

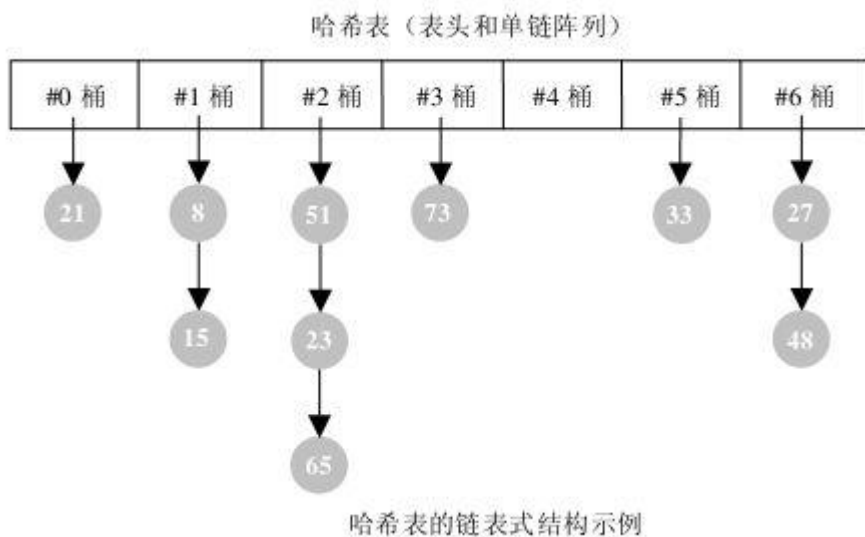
1.介绍

哈希表就是一种以 键-值(*key-indexed*) 存储数据的结构，我们只要输入待查找的值即 *key*，即可查找到其对应的值。

哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

2.链式哈希表

链式哈希表从根本上说是由一组链表构成。每个链表都可以看做是一个“桶”，我们将所有的元素通过散列的方式放到具体的不同的桶中。插入元素时，首先将其键传入一个哈希函数（该过程称为哈希键），函数通过散列的方式告知元素属于哪个“桶”，然后在相应的链表头插入元素。查找或删除元素时，用同样的方式先找到元素的“桶”，然后遍历相应的链表，直到发现我们想要的元素。因为每个“桶”都是一个链表，所以链式哈希表并不限制包含元素的个数。然而，如果表变得太大，它的性能将会降低。



3.应用场景

我们熟知的缓存技术（比如 redis、memcached）的核心其实就是在内存中维护一张巨大的哈希表，还有大家熟知的 HashMap、CurrentHashMap 等的应用。

ConcurrentHashMap 与 HashMap 等的区别

1.HashMap

我们知道 HashMap 是线程不安全的，在多线程环境下，使用 Hashmap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap。

2.HashTable

HashTable 和 HashMap 的实现原理几乎一样，差别无非是

- HashTable 不允许 key 和 value 为 null
- HashTable 是线程安全的

但是 HashTable 线程安全的策略实现代价却太大了，简单粗暴，get/put 所有相关操作都是 synchronized 的，这相当于**给整个哈希表加了一把大锁**。

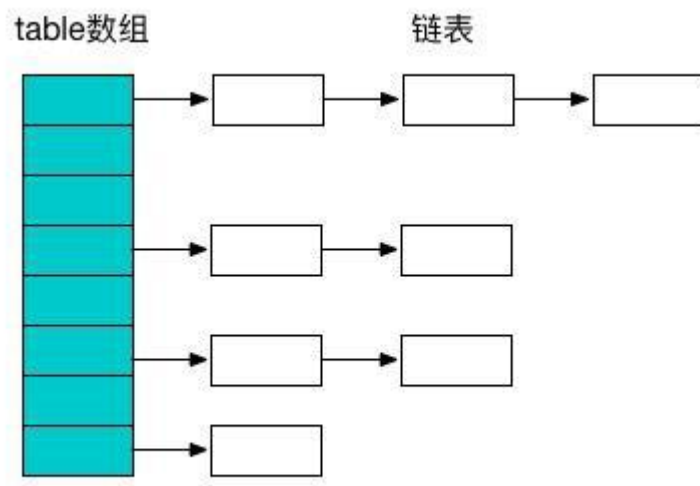
多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作**串行化**，在竞争激烈的并发场景中性能就会非常差。

3.ConcurrentHashMap

主要就是为了应对 hashmap 在并发环境下不安全而诞生的，

ConcurrentHashMap 的设计与实现非常精巧，大量的利用了 volatile，final，CAS 等 lock-free 技术来减少锁竞争对于性能的影响。

我们都知道 Map 一般都是**数组+链表结构**（JDK1.8 该为数组+红黑树）。



ConcurrentHashMap 避免了对全局加锁改成了局部加锁操作，这样就极大地提高了并发环境下的操作速度，由于 ConcurrentHashMap 在 JDK1.7 和 1.8 中的实现非常不同，接下来我们谈谈 JDK 在 1.7 和 1.8 中的区别。

