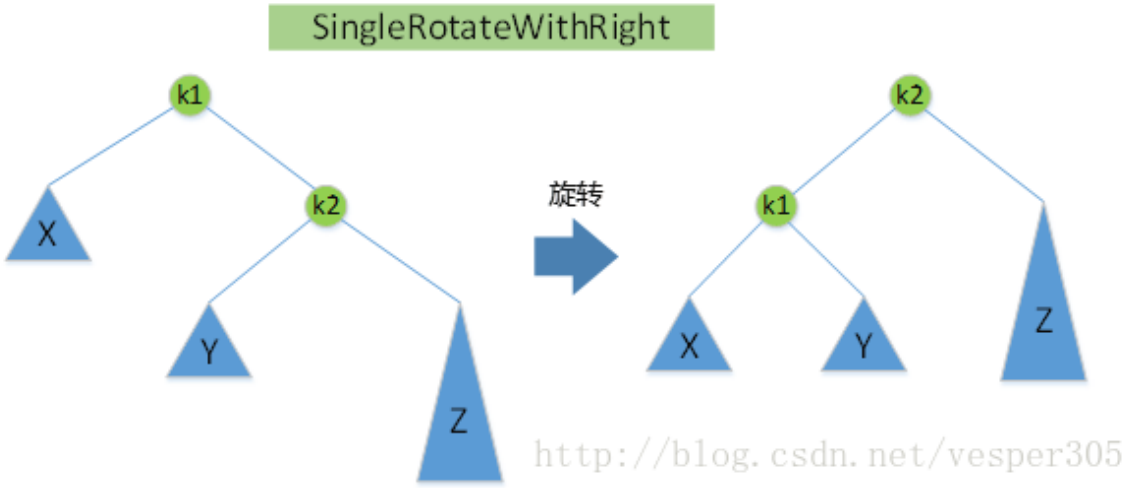
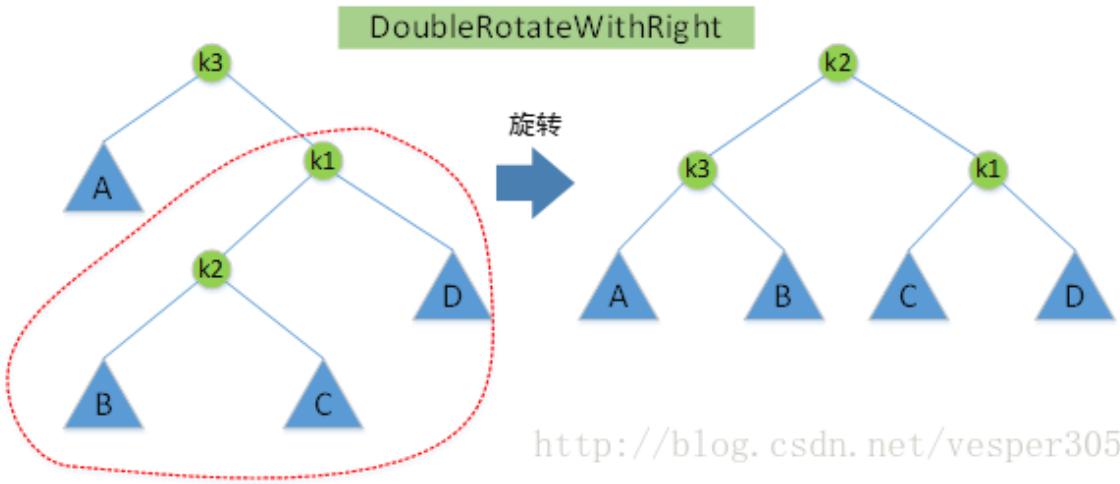
数据结构

AVL

2 AVL树是最先发明的自平衡二叉查找树。在AVL树中任何节点的两个儿子子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是O(log n)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。





一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。
从根节点到叶子节点的最长的路径不多于最短的路径的长度的两倍。对于红黑树，插入，删除，查找的复杂度都是O（log N）。

5.红黑树相比于BST和AVL树有什么优点?
红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较"便宜"的解决方案。
相比于BST，因为红黑树可以能确保树的最长路径不大于两倍的最短路径的长度，所以可以看出它的查找效果是有最低保证的。在最坏的情况下也可以保证O(logN)的，这是要好于二叉查找树的。因为二叉查找树最坏情况可以让查找达到O(N)。
红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高，所以在插入和删除中所做的后期维护操作肯定会比红黑树要耗时好多，但是他们的查找效率都是O(logN)，所以红黑树应用还是高于AVL树的. 实际上插入 AVL 树和红黑树的速度取决于你所插入的数据.如果你的数据分布较好,则比较宜于采用 AVL树(例如随机产生系列数),但是如果你想处理比较杂乱的情况,则红黑树是比较快的

Answer 1：
1. 如果插入一个node引起了树的不平衡，AVL和RB-Tree都是最多只需要2次旋转操作，即两者都是O(1)；但是在删除node引起树的不平衡时，最坏情况下，AVL需要维护从被删node到root这条路径上所有node的平衡性，因此需要旋转的量级O(logN)，而RB-Tree最多只需3次旋转，只需要O(1)的复杂度。

2. 其次， AVL的结构相较RB-Tree来说更为平衡，在插入和删除node更容易引起Tree的unbalance，因此在大量数据需要插入或者删除时，AVL需要rebalance的频率会更高。因此，RB-Tree在需要大量插入和删除node的场景下，效率更高。自然，由于AVL高度平衡，因此AVL的search效率更高。

3. map的实现只是折衷了两者在search、insert以及delete下的效率。总体来说，RB-tree的统计性能是高于AVL的。

HashMap:
这时候c的二进制第4位（从右向左数）就"失效"了，也就是说，无论第c的4位取什么值，都会导致H(c)的值一样．这时候c的第四位就根本不参与H(c)的运算，这样H(c)就无法完整地反映c的特性，增大了导致冲突的几率．

取其他合数时，都会不同程度的导致c的某些位"失效"，从而在一些常见应用中导致冲突.
但是取质数，基本可以保证c的每一位都参与H(c)的运算，从而在常见应用中减小冲突几率. .

hashmap底层实现原理

首先有一个每个元素都是链表（可能表述不准确）的数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一各链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。

HashMap:
上次我们最后聊到HashMap在多线程环境下存在线程安全问题，那你一般都是怎么处理这种情况的?
美丽迷人的面试官您好，一般在多线程的场景，我都会使用好几种不同的方式去代替:

使用Collections.synchronizedMap(Map)创建线程安全的map集合；

在SynchronizedMap内部维护了一个普通对象Map，还有排斥锁mutex，如

Hashtable

你能说说他效率低的原因么?
嗯嗯面试官，我看过他的源码，他在对数据操作的时候都会上锁

ConcurrentHashMap
不过出于线程并发度的原因，我都会舍弃前两者使用最后的ConcurrentHashMap，他的性能和效率明显高于前两者。

hashmap与hashtable区别：

实现方式不同：Hashtable 继承了 Dictionary类，而 HashMap 继承的是 AbstractMap 类。
Dictionary 是 JDK 1.0 添加的，貌似没人用过这个，我也没用过。
初始化容量不同：HashMap 的初始容量为：16，Hashtable 初始容量为：11，两者的负载因子默认都是：0.75。
扩容机制不同：当现有容量大于总容量 * 负载因子时，HashMap 扩容规则为当前容量翻倍，Hashtable 扩容规则为当前容量翻倍 + 1。
迭代器不同：HashMap 中的 Iterator 迭代器是 fail-fast 的，而 Hashtable 的

Enumerator 不是 fail-fast 的。
所以，当其他线程改变了HashMap 的结构，如：增加、删除元素，将会抛出ConcurrentModificationException 异常，而 Hashtable 则不会。

快速失败（fail—fast）是java集合中的一种机制，在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception。
Tip：这里异常的抛出条件是检测到 modCount！=expectedmodCount 这个条件。如果集合发生变化时修改modCount值刚好又设置为了expectedmodCount值，则异常不会抛出。

java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）算是一种安全机制吧。

Tip：**安全失败（fail—safe）**大家也可以了解下，java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

线程安全不一样
Hashtable 中的方法是同步的，而HashMap中的方法在默认情况下是非同步的。在多线程并发的环境下，可以直接使用Hashtable，但是要使用HashMap的话就要自己增加同步处理了。

第三、允不允许null值

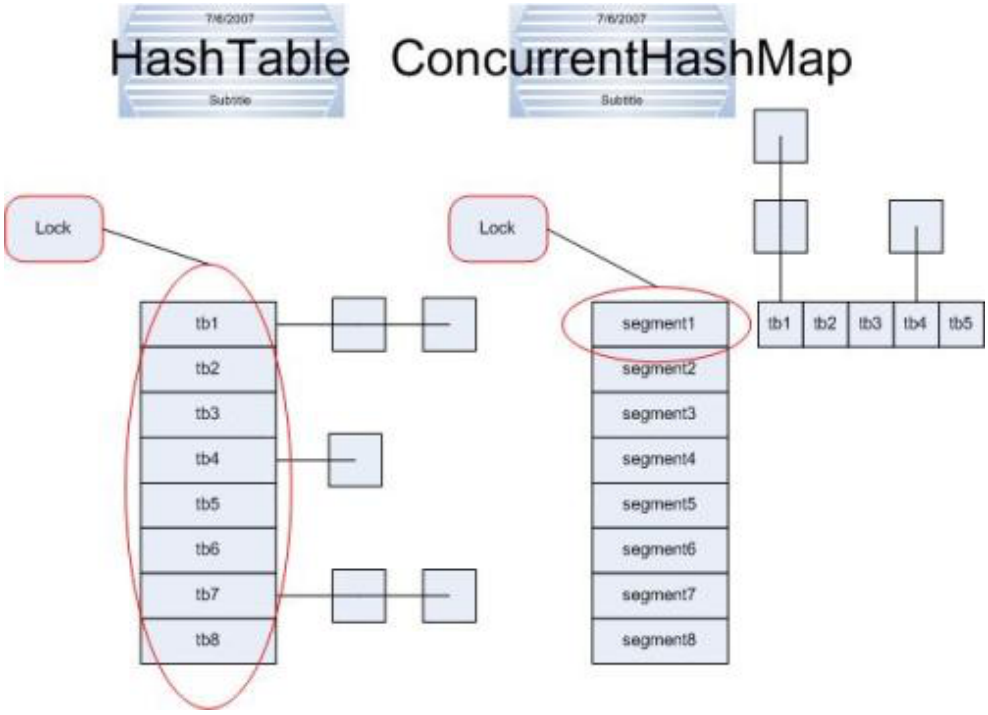
从上面的put()方法源码可以看到，Hashtable中，key和value都不允许出现null值，否则会抛出NullPointerException异常。
而在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示 HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。

第四、遍历方式的内部实现上不同
Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。

第五、哈希值的使用不同
HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

第六、内部实现方式的数组的初始大小和扩容的方式不一样

HashTable中的hash数组初始大小是11，增加的方式是 old*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。



GET

```
V get(Object key, int hash) {
      if (count != 0) { // read-volatile // ①
         HashEntry<K,V> e = getFirst(hash);
         while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
               V v = e.value;
               if (v != null)  // ② 注意这里
                  return v;
               return readValueUnderLock(e); // recheck
            }
            e = e.next;
         }
      }
      return null;
}
```

它也没有使用锁来同步，只是判断获取的entry的value是否为null，为null时才使用加锁的方式再次去获取。
这个实现很微妙，没有锁同步的话，靠什么保证同步呢？我们一步步分析。
第一步，先判断一下 count != 0；count变量表示segment中存在entry的个数。如果为0就不用找了。
假设这个时候恰好另一个线程put或者remove了这个segment中的一个entry，会不会导致两个线程看到的count值不一致呢？
看一下count变量的定义： transient volatile int count;
它使用了volatile来修改。我们前文说过，Java5之后，JMM实现了对volatile的保证：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。
所以，每次判断count变量的时候，即使恰好其他线程改变了segment也会体现出来。
第二步，获取到要该key所在segment中的索引地址，如果该地址有相同的hash对象，顺着链表一直比较下去找到该entry。当找到entry的时候，先做了一次比较： if(v != null) 我们用红色注释的地方。

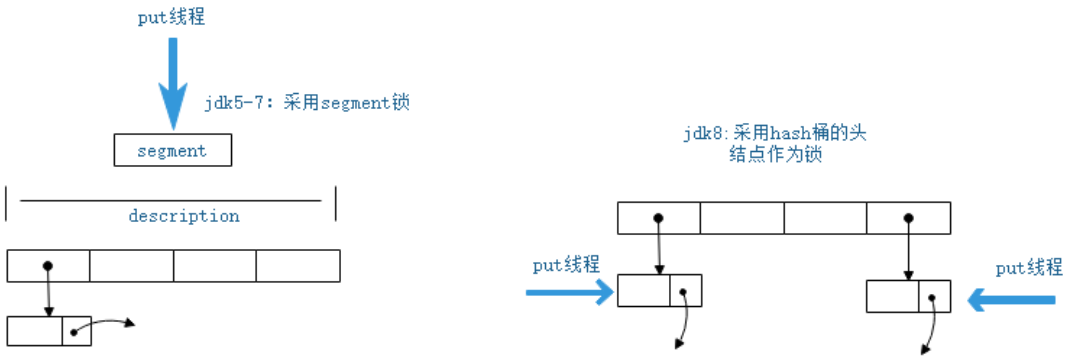
考虑一下，如果这个时候，另一个线程恰好新增/删除了entry，或者改变了entry的value
1) 在get代码的①和②之间，另一个线程新增了一个entry
因为每个HashEntry中的next也是final的，没法对链表最后一个元素增加一个后续entry所以新增一个entry的实现方式只能通过头结点来插入了。
newEntry对象是通过 new HashEntry(K k , V v, HashEntry next) 来创建的。如果另一个线程刚好new 这个对象时，当前线程来get它。因为没有同步，就可能会出现当前线程得到的newEntry对象是一个没有完全构造好的对象引用。
回想一下我们之前讨论的DCL的问题，这里也一样，没有锁同步的话，new 一个对象对于多线程看到这个对象的状态是没有保障的，这里同样有可能一个线程new这个对象的时候还没有执行完构造函数就被另一个线程得到这个对象引用。
所以才需要判断一下：if (v != null) 如果确实是一个不完整的对象，则使用锁的方式再次get一次。

2) 在get代码的①和②之间，另一个线程修改了一个entry的value
value是用volitale修饰的，可以保证读取时获取到的是修改后的值。
3) 在get代码的①之后，另一个线程删除了一个entry
假设我们的链表元素是：e1-> e2 -> e3 -> e4 我们要删除 e3这个entry，因为HashEntry中next的不可变，所以我们无法直接把e2的next指向e4，而是将要删除的节点之前的节点复制一份，形成新的链表。

PUT 操作一上来就锁定了整个segment，这当然是为了并发的安全，修改数据是不能并发进行的，必须得有个判断是否超限的语句以确保容量不足时能够 rehash，而比较难懂的是这句int index = hash & (tab.length - 1)，原来**segment里面才是真正的hashtable，即每个**

segment是一个传统意义上的hashtable,如上图，从两者的结构就可以看出区别，这里就是找出需要的entry在table的哪一个位置，之后得到的entry就是这个链的第一个节点，如果e!=null，说明找到了，这是就要替换节点的值（onlyIfAbsent == false），否则，我们需要new一个entry，它的后继是first，而让tab[index]指向它，什么意思呢？实际上就是将这个新entry 插入到链头

旧版本的一个segment锁，保护了多个hash桶，而jdk8版本的一个锁只保护一个hash桶，由于锁的粒度变小了，写操作的并发性得到了极大的提升。



1 **更多的扩容线程**

扩容时，需要锁的保护。因此：**旧版本最多可以同时扩容的线程数是segment锁的个数。**

而jdk8的版本，理论上最多可以同时扩容的线程数是：hash桶的个数（table数组的长度）。

但是为了防止扩容线程过多，ConcurrentHashMap规定了扩容线程每次最少迁移16个hash桶，因此**jdk8的版本实际上最多可以同时扩容的线程数是：hash桶的个数/16**

