17 | 并发容器的使用：识别不同场景下最优容器

2019-06-27 刘超

你好，我是刘超。

在并发编程中，我们经常会用到容器。今天我要和你分享的话题就是：在不同场景下我们该如何选择最优容器。

并发场景下的Map容器

假设我们现在要给一个电商系统设计一个简单的统计商品销量TOP 10的功能。常规情况下，我们是用一个哈希表来存储商品和销量键值对，然后使用排序获得销量前十的商品。在这里，哈希表是实现该功能的关键。那么请思考一下，如果要你设计这个功能，你会使用哪个容器呢？

在07讲中，我曾详细讲过HashMap的实现原理，以及HashMap结构的各个优化细节。我说过HashMap的性能优越，经常被用来存储键值对。那么这里我们可以使用HashMap吗？

答案是不可以，我们切忌在并发场景下使用HashMap。因为在JDK1.7之前，在并发场景下使用HashMap会出现死循环，从而导致CPU使用率居高不下，而扩容是导致死循环的主要原因。虽然Java在JDK1.8中修复了HashMap扩容导致的死循环问题，但在高并发场景下，依然会有数据丢失以及不准确的情况出现。

这时为了保证容器的线程安全，Java实现了Hashtable、ConcurrentHashMap以及ConcurrentSkipListMap等Map容器。

Hashtable、ConcurrentHashMap是基于HashMap实现的，对于小数据量的存取比较有优势。

ConcurrentSkipListMap是基于TreeMap的设计原理实现的，略有不同的是前者基于跳表实现，后者基于红黑树实现，ConcurrentSkipListMap的特点是存取平均时间复杂度是O（log（n）），适用于大数据量存取的场景，最常见的是基于跳跃表实现的数据量比较大的缓存。

回归到开始的案例再看一下，如果这个电商系统的商品总量不是特别大的话，我们可以用Hashtable或ConcurrentHashMap来实现哈希表的功能。

Hashtable ⅋ ConcurrentHashMap

更精准的话，我们可以进一步对比看看以上两种容器。

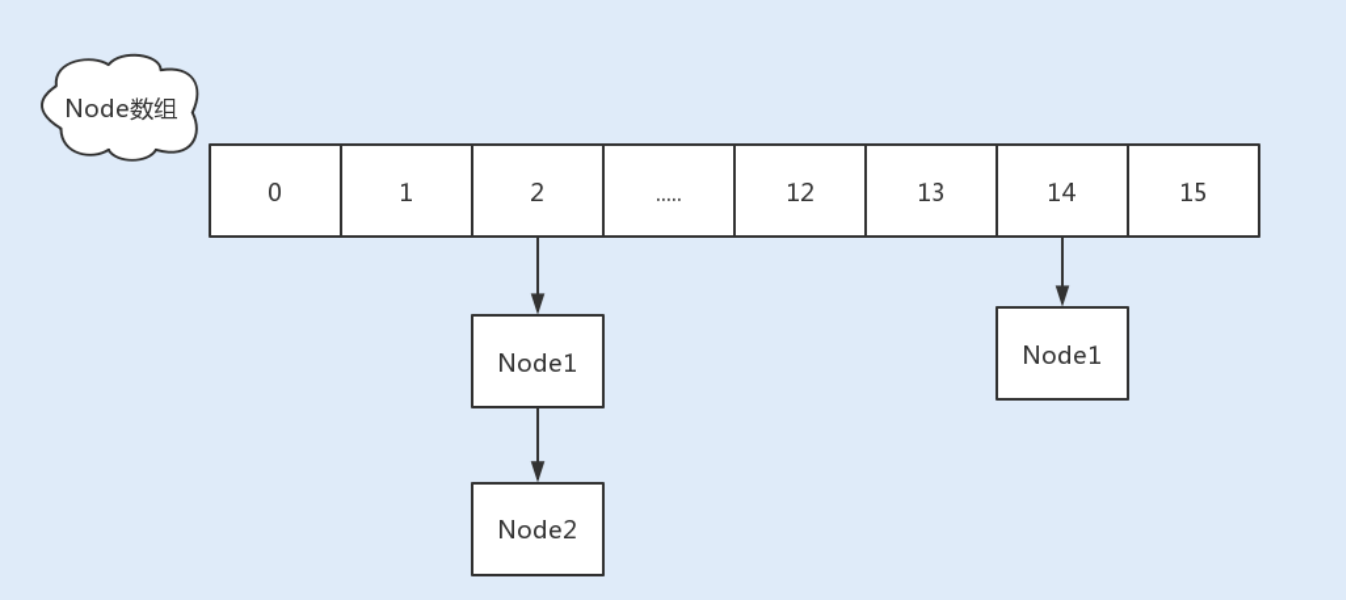
在数据不断地写入和删除，且不存在数据量累积以及数据排序的场景下，我们可以选用Hashtable或ConcurrentHashMap。

