了一个叫做 Segment 的结构，一个 Segment 其实就是一个类 Hash Table 的结构，Segment 内部维护了一个链表数组，我们用下面这一幅图来看下 ConcurrentHashMap 的内部结构,从下面的结构我们可以了解到，ConcurrentHashMap 定位一个元素的过程需要进行两次Hash操作，第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是 Hash 的过程要比普通的 HashMap 要长，但是带来的好处是写操作的时候可以只对元素所在的 Segment 进行操作即可，不会影响到其他的 Segment，这样，在最理想的情况下，ConcurrentHashMap 可以最高同时支持 Segment 数量大小的写操作（刚好这些写操作都非常平均地分布在所有的 Segment上），所以，通过这一种结构，ConcurrentHashMap 的并发能力可以大大的提高。我们用下面这一幅图来看下ConcurrentHashMap的内部结构详情图，如下:

