

讲堂 > Java核心技术36讲 > 文章详情

第10讲 | 如何保证集合是线程安全的？ConcurrentHashMap如何实现高效地线程安全？

2018-05-26 杨晓峰



第10讲 | 如何保证集合是线程安全的？ConcurrentHashMap如何实现高效地线程安...

朗读人：黄洲君 10'46" | 4.93M

我在之前两讲介绍了 Java 集合框架的典型容器类，它们绝大部分都不是线程安全的，仅有的线程安全实现，比如 Vector、Stack，在性能方面也远不尽如人意。幸好 Java 语言提供了并发包（`java.util.concurrent`），为高度并发需求提供了更加全面的工具支持。

今天我要问你的问题是，**如何保证容器是线程安全的？ConcurrentHashMap 如何实现高效地线程安全？**

典型回答

Java 提供了不同层面的线程安全支持。在传统集合框架内部，除了 Hashtable 等同步容器，还提供了所谓的同步包装器（Synchronized Wrapper），我们可以调用 Collections 工具类提供的包装方法，来获取一个同步的包装容器（如 `Collections.synchronizedMap`），但是它们都是利用非常粗粒度的同步方式，在高并发情况下，性能比较低下。

另外，更加普遍的选择是利用并发包提供的线程安全容器类，它提供了：

- 各种并发容器，比如 ConcurrentHashMap、CopyOnWriteArrayList。
- 各种线程安全队列（Queue/Deque），如 ArrayBlockingQueue、SynchronousQueue。
- 各种有序容器的线程安全版本等。

具体保证线程安全的方式，包括有从简单的 synchronize 方式，到基于更加精细化的，比如基于分离锁实现的 ConcurrentHashMap 等并发实现等。具体选择要看开发的场景需求，总体来说，并发包内提供的容器通用场景，远优于早期的简单同步实现。

考点分析

谈到线程安全和并发，可以说是 Java 面试中必考的考点，我上面给出的回答是一个相对宽泛的总结，而且 ConcurrentHashMap 等并发容器实现也在不断演进，不能一概而论。

如果要深入思考并回答这个问题及其扩展方面，至少需要：

- 理解基本的线程安全工具。
- 理解传统集合框架并发编程中 Map 存在的问题，清楚简单同步方式的不足。
- 梳理并发包内，尤其是 ConcurrentHashMap 采取了哪些方法来提高并发表现。
- 最好能够掌握 ConcurrentHashMap 自身的演进，目前的很多分析资料还是基于其早期版本。

今天我主要是延续专栏之前两讲的内容，重点解读经常被同时考察的 HashMap 和 ConcurrentHashMap。今天这一讲并不是对并发方面的全面梳理，毕竟这也不是专栏一讲可以介绍完整的，算是个开胃菜吧，类似 CAS 等更加底层的机制，后面会在 Java 进阶模块中的并发主题有更加系统的介绍。

知识扩展


1. 为什么需要 ConcurrentHashMap？

Hashtable 本身比较低效，因为它的实现基本就是将 put、get、size 等各种方法加上“synchronized”。简单来说，这就导致了所有并发操作都要竞争同一把锁，一个线程在进行同步操作时，其他线程只能等待，大大降低了并发操作的效率。

前面已经提过 HashMap 不是线程安全的，并发情况会导致类似 CPU 占用 100% 等一些问题，那么能不能利用 Collections 提供的同步包装器来解决问题呢？

看看下面的代码片段，我们发现同步包装器只是利用输入 Map 构造了另一个同步版本，所有操作虽然不再声明成为 synchronized 方法，但是还是利用了“this”作为互斥的 mutex，没有真

正意义上的改进！

 复制代码

```
1 private static class SynchronizedMap<K,V>
2     implements Map<K,V>, Serializable {
3     private final Map<K,V> m;        // Backing Map
4     final Object      mutex;         // Object on which to synchronize
5     // ...
6     public int size() {
7         synchronized (mutex) {return m.size();}
8     }
9     // ...
10 }
11
```

所以，Hashtable 或者同步包装版本，都只是适合在非高度并发的场景下。

2.ConcurrentHashMap 分析

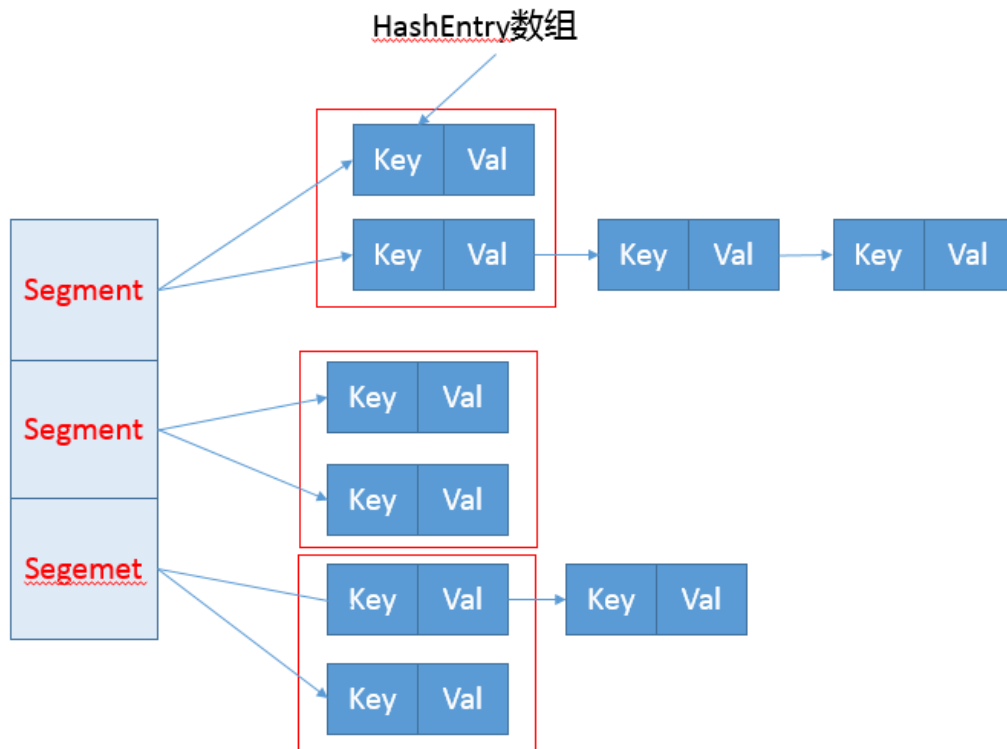
我们再来看看 ConcurrentHashMap 是如何设计实现的，为什么它能大大提高并发效率。

首先，我这里强调，ConcurrentHashMap 的设计实现其实一直在演化，比如在 Java 8 中就发生了非常大的变化（Java 7 其实也有不少更新），所以，我这里将比较分析结构、实现机制等方面，对比不同版本的主要区别。

早期 ConcurrentHashMap，其实现是基于：

- 分离锁，也就是将内部进行分段（Segment），里面则是 HashEntry 的数组，和 HashMap 类似，哈希相同的条目也是以链表形式存放。
- HashEntry 内部使用 volatile 的 value 字段来保证可见性，也利用了不可变对象的机制以改进利用 Unsafe 提供的底层能力，比如 volatile access，去直接完成部分操作，以最优化性能，毕竟 Unsafe 中的很多操作都是 JVM intrinsic 优化过的。

你可以参考下面这个早期 ConcurrentHashMap 内部结构的示意图，其核心是利用分段设计，在进行并发操作的时候，只需要锁定相应段，这样就有效避免了类似 Hashtable 整体同步的问题，大大提高了性能。



在构造的时候，Segment 的数量由所谓的 concurrencyLevel 决定，默认是 16，也可以在相应构造函数直接指定。注意，Java 需要它是 2 的幂数值，如果输入是类似 15 这种非幂值，会被自动调整到 16 之类 2 的幂数值。

具体情况，我们一起来看看一些 Map 基本操作的[源码](#)，这是 JDK 7 比较新的 get 代码。针对具体的优化部分，为方便理解，我直接注释在代码段里，get 操作需要保证的是可见性，所以并没有什么同步逻辑。


[复制代码](#)

```

1 public V get(Object key) {
2     Segment<K,V> s; // manually integrate access methods to reduce overhead
3     HashEntry<K,V>[] tab;
4     int h = hash(key.hashCode());
5     // 利用位操作替换普通数学运算
6     long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
7     // 以 Segment 为单位，进行定位
8     // 利用 Unsafe 直接进行 volatile access
9     if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
10         (tab = s.table) != null) {
11         // 省略
12     }
13     return null;
14 }


```

而对于 put 操作，首先是通过二次哈希避免哈希冲突，然后以 Unsafe 调用方式，直接获取相应的 Segment，然后进行线程安全的 put 操作：

 复制代码

```
1 public V put(K key, V value) {
2     Segment<K,V> s;
3     if (value == null)
4         throw new NullPointerException();
5     // 二次哈希，以保证数据的分散性，避免哈希冲突
6     int hash = hash(key.hashCode());
7     int j = (hash >>> segmentShift) & segmentMask;
8     if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
9         (segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment
10        s = ensureSegment(j);
11    return s.put(key, hash, value, false);
12 }
13
```

其核心逻辑实现在下面的内部方法中：

 复制代码

```
1 final V put(K key, int hash, V value, boolean onlyIfAbsent) {
2     // scanAndLockForPut 会去查找是否有 key 相同 Node
3     // 无论如何，确保获取锁
4     HashEntry<K,V> node = tryLock() ? null :
5         scanAndLockForPut(key, hash, value);
6     V oldValue;
7     try {
8         HashEntry<K,V>[] tab = table;
9         int index = (tab.length - 1) & hash;
10        HashEntry<K,V> first = entryAt(tab, index);
11        for (HashEntry<K,V> e = first;;) {
12            if (e != null) {
13                K k;
14                // 更新已有 value...
15            }
16            else {
17                // 放置 HashEntry 到特定位置，如果超过阈值，进行 rehash
18                // ...
19            }
20        }
21    } finally {
22        unlock();
23    }
24    return oldValue;
25 }
26
```

所以，从上面的源码清晰的看出，在进行并发写操作时：

- ConcurrentHashMap 会获取再入锁，以保证数据一致性，Segment 本身就是基于 ReentrantLock 的扩展实现，所以，在并发修改期间，相应 Segment 是被锁定的。
- 在最初阶段，进行重复性的扫描，以确定相应 key 值是否已经在数组里面，进而决定是更新还是放置操作，你可以在代码里看到相应的注释。重复扫描、检测冲突是 ConcurrentHashMap 的常见技巧。
- 我在专栏上一讲介绍 HashMap 时，提到了可能发生的扩容问题，在 ConcurrentHashMap 中同样存在。不过有一个明显区别，就是它进行的不是整体的扩容，而是单独对 Segment 进行扩容，细节就不介绍了。

另外一个 Map 的 size 方法同样需要关注，它的实现涉及分离锁的一个副作用。

试想，如果不进行同步，简单的计算所有 Segment 的总值，可能会因为并发 put，导致结果不准确，但是直接锁定所有 Segment 进行计算，就会变得非常昂贵。其实，分离锁也限制了 Map 的初始化等操作。


所以，ConcurrentHashMap 的实现是通过重试机制（RETRIES_BEFORE_LOCK，指定重试次数 2），来试图获得可靠值。如果没有监控到发生变化（通过对比 Segment.modCount），就直接返回，否则获取锁进行操作。

下面我来对比一下，在 Java 8 和之后的版本中，ConcurrentHashMap 发生了哪些变化呢？

- 总体结构上，它的内部存储变得和我在专栏上一讲介绍的 HashMap 结构非常相似，同样是大的桶（bucket）数组，然后内部也是一个个所谓的链表结构（bin），同步的粒度要更细致一些。
- 其内部仍然有 Segment 定义，但仅仅是为了保证序列化时的兼容性而已，不再有任何结构上的用处。
- 因为不再使用 Segment，初始化操作大大简化，修改为 lazy-load 形式，这样可以有效避免初始开销，解决了老版本很多人抱怨的这一点。
- 数据存储利用 volatile 来保证可见性。
- 使用 CAS 等操作，在特定场景进行无锁并发操作。
- 使用 Unsafe、LongAdder 之类底层手段，进行极端情况的优化。

先看看现在的数据存储内部实现，我们可以发现 Key 是 final 的，因为在生命周期中，一个条目的 Key 发生变化是不可能的；与此同时 val，则声明为 volatile，以保证可见性。


```
1 static class Node<K,V> implements Map.Entry<K,V> {
```

 复制代码

```
2    final int hash;
3    final K key;
4    volatile V val;
5    volatile Node<K,V> next;
6    // ...
7 }
```

我这里就不再介绍 get 方法和构造函数了，相对比较简单，直接看并发的 put 是如何实现的。

```
1 final V putVal(K key, V value, boolean onlyIfAbsent) { if (key == null || value == null) throw r
2     int hash = spread(key.hashCode());
3     int binCount = 0;
4     for (Node<K,V>[] tab = table;;) {
5         Node<K,V> f; int n, i, fh; K fk; V fv;
6         if (tab == null || (n = tab.length) == 0)
7             tab = initTable();
8         else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
9             // 利用 CAS 去进行无锁线程安全操作，如果 bin 是空的
10            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
11                break;
12        }
13        else if ((fh = f.hash) == MOVED)
14            tab = helpTransfer(tab, f);
15        else if (onlyIfAbsent // 不加锁，进行检查
16                && fh == hash
17                && ((fk = f.key) == key || (fk != null && key.equals(fk)))
18                && (fv = f.val) != null)
19            return fv;
20        else {
21            V oldVal = null;
22            synchronized (f) {
23                // 细粒度的同步修改操作...
24            }
25        }
26        // Bin 超过阈值，进行树化
27        if (binCount != 0) {
28            if (binCount >= TREEIFY_THRESHOLD)
29                treeifyBin(tab, i);
30            if (oldVal != null)
31                return oldVal;
32            break;
33        }
34    }
35 }
36 addCount(1L, binCount);
37 return null;
38 }
39
```

 复制代码

初始化操作实现在 `initTable` 里面，这是一个典型的 CAS 使用场景，利用 `volatile` 的 `sizeCtl` 作为互斥手段：如果发现竞争性的初始化，就 `spin` 在那里，等待条件恢复；否则利用 CAS 设置排他标志。如果成功则进行初始化；否则重试。

请参考下面代码：

[复制代码](#)

```
1 private final Node<K,V>[] initTable() {
2     Node<K,V>[] tab; int sc;
3     while ((tab = table) == null || tab.length == 0) {
4         // 如果发现冲突，进行 spin 等待
5         if ((sc = sizeCtl) < 0)
6             Thread.yield();
7         // CAS 成功返回 true，则进入真正的初始化逻辑
8         else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
9             try {
10                 if ((tab = table) == null || tab.length == 0) {
11                     int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
12                     @SuppressWarnings("unchecked")
13                     Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
14                     table = tab = nt;
15                     sc = n - (n >>> 2);
16                 }
17             } finally {
18                 sizeCtl = sc;
19             }
20             break;
21         }
22     }
23     return tab;
24 }
25
```

当 `bin` 为空时，同样是没有必要锁定，也是以 CAS 操作去放置。

你有没有注意到，在同步逻辑上，它使用的是 `synchronized`，而不是通常建议的 `ReentrantLock` 之类，这是为什么呢？现代 JDK 中，`synchronized` 已经被不断优化，可以不再过分担心性能差异，另外，相比于 `ReentrantLock`，它可以减少内存消耗，这是个非常大的优势。

与此同时，更多细节实现通过使用 `Unsafe` 进行了优化，例如 `tabAt` 就是直接利用 `getObjectAcquire`，避免间接调用的开销。

[复制代码](#)

```
1 static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
2     return (Node<K,V>)U.getObjectAcquire(tab, ((long)i << ASHIFT) + ABASE);
3 }
4
```


再看看，现在是如何实现 size 操作的。[阅读代码](#)你会发现，真正的逻辑是在 sumCount 方法中，那么 sumCount 做了什么呢？

```
1 final long sumCount() {
2     CounterCell[] as = counterCells; CounterCell a;
3     long sum = baseCount;
4     if (as != null) {
5         for (int i = 0; i < as.length; ++i) {
6             if ((a = as[i]) != null)
7                 sum += a.value;
8         }
9     }
10    return sum;
11 }
12
```

[复制代码](#)

我们发现，虽然思路仍然和以前类似，都是分而治之的进行计数，然后求和处理，但实现却基于一个奇怪的 CounterCell。难道它的数值，就更加准确吗？数据一致性是怎么保证的？

```
1 static final class CounterCell {
2     volatile long value;
3     CounterCell(long x) { value = x; }
4 }
```

[复制代码](#)