Hashtable使用Synchronized同步锁修饰了put、get、remove等方法，因此在高并发场景下，读写操作都会存在大量锁竞争，给系统带来性能开销。

相比Hashtable，ConcurrentHashMap在保证线程安全的基础上兼具了更好的并发性能。在JDK1.7中，ConcurrentHashMap就使用了分段锁Segment减小了锁粒度，最终优化了锁的并发操作。

到了JDK1.8，ConcurrentHashMap做了大量的改动，摒弃了Segment的概念。由于Synchronized锁在Java6之后的性能已经得到了很大的提升，所以在JDK1.8中，Java重新启用了Synchronized同步锁，通过Synchronized实现HashEntry作为锁粒度。这种改动将数据结构变得更加简单了，操作也更加清晰流畅。

与JDK1.7的put方法一样，JDK1.8在添加元素时，在没有哈希冲突的情况下，会使用CAS进行添加元素操作；如果有冲突，则通过Synchronized将链表锁定，再执行接下来的操作。



综上所述，我们在设计销量TOP10功能时，首选ConcurrentHashMap。

但要注意一点，虽然ConcurrentHashMap的整体性能要优于Hashtable，但在某些场景中，ConcurrentHashMap依然不能代替Hashtable。例如，在强一致的场景中ConcurrentHashMap就不适用，原因是ConcurrentHashMap中的get、size等方法没有用到锁，ConcurrentHashMap是弱一致性的，因此有可能会导致某次读无法马上获取到写入的数据。

ConcurrentHashMap ⅋ ConcurrentSkipListMap

我们再看一个案例，我上家公司的操作系统中有这样一个功能，提醒用户手机卡实时流量不足。主要的流程是服务端先通过虚拟运营商同步用户实时流量，再通过手机端定时触发查询功能，如果流量不足，就弹出系统通知。

该功能的特点是用户量大，并发量高，写入多于查询操作。这时我们就需要设计一个缓存，用来存放这些用户以及对应的流量键值对信息。那么假设让你来实现一个简单的缓存，你会怎么设计呢？

你可能会考虑使用ConcurrentHashMap容器，但我在07讲中说过，该容器在数据量比较大的时候，链表会转换为红黑树。红黑树在并发情况下，删除和插入过程中有个平衡的过程，会牵涉到大量节点，因此竞争锁资源的代价相对比较高。

而跳跃表的操作针对局部，需要锁住的节点少，因此在并发场景下的性能会更好一些。你可能会问了，在非线程安全的Map容器中，我并没有看到基于跳跃表实现的SkipListMap呀？这是因为在非线程安全的Map容器中，基于红黑树实现的TreeMap在单线程中的性能表现得并不比跳跃表差。

因此就实现了在非线程安全的Map容器中，用TreeMap容器来存取大数据；在线程安全的Map容器中，用SkipListMap容器来存取大数据。

那么ConcurrentSkipListMap是如何使用跳跃表来提升容器存取大数据的性能呢？我们先来了解下跳跃表的实现原理。

什么是跳跃表

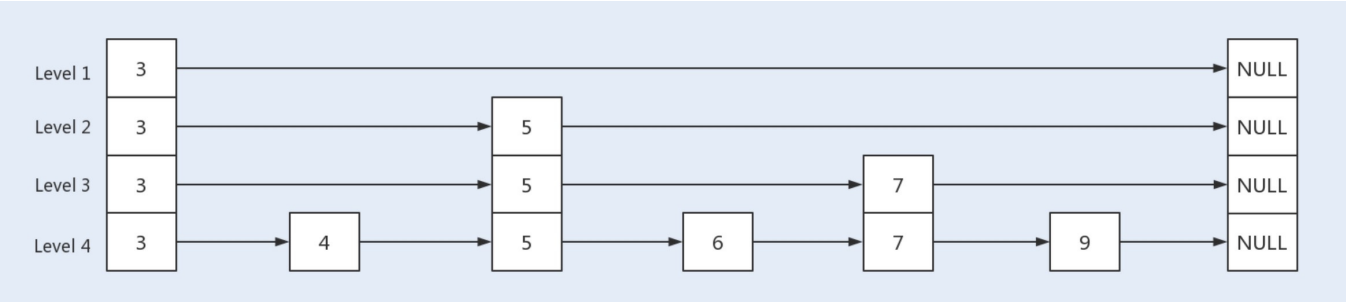
跳跃表是基于链表扩展实现的一种特殊链表，类似于树的实现，跳跃表不仅实现了横向链表，还实现了垂直方向的分层索引。

一个跳跃表由若干层链表组成，每一层都实现了一个有序链表索引，只有最底层包含了所有数据，每一层由下往上依次通过一个指针指向上层相同值的元素，每层数据依次减少，等到了最顶层就只会保留部分数据了。

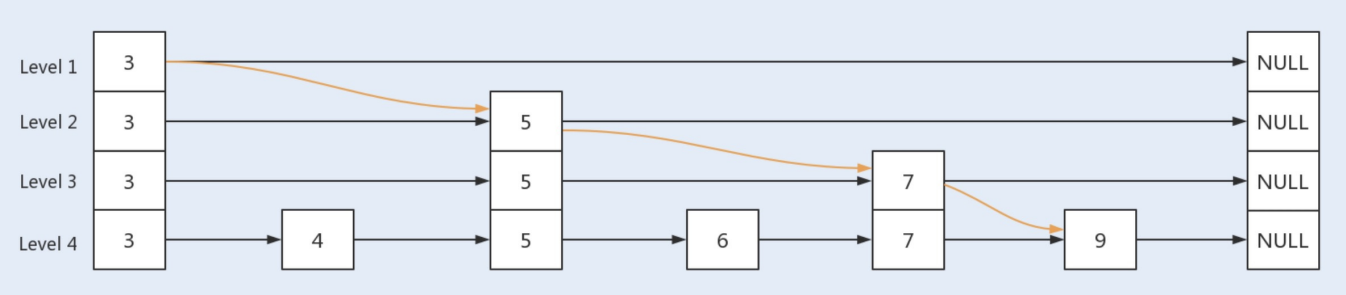
跳跃表的这种结构，是利用了空间换时间的方法来提高了查询效率。程序总是从最顶层开始查询