JDK1.7 版本的 CurrentHashMap 的实现原理

在 JDK1.7 中 ConcurrentHashMap 采用了**数组+Segment+分段锁**的方式实现。

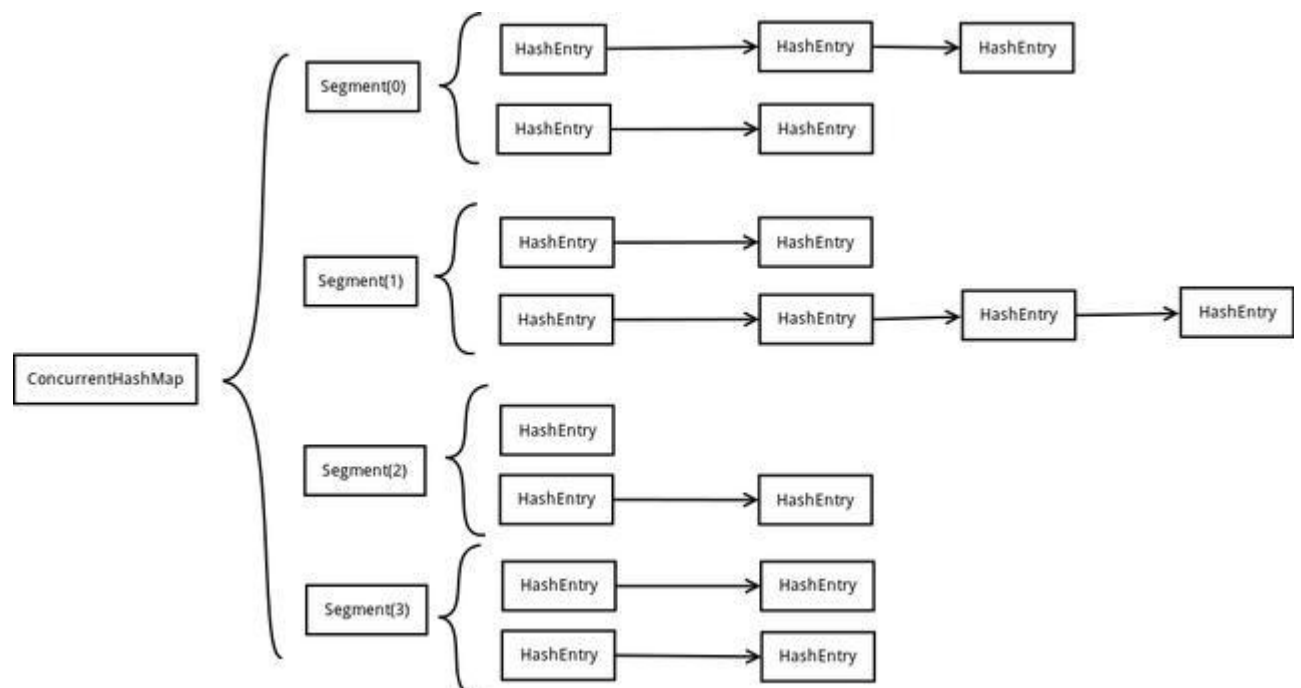
1.Segment(分段锁)

ConcurrentHashMap 中的**分段锁称为 Segment** ,它即类似于 HashMap 的结构，即内部拥有一个 Entry 数组，数组中的每个元素又是一个链表,同时又是一个 ReentrantLock (Segment 继承了 ReentrantLock)。

2.内部结构

ConcurrentHashMap 使用分段锁技术，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，能够实现真正的并发访问。如下图是

ConcurrentHashMap 的内部结构图：



从上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。

3.该结构的优劣势

坏处

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

好处

写操作的时候可以只对元素所在的 Segment 进行加锁即可，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。

JDK1.8 版本的 CurrentHashMap 的实现原理

JDK8 中 ConcurrentHashMap 参考了 JDK8 HashMap 的实现，采用了**数组+链表+红黑树**的实现方式来设计，**内部大量采用 CAS 操作**，这里我简要介绍下 CAS。

CAS 是 compare and swap 的缩写，即我们所说的比较交换。cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源

锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置 (V)、预期原值 (A) 和新值(B)。

如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。

CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

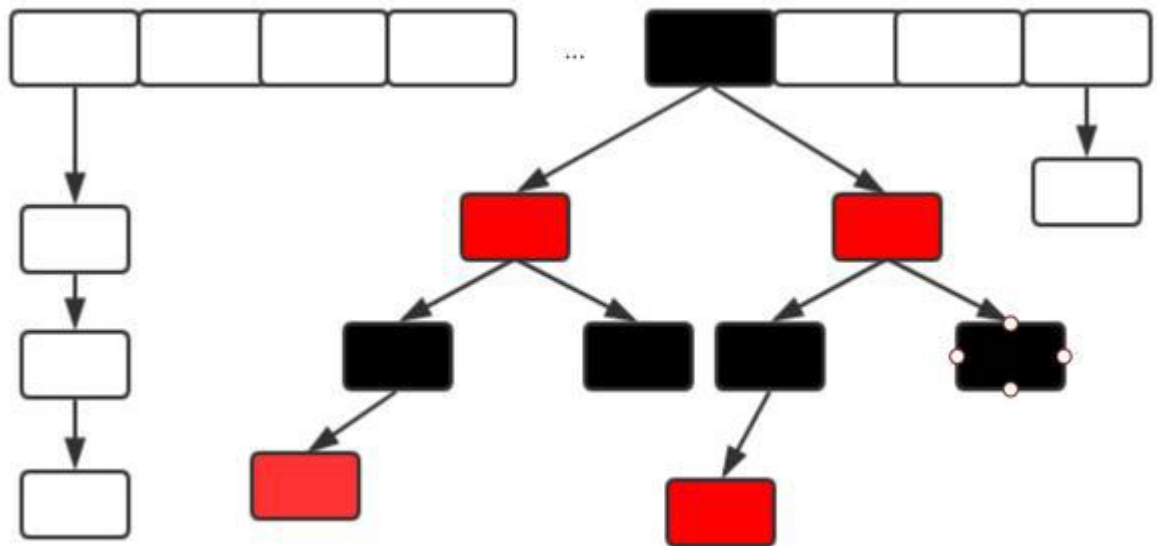
JDK8 中彻底放弃了 Segment 转而采用的是 Node，其设计思想也不再是 JDK1.7 中的分段锁思想。

Node: 保存 *key*, *value* 及 *key* 的 *hash* 值的数据结构。其中 *value* 和 *next* 都用 *volatile* 修饰，保证并发的可见性。

```
class Node<K,V> implements Map.Entry<K,V> {  
  
    final int hash;  
  
    final K key;  
  
    volatile V val;  
  
    volatile Node<K,V> next;  
  
    //... 省略部分代码}
```

Java8 `ConcurrentHashMap` 结构基本上和 Java8 的 `HashMap` 一样，不过保证线程安全性。

在 JDK8 中 `ConcurrentHashMap` 的结构，由于引入了红黑树，使得 `ConcurrentHashMap` 的实现非常复杂，我们都知道，红黑树是一种性能非常好的二叉查找树，其查找性能为 $O(\log N)$ ，但是其实现过程也非常复杂，而且可读性也非常差，Doug Lea 的思维能力确实不是一般人能比的，早期完全采用链表结构时 Map 的查找时间复杂度为 $O(N)$ ，JDK8 中 `ConcurrentHashMap` 在链表的长度大于某个阈值的时候会将链表转换成红黑树进一步提高其查找性能。



总结

其实可以看出 JDK1.8 版本的 ConcurrentHashMap 的数据结构已经接近 HashMap，相对而言，ConcurrentHashMap 只是增加了同步的操作来控制并发，从 JDK1.7 版本的 ReentrantLock+Segment+HashEntry，到 JDK1.8 版本中 synchronized+CAS+HashEntry+红黑树。

1.数据结构：取消了 Segment 分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。

2.保证线程安全机制：JDK1.7 采用 segment 的分段锁机制实现线程安全，其中 segment 继承自 ReentrantLock。JDK1.8 采用 CAS+Synchronized 保证线程安全。

3.锁的粒度：原来是对需要进行数据操作的 Segment 加锁，现调整为对每个数组元素加锁（Node）。

4.链表转化为红黑树：定位结点的 hash 算法简化会带来弊端，Hash 冲突加剧，因此当链表节点数量大于 8 时，会将链表转化为红黑树进行存储。

5.查询时间复杂度：从原来的遍历链表 $O(n)$ ，变成遍历红黑树 $O(\log N)$ 。

2020 年最新 Java 架构师系统进阶资料免费领取

需要【 一线大厂最新面试题与答案汇总 】的朋友请加 QQ 群/ 微信群 分布式/源码/性能交流 QQ 群：833977986

点击链接加入群聊免费获取面试资料：



微信扫描二维码获取资料学习

【 一线大厂最新面试题与答案汇总 】 包含阿里，京东、百 度、腾讯、等一线大厂最新面试题与面试题答案。群里还会 讨论 Kafka、Mysql、Tomcat、Docker、Spring、MyBatis、 Nginx、Netty、Dubbo、Redis、Netty、Spring cloud、 JVM、分布式、高并发、性能调优、微服务等架构师最新技能 与问题学习——进群备注好信息即可免费领取。