

目录

第1章 基础

01 开篇词：为什么学习本专栏

02 String、Long 源码解析和面试题

03 Java 常用关键字理解

04 Arrays、Collections、Objects 常用方法源码解析

第2章 集合

05 ArrayList 源码解析和设计思路

06 LinkedList 源码解析

07 List 源码会问哪些面试题

08 HashMap 源码解析

09 TreeMap 和 LinkedHashMap 核心源码解析

10 Map源码会问哪些面试题

11 HashSet、TreeSet 源码解析

12 彰显细节：看集合源码对我们实际工作的帮助和应用

13 差异对比：集合在 Java 7 和 8 有何不同和改进

14 简化工作：Guava Lists Maps 实际工作运用和源码

第3章 并发集合类

15 CopyOnWriteArrayList 源码解析和设计思路

16 ConcurrentHashMap 源码解析和设计思路

17 并发 List、Map源码面试题 最近阅读

18 场景集合：并发 List、Map的应用

17 并发 List、Map源码面试题

更新时间：2019-10-09 05:44:11



“梦想只要能持久，就能成为现实。我们不就是生活在梦想中的吗？”
——丁尼生

果断更，请联系QQ/微信6426006

并发 List 和 Map 是技术面时常问的问题，问的问题也都比较深入，有很多问题都是面试官自创的，市面上找不到，所以说通过背题的方式，这一关大部分是过不了的，只有我们真正理解了 API 内部的实现，阅读过源码，才能自如应对各种类型的面试题，接着我们来看一下并发 List、Map 源码相关的面试题集。

1 CopyOnWriteArrayList 相关

1.1 和 ArrayList 相比有哪些相同点和不同点？

答：相同点：底层的数据结构是相同的，都是数组的数据结构，提供出来的 API 都是对数组结构进行操作，让我们更好的使用。

不同点：后者是线程安全的，在多线程环境下使用，无需加锁，可直接使用。

1.2 CopyOnWriteArrayList 通过哪些手段实现了线程安全？

答：主要有：1. 数组容器被 volatile 关键字修饰，保证了数组内存地址被任意线程修改后，都会通知到其他线程；

2. 对数组的所有修改操作，都进行了加锁，保证了同一时刻，只能有一个线程对数组进行修改，比如我在 add 时，就无法 remove；

3. 修改过程中对原数组进行了复制，是在新数组上进行修改的，修改过程中，不会对原数组产生任何影响。

1.3 在 add 方法中，对数组进行加锁后，不是已经是线程安全了么，为什么还需要对老数组进行拷贝？

答：的确，对数组进行加锁后，能够保证同一时刻，只有一个线程能对数组进行 add，在同单核 CPU 下的多线程环境下肯定没有问题，但我们现在的机器都是多核 CPU，如果我们不通过复制拷贝新建数组，修改原数组容器的内存地址的话，是无法触发 volatile 可见性效果的，那么其他 CPU 下的线程就无法感知数组原来已经被修改了，就会引发多核 CPU 下的线程安全问题。

假设我们不复制拷贝，而是在原来数组上直接修改值，数组的内存地址就不会变，而数组被 volatile 修饰时，必须当数组的内存地址变更时，才能及时的通知到其他线程，内存地址不变，仅仅是数组元素值发生变化时，是无法把数组元素值发生变动的事实，通知到其它线程的。

1.4 对老数组进行拷贝，会有性能损耗，我们平时使用需要注意什么么？

答：主要有：

1. 在批量操作时，尽量使用 addAll、removeAll 方法，而不要在循环里面使用 add、remove 方法，主要是因为 for 循环里面使用 add、remove 的方式，在每次操作时，都会进行一次数组的拷贝(甚至多次)，非常耗性能，而 addAll、removeAll 方法底层做了优化，整个操作只会进行一次数组拷贝，由此可见，当批量操作的数据越多时，批量方法的高性能体现的越明显。

1.5 为什么 CopyOnWriteArrayList 迭代过程中，数组结构变动，不会抛出 ConcurrentModificationException？

答：主要是因为 CopyOnWriteArrayList 每次操作时，都会产生新的数组，而迭代时，持有的仍然是老数组的引用，所以我们说的数组结构变动，是用新数组替换了老数组，老数组的结构并没有发生变化，所以不会抛出异常了。

1.6 插入的数据正好在 List 的中间，请问两种 List 分别拷贝数组几次？为什么？

答：ArrayList 只需拷贝一次，假设插入的位置是 2，只需要把位置 2（包含 2）后面的数据都往后移动一位即可，所以拷贝一次。

CopyOnWriteArrayList 拷贝两次，因为 CopyOnWriteArrayList 多了把老数组的数据拷贝到新数组上这一步，可能的同学会想到这种方式：先把老数组拷贝到新数组，再把 2 后面的数据往后移动一位，这的确是一种拷贝的方式，但 CopyOnWriteArrayList 底层实现更加灵活，而是：把老数组 0 到 2 的数据拷贝到新数组上，预留出新数组 2 的位置，再把老数组 3 ~ 最后的数据拷贝到新数组上，这种拷贝方式可以减少我们拷贝的数据，虽然是两次拷贝，但拷贝的数据却仍然是老数组的大小，设计的非常巧妙。