访问，通过判断元素值来缩小查询范围。我们可以通过以下几张图来了解下跳跃表的具体实现原理。

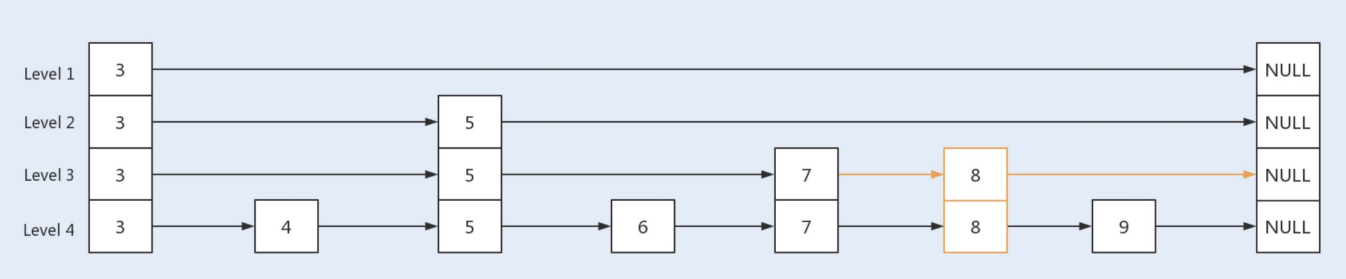
首先是一个初始化的跳跃表：



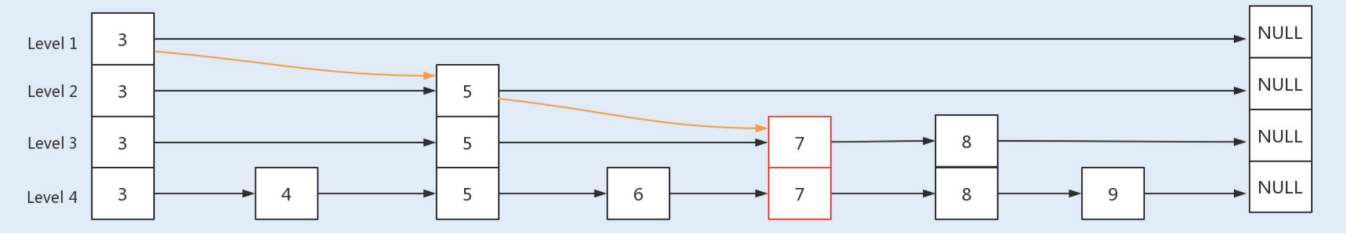
当查询key值为9的节点时，此时查询路径为：



当新增一个key值为8的节点时，首先新增一个节点到最底层的链表中，根据概率算出level值，再根据level值新建索引层，最后链接索引层的新节点。新增节点和链接索引都是基于CAS操作实现。



当删除一个key值为7的结点时，首先找到待删除结点，将其value值设置为null；之后再向待删除结点的next位置新增一个标记结点，以便减少并发冲突；然后让待删结点的前驱节点直接越过本身指向的待删结点，直接指向后继结点，中间要被删除的结点最终将会被JVM垃圾回收处理掉；最后判断此次删除后是否导致某一索引层没有其它节点了，并视情况删除该层索引 。



通过以上两个案例，我想你应该清楚了Hashtable、ConcurrentHashMap以及

ConcurrentSkipListMap这三种容器的适用场景了。

如果对数据有强一致要求，则需使用Hashtable；在大部分场景通常都是弱一致性的情况下，使用ConcurrentHashMap即可；如果数据量在千万级别，且存在大量增删改操作，则可以考虑使用ConcurrentSkipListMap。