其实，对于 CounterCell 的操作，是基于 java.util.concurrent.atomic.LongAdder 进行的，是一种 JVM 利用空间换取更高效的方法，利用了[Striped64](#)内部的复杂逻辑。这个东西非常小众，大多数情况下，建议还是使用 AtomicLong，足以满足绝大部分应用的性能需求。

今天我从线程安全问题开始，概念性的总结了基本容器工具，分析了早期同步容器的问题，进而分析了 Java 7 和 Java 8 中 ConcurrentHashMap 是如何设计实现的，希望 ConcurrentHashMap 的并发技巧对你在日常开发可以有所帮助。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一个思考题给你，在产品代码中，有没有典型的场景需要使用类似 ConcurrentHashMap 这样的并发容器呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Java核心技术36讲

—— 前 Oracle 首席工程师
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



徐金铎

👍 26

需要注意的一点是，1.8以后的锁的颗粒度，是加在链表头上的，这个是个思路上的突破。

2018-05-26

作者回复

是的

2018-05-28



明翼

👍 19

1.7

put加锁

通过分段加锁segment，一个hashmap里有若干个segment，每个segment里有若干个桶，桶里存放K-V形式的链表，put数据时通过key哈希得到该元素要添加到的segment，然后对segment进行加锁，然后在哈希，计算得到给元素要添加到的桶，然后遍历桶中的链表，替换或新增节点到桶中

size

分段计算两次，两次结果相同则返回，否则对所以段加锁重新计算

1.8

put CAS 加锁

1.8中不依赖与segment加锁，segment数量与桶数量一致；

首先判断容器是否为空，为空则进行初始化利用volatile的sizeCtl作为互斥手段，如果发现竞

争性的初始化，就暂停在那里，等待条件恢复，否则利用CAS设置排他标志（U.compareAndSwapInt(this, SIZECTL, sc, -1)）；否则重试

对key hash计算得到该key存放的桶位置，判断该桶是否为空，为空则利用CAS设置新节点
否则使用synchronize加锁，遍历桶中数据，替换或新增加点到桶中

最后判断是否需要转为红黑树，转换之前判断是否需要扩容

size

利用LongAdd累加计算

2018-07-04



雷霹雳的爸爸

👍 16

今天这个纯粹知识盲点，纯赞，源码也得不停看

2018-05-26



j.c.

👍 14

期待unsafe和cas的文章

2018-05-26



Sean

👍 9

最近用ConcurrentHashMap的场景是，由于系统是一个公共服务，全程异步处理。最后一环节需要http rest主动响应接入系统，于是为了定制化需求，利用netty写了一版异步http client。其在缓存tcp链接时用到了。

看到下面有一位朋友说起了自旋锁和偏向锁。

自旋锁个人理解的是cas的一种应用方式。并发包中的原子类是典型的应用。

偏向锁个人理解的是获取锁的优化。在ReentrantLock中用于实现已获取完锁的的线程重入问题。

不知道理解的是否有误差。欢迎指正探讨。谢谢

2018-05-28

作者回复

正确，互相交流

偏向锁，侧重是低竞争场景的优化，去掉可能不必要的同步

2018-05-28



虞飞

👍 5

老师在课程里讲到同步包装类比较低效，不太适合高并发的场景，那想请教一下老师，在list接口的实现类中。在高并发的场景下，选择哪种实现类比较好？因为ArrayList是线程不安全的，同步包装类又很低效，CopyonwriteArrayList又是以快照的形式来实现的，在频繁写入数据的时候，其实也很低效，那这个类型该怎么选择比较好？

2018-05-27

作者回复

目前并发list好像就那一个，我觉得不必拘泥于list，不还有queue之类，看场景需要的真是list吗

2018-05-28



coder王

👍 3

您说的synchronized被改进很多很多了，那么在我们平常使用中，就用这个synchronized完成一些同步操作是不是OK？😊

2018-05-28

| 作者回复

通常是的，前提是JDK版本需要新一点

2018-05-28



shawn

👍 2

老师，什么只有bin为空的时候才使用cas，其他地方用synchronized呢？

2018-07-02



Kyle

👍 2

之前用JavaFX做一个客户端IM工具的时候，我将拉来的未被读取的用户聊天信息用ConcurrentHashMap存储（同时异步存储到Sqlite），Key存放用户id，Value放未读取的聊天消息列表。因为我考虑到存消息和读消息是由两个线程并发处理的，这两个线程共同操作一个ConcurrentHashMap。可能是我没处理好，最后直到我离职了还有消息重复、乱序的问题。请问我这种应用场景有什么问题吗？

2018-05-28



行者

👍 2

老师麻烦讲讲自旋锁，偏向锁的特点和区别吧，一直不太清楚。

2018-05-27

| 作者回复

好，后面有章节

2018-05-28



mongo

👍 2

请教老师：putVal方法中，什么情况下会进入else if ((fh=f.hash) == MOVED)分支？是进行扩容的时候吗？nextTable是做什么用的？

2018-05-26

| 作者回复

我理解是的，判断是个ForwardingNode，resize正在进行；

nexttable是扩容时的临时过渡

2018-05-28



mongo

👍 2

请教老师：putVal方法的第二个if分支，为什么要用tabAt？我的认识里直接数组下标寻址tab[i]=(n-1) & hash]也是一个原子操作，不是吗？tabAt里面的getObjectVolatile()方法跟直接用数组下标tab[i]=(n-1) & hash]寻址有什么区别？

2018-05-26

| 作者回复

这个有volatile load语义

2018-05-28



曹铮

👍 2

这期内容太难，分寸不好把握

看8的concurrenthashmap源码感觉挺困难，网上的博文帮助也不大，尤其是扩容这部分（似乎文章中没提）

求问杨大有没有什么窍门，或者有什么启发性的paper或文章？

可以泛化成，长期对lock free实现多个状态修改的问题比较困惑，希望得到启发

2018-05-26

作者回复

本文尽量梳理了相对比较容易理解的部分；扩容细节我觉得是个加分项，不是每个人都会在乎那么深入；窍门，可以考虑画图辅助理解，我是比较笨的类型，除了死磕，不会太多窍门.....

2018-05-28



Answer

👍 1

Unsafe ?

2018-07-03



Xg huang

👍 1

这里有个地方想跟老师交流一下想法, 从文中"所以, ConcurrentHashMap 的实现是通过重试机制 (RETRIES_BEFORE_LOCK , 指定重试次数 2) , 来试图获得可靠值。如果没有监控到发生变化 (通过对比 Segment.modCount) , 就直接返回, 否则获取锁进行操作。" 可以看出, 在高并发的情况下, "size()" 方法只是返回"近似值", 而我的问题是: 既然只是一个近似值, 为啥要用这种"重试,分段锁" 的复杂做法去计算这个值? 直接在不加锁的情况下返回segment 的size 岂不是更简单? 我能理解jdk开发者想尽一切努力在高性能地返回最精确的数值, 但这个"精确" 度无法量化啊, 对于调用方来说, 这个值依然是不可靠的啊. 所以, 在我看来, 这种做法收益很小(可能是我也比较懒吧), 或者有些设计上的要点我没有领悟出来, 希望老师指点一下.

2018-06-08

作者回复

这个是在代价可接受情况下，尽量准确，就像含金量90%和99.9%，99.999%，还是有区别的，虽然不是百分百

2018-06-08



Levy

👍 1

老师你好，tabAt里面的getObjectVolatile () 方法跟直接用数组下标tab[i]=(n-1) & hash] 寻址有什么区别，这个我也不懂，volatile不是已经保证内存可见性吗？

2018-05-30

作者回复

volatile保证的是数组，不是数组元素

2018-05-31



Leiy

👍 1

我感觉jdk8就相当于把segment分段锁更细粒度了，每个数组元素就是原来一个segment，那并发度就由原来segment数变为数组长度？而且用到了cas乐观锁，所以能支持更高的并发，不知道我这种理解对吗？如果对的话，我就在想，为什么并发大神之前没想到这种，哈哈😄，恳请指正。谢谢

2018-05-29

| 作者回复

基本正确，cas只用在部分场景；

事后看容易啊，说比做起来容易，😄

2018-05-29



Heshher

👍 1

并发包用的很少，这一节内容的前置知识比较多，对于使用经验少的人来说貌似是有点难了。问题很好，正好可以见识一下各种使用场景，不过留言大部分是针对内容的难点提问，而真正回答问题的还没有出现。

2018-05-28

| 作者回复

后面并发部分会详细分析

2018-05-28



tim

👍 0

syncMap 还是有改进的，起码加了synchronized 不会导致扩容死锁

2018-07-28



t

👍 0

对于我这种菜鸟来说，应该来一期讲讲volatile😄

2018-07-03