2 **扩容期间，依然保证较高的并发度**

旧版本的segment锁，锁定范围太大，导致扩容期间，写并发度，严重下降。

而新版本的采用更加细粒度的hash桶级别锁，扩容期间，依然可以保证写操作的并发度。

我们思考一个问题：

旧版本对table数组元素的读写，都是在segment锁保护的情况下进行的，因此不会内存可见性问题。而**jdk8的实现中，锁的粒度是hash桶,因此对table数组元素的读写，大部分都是在没有锁的保护下进行的**，那么该如何保证table数组元素的内存可见性？

要解决这个问题，我们要了解如下2点：

1. volatile的happens-before规则：对一个*volatile*变量的写一定可见（*happens-before)*于随后对它的读

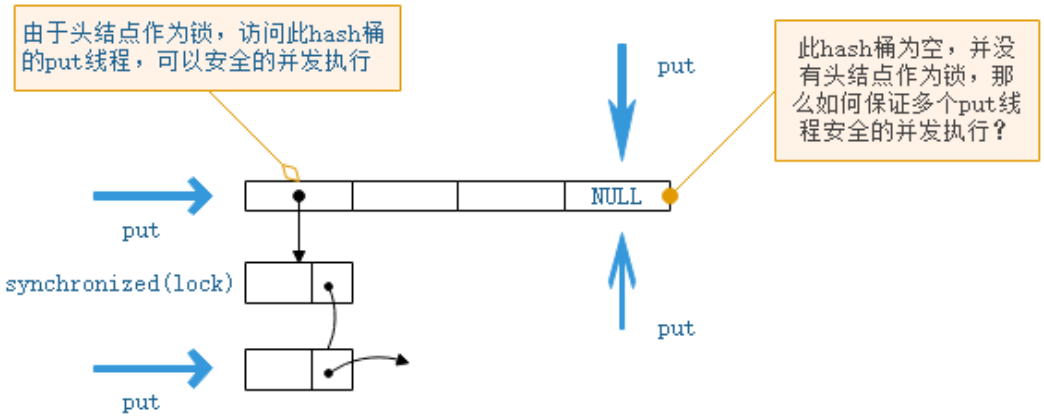
2. CAS同时具有volatile读和volatile写的内存语义

PUT方法

1. 由于锁的粒度是hash桶，多个put线程只有在请求同一个hash桶时，才会被阻塞。请求不同hash桶的put线程，可以并发执行。

2. put线程，请求的hash桶为空时，采用for循环+CAS的方式无锁插入。

如上图所示，在不加锁的情况下：线程A成功执行casTabAt操作后，随后的线程B可以通过tabAt方法立刻看到table[i]的改变。原因如下：线程A的casTabAt操作，具有volatile读写相同的内存语义，根据volatile的happens-before规则：线程A的casTabAt操作，一定对线程B的tabAt操作可见。
我们再思考如下问题:
ConcurrentHashMap中的锁是hash桶的头结点，那么当多个put线程访问头结点为空的hash桶时，在没有互斥锁保护的情况下，多个put线程都会尝试将元素插入头结点，此时如何确保并发安全呢?



然后，我们看下ConcurrentHashMap的put方法是如何通过CAS确保线程安全的:
假设此时有2个put线程,都发现此时桶为空，线程一执行casTabAt(tab,i,null,node1),此时tab[i]等于预期值null,因此会插入node1。随后线程二执行casTabAt(tba,i,null,node2),此时tab[i]不等于预期值null，插入失败。然后线程二会回到for循环开始处，重新获取tab[i]作为预期值，重复上述逻辑。

get方法

*get*方法同样利用了*volatile*特性，实现了无锁读。

查找value的过程如下：

1. 根据key定位hash桶，通过**tabAt的volatile读**，获取hash桶的头结点。
2. 通过头结点Node的**volatile属性next**，遍历Node链表
3. 找到目标node后，读取Node的**volatile属性val**