并发场景下的List容器

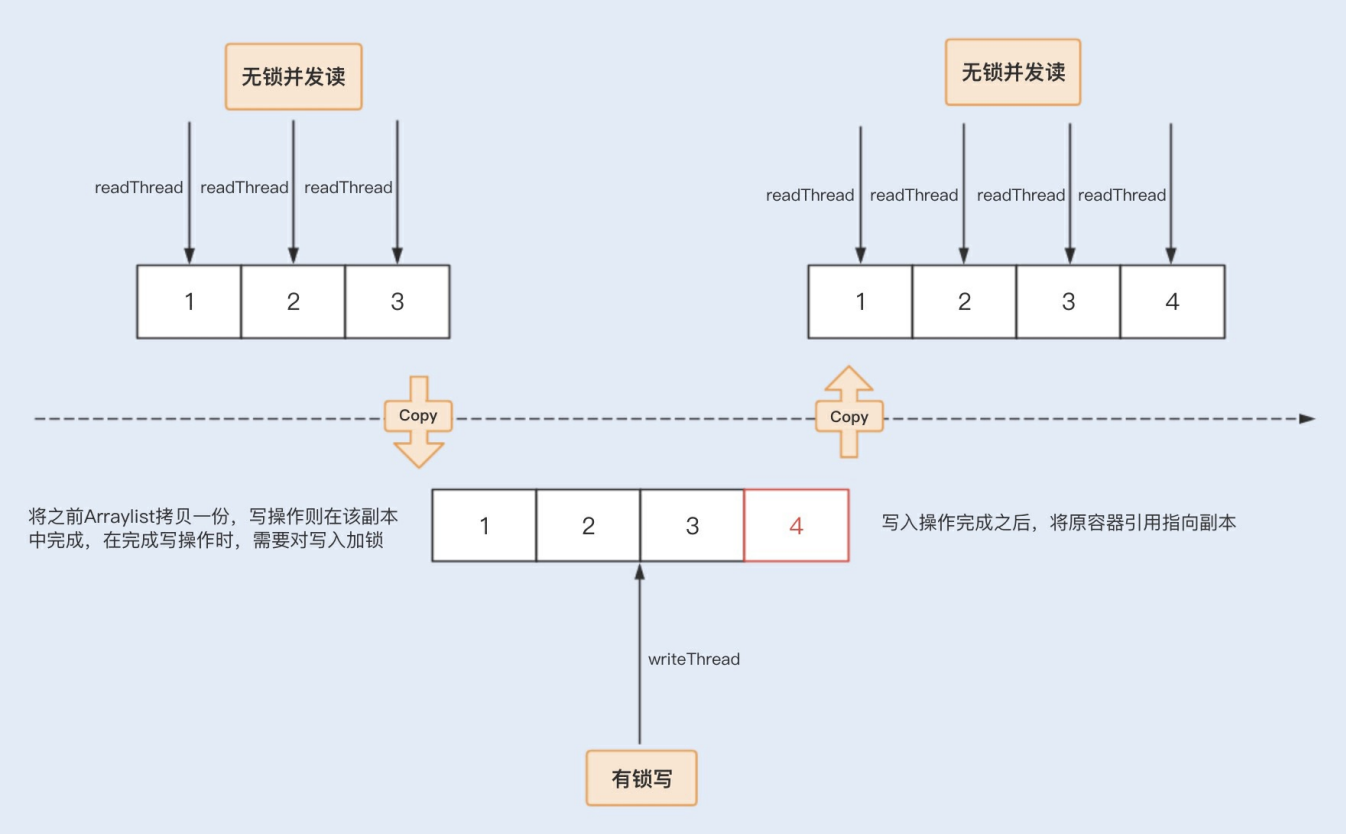
下面我们再来看一个实际生产环境中的案例。在大部分互联网产品中，都会设置一份黑名单。例如，在电商系统中，系统可能会将一些频繁参与抢购却放弃付款的用户放入到黑名单列表。想想这个时候你又会使用哪个容器呢？

首先用户黑名单的数据量并不会很大，但在抢购中需要查询该容器，快速获取到该用户是否存在于黑名单中。其次用户ID是整数类型，因此我们可以考虑使用数组来存储。那么ArrayList是否是你第一时间想到的呢？

我讲过ArrayList是非线程安全容器，在并发场景下使用很可能会导致线程安全问题。这时，我们就可以考虑使用Java在并发编程中提供的线程安全数组，包括Vector和CopyOnWriteArrayList。