2 ConcurrentHashMap 相关

2.1ConcurrentHashMap 和 HashMap 的相同点和不同点

答：相同点：1. 都是数组 + 链表 + 红黑树的数据结构，所以基本操作的思想相同；

2. 都实现了 Map 接口，继承了 AbstractMap 抽象类，所以两者的方法大多都是相似的，可以互相切换。

2. 数据结构上，ConcurrentHashMap 多了转移节点，主要用于保证扩容时的线程安全。

目录

2.2 ConcurrentHashMap 通过哪些手段保证了线程安全。

答：主要有以下几点：

1. 储存 Map 数据的数组被 volatile 关键字修饰，一旦被修改，立马就能通知其他线程，因为是数组，所以需要改变其内存值，才能真正的发挥出 volatile 的可见特性；
2. put 时，如果计算出来的数组下标索引没有值的话，采用无限 for 循环 + CAS 算法，来保证一定可以新增成功，又不会覆盖其他线程 put 进去的值；
3. 如果 put 的节点正好在扩容，会等待扩容完成之后，再进行 put，保证了在扩容时，老数组的值不会发生变化；
4. 对数组的槽点进行操作时，会先锁住槽点，保证只有当前线程才能对槽点上的链表或红黑树进行操作；
5. 红黑树旋转时，会锁住根节点，保证旋转时的线程安全。

2.3 描述一下 CAS 算法在 ConcurrentHashMap 中的应用？

答：CAS 其实是一种乐观锁，一般有三个值，分别为：赋值对象，原值，新值，在执行的时候，会先判断内存中的值是否和原值相等，相等的话把新值赋值给对象，否则赋值失败，整个过程都是原子性操作，没有线程安全问题。

ConcurrentHashMap 的 put 方法中，有使用到 CAS，是结合无限 for 循环一起使用的，步骤如下：

1. 计算出数组索引下标，拿出下标对应的原值；
2. CAS 覆盖当前下标的值，赋值时，如果发现内存值和 1 拿出来原值相等，执行赋值，退出循环，否则不赋值，转到 3；
3. 进行下一次 for 循环，重复执行 1，2，直到成功为止。

可以看到这样做的好处，第一是不会盲目的覆盖原值，第二是一定可以赋值成功。

2.4 ConcurrentHashMap 是如何发现当前槽点正在扩容的。

答：ConcurrentHashMap 新增了一个节点类型，叫做转移节点，当我们发现当前槽点是转移节点时（转移节点的 hash 值是 -1），即表示 Map 正在进行扩容。

2.5 发现槽点正在扩容时，put 操作会怎么办？

答：无限 for 循环，或者走到扩容方法中去，帮助扩容，一直等待扩容完成之后，再执行 put 操作。

2.6 两种 Map 扩容时，有啥区别？

答：区别很大，HashMap 是直接老数据上面进行扩容，多线程环境下，会有线程安全的问题，而 ConcurrentHashMap 就不太一样，扩容过程是这样的：

1. 从数组的队尾开始拷贝；
2. 拷贝数组的槽点时，先把原数组槽点锁住，拷贝成功到新数组时，把原数组槽点赋值为转移节点；
3. 从数组的尾部拷贝到头部，每拷贝成功一次，就把原数组的槽点设置成转移节点；

间里米说，通过扩容时槽点加锁，和发现槽点止住扩容就等待的策略，保证了 ConcurrentHashMap 可以慢慢一个一个槽点的转移，保证了扩容时的线程安全，转移节点比较重要，平时问的人也比较多。

2.7 ConcurrentHashMap 在 Java 7 和 8 中关于线程安全的做法有啥不同？

答：非常不一样，拿 put 方法为例，Java 7 的做法是：

1. 把数组进行分段，找到当前 key 对应的是那一段；
2. 将当前段锁住，然后再根据 hash 寻找对应的值，进行赋值操作。

Java 7 的做法比较简单，缺点也很明显，就是当我们需要 put 数据时，我们会锁住改该数据对应的某一段，这一段数据可能会有很多，比如我只想 put 一个值，锁住的却是一段数据，导致这一段的其他数据都不能进行写入操作，大大的降低了并发性的效率。Java 8 解决了这个问题，从锁住某一段，修改成锁住某一个槽点，提高了并发效率。

不仅仅是 put，删除也是，仅仅是锁住当前槽点，缩小了锁的范围，增大了效率。

3 总结

因为目前大多数公司都已经在使用 Java 8 了，所以大部分面试内容还是以 Java 8 的 API 为主，特别是 CopyOnWriteArrayList 和 ConcurrentHashMap 两个 API，文章毕竟篇幅有限，建议大家多多阅读剩余源码。

果断更，请联系QQ/微信6426006

ConcurrentHashMap 源码解析
设计思路

18 场景集合：并发 List、Map 的应用场景

精选留言 4

欢迎在这里发表留言，作者筛选后可公开显示

wt4446

槽点？啥意思

👍 0 回复

2019-11-30

文贺 回复 wt4446

在 JDK1.8 中表示数组中的某一个元素。

回复

7天前

慕仙6328494

老师, CopyOnWriteArrayList中你说volatile修饰数组是保证可见性,必须数组内存地址改变才触发, 而ConcurrentHashMap用volatile修饰数组怎么保证可见性呢,内存地址也不变啊

👍 0 回复

2019-11-23

目录	回复	2019-11-30 13:32:58
----	----	---------------------

weixin_慕工程5089940

老师你写的 HashMap 是直接在老数据上面进行扩容 这句话怎么理解啊？

👍 0 回复

2019-11-14

文贺 回复 weixin_慕工程5089940

这个是和 ConcurrentHashMap 的扩容相对而言的，HashMap 扩展是直接在 oldTable 上操作的，ConcurrentHashMap 则不同，ConcurrentHashMap 的如果操作的可以看看文章描述哈。

回复

2019-11-17 10:45:15

街边七号

2.3 `CAS其实是一种悲观锁`，此处是否笔误了。

👍 0 回复

2019-10-04

文贺 回复 街边七号

是的，练习编辑更正中，谢谢

回复

2019-10-08 19:15:05

果断更，请联系QQ/微信64260060

千学不如一看，千看不如一练