Vector也是基于Synchronized同步锁实现的线程安全，Synchronized关键字几乎修饰了所有对外暴露的方法，所以在读远大于写的操作场景中，Vector将会发生大量锁竞争，从而给系统带来性能开销。

相比之下，CopyOnWriteArrayList是java.util.concurrent包提供的方法，它实现了读操作无锁，写操作则通过操作底层数组的新副本来实现，是一种读写分离的并发策略。我们可以通过以下图示来了解下CopyOnWriteArrayList的具体实现原理。



回到案例中，我们知道黑名单是一个读远大于写的操作业务，我们可以固定在某一个业务比较空闲的时间点来更新名单。

这种场景对写入数据的实时获取并没有要求，因此我们只需要保证最终能获取到写入数组中的用户ID就可以了，而CopyOnWriteArrayList这种并发数组容器无疑是最适合这类场景的了。

总结

在并发编程中，我们经常会使用容器来存储数据或对象。Java在JDK1.1到JDK1.8这个漫长的发展过程中，依据场景的变化实现了同类型的多种容器。我将今天的主要内容为你总结了一张表格，希望能对你有所帮助，也欢迎留言补充。

分类	名称	特性	适用场景
Map并发容器	Hashtable	强一致性	对数据强一致性有要求的场景
	ConcurrentHashMap	基于数据+链表+红黑树实现，CAS+Synchronized实现原子性，部分操作属于无锁操作，具有弱一致性	存取数据量小，查询操作频繁，且对数据没有强一致要求的高并发场景
	ConcurrentSkipListMap	基于跳跃表实现，具有弱一致性	存取数据量大，增删改查操作频繁，且对数据没有强一致要求的高并发场景
List并发容器	Vector	具有强一致性	对数据强一致性有要求的场景
	CopyOnWriteArrayList	基于复制副本用于有锁写操作，操作完成之后，Array容器重新指向新的副本容器，具有弱一致性	读远大于写操作的场景

思考题

在抢购类系统中，我们经常会使用队列来实现抢购的排队等待，如果要你来选择或者设计一个队列，你会怎么考虑呢？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。

精选留言

undifined

4

抢购的过程中存在并发操作，所以需要用线程安全的容器，同时，抢购的用户会很多，应当使用链表的数据结构，这种场景往往是写多读少，还需要排队，所以 ConcurrentLinkedQueue应该是最合适的

2019-06-27

作者回复

对的，ConcurrentLinkedQueue是基于CAS乐观锁来实现线程安全。ConcurrentLinkedQueue是无界的，所以使用的时候要特别注意内存溢出问题。

2019-06-27

东方奇骥

3

如果数据变动频繁，就不建议使用CopyOnWriteArrayList了，因为每次写都要拷贝一份，代价太大。老师，怎么直观理解强一致性和弱一致性？之前一直觉得ConcurrentHashMap就是用来代替HashTable的，因为HashTable并发时因为同步锁性能差。

2019-06-27

作者回复

对的，CopyOnWriteArrayList只适合偶尔一两次数据更改的操作。我们很多缓存数据往往是在深夜在没有读操作时，进行修改。这种场景适合使用CopyOnWriteArrayList。

我们先来理解下happens-before规则中，对锁的规则：
一个unLock操作先行发生于后面对同一个锁的lock操作；

也就是说，ConcurrentHashMap中的get如果有锁操作，在put操作之后，get操作是一定能拿到put后的数据；而实际上get操作时没有锁的，也就是说下面这种情况：

```
void func(){
map.put(key1,value1);
map.get(key1);
.
.
//use key1 value to do something
}
```

此时，get获取值的可能不是put修改的值，而此时get没有获取到真正要获取的值，此时就是弱一致了。

2019-06-28

大雁小鱼 2

老师，ConcurrentHashMap为啥是弱一致性的？

2019-06-27

作者回复

因为ConcurrentHashMap有些方法是没有锁的，例如get 方法。假设A修改了数据，而B后于A一瞬间去获取数据，有可能拿到的数据是A修改之前的数据。

还有 clear foreach方法在操作时，都有可能存在数据不确定性。

2019-06-28

QQ怪 1

麻烦老师加餐出个queue并发相关的文章，感激不尽！！！

2019-06-27

WL 1

请问一下老师两个问题:
1. 为什么在无锁是红黑树和跳表的性能差不多呢, 红黑树再平衡的操作会不会更复杂一些.
2. 从本篇文章看好像ConcurrentSkipListMap的性能比ConcurrentHashMap性能要好, 那为啥平时还是用后者的人更多呢, 我想很定是后者相对前者也有一定的优势吧, 但我自己没想出来, 老师能不能指点一下是啥优势.

2019-06-27

作者回复

1、两者的平均查询复杂度都是O(logn)，所以查询性能差不多。而在新增和删除操作，红黑树有平衡操作，但跳跃表也有建立索引层操作。跳跃表的结构简单易懂。

2、这里是基于数据量比较大（例如千万级别）且写入操作多的情况下，ConcurrentSkipListMap性能要比ConcurrentHashMap好一些，并不是在任何情况下都要优于ConcurrentHashMap的。

2019-06-28

Liam 1

老师，我有2个问题:

1 top 10 问题涉及到排序, 我感觉用优先级队列或带排序功能的ConcurrentSkipListMap更合适？ConcurrentHashMap不支持排序吧

2 CopyOnWrite的list为什么还要加锁呢，副本不是线程独享的吗？

2019-06-27

作者回复

ConcurrentSkipListMap只是key值的升排序，并没有对value进行排序；

CopyOnWrite在副本写时，是需要加锁的。

2019-06-28

李 0

在目前线上机器都是多台部署的，想问下老师，如上的数据结构如何选择，或者用不到如上的选择，都彩玉第3方存储，比如redis实现

2019-06-30

WL 0

赞成讲下那几个blockingQueue

2019-06-29

晓杰 0

可以用ConcurrentLinkedQueue，优势如下：
1、抢购场景一般都是写多读少，该队列基于链表实现，所以新增和删除元素性能较高
2、写数据时通过cas操作，性能较高。
但是LinkedQueue有一个普遍存在的问题，就是该队列是无界的，需要控制容量，否则可能引起内存溢出

2019-06-28

作者回复

对的

2019-06-30

nightmare

0

抢购队列我觉得写大于读，链表比较合适，但是抢购数据确定，可以用linkedBlockQueue设置长度为抢购数据，就算分布式部署，为非也就是抢购数量乘以机器数据，ConcurentLinkedDequeue无界队列不合适

2019-06-28

左瞳

0

黑名单查询用户是否存在不是应该用HashSet吗？

2019-06-28

作者回复

如果是单线程写入，可以考虑使用HashSet。

2019-06-28

....

0

元素增减对每层链表的变化是怎么样的

2019-06-28

作者回复

在新增时，是通过随机算出level值，根据level新建垂直索引链。

```
//获取一个线程无关的随机数，占四个字节，32 个比特位
int rnd = ThreadLocalRandom.nextSecondarySeed();
//和 1000 0000 0000 0000 0000 0000 0000 0001 与
//如果等于 0，说明这个随机数最高位和最低位都为 0，这种概率很大
//如果不等于 0，那么将仅仅把新节点插入到最底层的链表中即可，不会往上层递归
if ((rnd & 0x80000001) == 0) {
int level = 1, max;
//用低位连续为 1 的个数作为 level 的值，也是一种概率策略
while (((rnd >>>= 1) & 1) != 0)
++level;
Index<K,V> idx = null;
HeadIndex<K,V> h = head;
//如果概率算得的 level 在当前跳表 level 范围内
//构建一个从 1 到 level 的纵列 index 结点引用
if (level <= (max = h.level)) {
for (int i = 1; i <= level; ++i)
idx = new Index<K,V>(z, idx, null);
}
//否则需要新增一个 level 层
else {
```

```
level = max + 1;
@SuppressWarnings("unchecked")
Index<K,V>[] idxs =(Index<K,V>[])new Index<?,?>[level+1];
for (int i = 1; i <= level; ++i)
idxs[i] = idx = new Index<K,V>(z, idx, null);
for (;;) {
h = head;
int oldLevel = h.level;
//level 肯定是比 oldLevel 大一的，如果小了说明其他线程更新过表了
if (level <= oldLevel)
break;
HeadIndex<K,V> newh = h;
Node<K,V> oldbase = h.node;
//正常情况下，循环只会执行一次，如果由于其他线程的并发操作导致 oldLevel 的值不稳定，
那么会执行多次循环体
for (int j = oldLevel+1; j <= level; ++j)
newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);
//更新头指针
if (casHead(h, newh)) {
h = newh;
idx = idxs[level = oldLevel];
break;
}
}
}
```

在删除时，会先删除底层的节点，最后通过扫描索引层是否失去了底层节点来回收掉索引层。

```
//判断此次删除后是否导致某一索引层没有其他节点了，并删除该层索引
if (head.right == null)
ryReduceLevel();
```

建议具体的可以深入到源码中查看。

2019-06-30

Geek_75b4cd

0

我也没太明白什么叫弱一致性

2019-06-27

作者回复

我们先来理解下happens-before规则中，对锁的规则：

一个unLock操作先行发生于后面对同一个锁的lock操作；

也就是说，ConcurrentHashMap中的get如果有锁操作，在put操作之后，get操作是一定能拿到put后的数据；而实际上get操作时没有锁的，也就是说下面这种情况：

```
void func(){
map.put(key1,value1);
map.get(key1);
.
.
//use key1 value to do something
}
```

此时，get获取值的可能不是put修改的值，而此时get没有获取到真正要获取的值，此时就是弱一致了。

2019-06-28

QQ怪 0

我想问下老师，concurrentHashMap是无序的，我不太明白是如何完成Top10这个需求的，麻烦解答

2019-06-27

作者回复

我们后面说了，在获取排名时，需要对concurrentHashMap进行排序。

2019-06-28

陆离 0

这一节要是有Queue就更好了，那几个blockingQueue还是很有意思的

2019-06-27

编辑回复

多留言哦～说不定在下一期的加餐中就惊现福利了！

2019-06-27

梁中华 0

ConcurrentHashmap和ConcurrentSkipListMap在功能上的主要区别是前者不排序，后者自动排序。并非数据量上的区别吧。控制好ConcurrentHashmap里元素key的hashcode值，尽量减少碰撞，在数据量比较大的情况下可能比ConcurrentSkipListMap的效率更好。

2019-06-27

作者回复

ConcurrentSkipListMap的key值是一个升序，而ConcurrentHashmap的默认没有排序。在查询速率来说，两者区别不大，区别在于存储大量数据情况下的写操作性能不一样。

2019-06-28

-W.LI-

老师好!为啥没有CopyOnWriteMap啊，ConpOnwriteArrayList。时间复杂度还是O(n)吧。

2019-06-27

Jxin

0

采用disruptor队列。一次性初始化所有元素。类似对象池；占位防止伪共享导致锁净增；元素存在相近的内存块

2019-06-27

-W.LI-

0

感觉跳跃表和B+树有点像

2019-06-27

大卫

0

老师，这一讲不是讲的线程池大小如何设置哈。

抢购场景后端服务是分布式部署的，是将库存分片放到不同服务器上吗，然后考虑容器的选择么

2019-06-27

作者回复

下一讲是线程池大小设置。分片到各个服务器上也是一种方案，这种方案问题就在可能有些服务库存没有了，其他服务上依然有库存。一般是基于分布式缓存来实现的，例如基于redis分布式锁来实现库存的扣减。

2019-06-27