

二

10 如何保证集合是线程安全的 ConcurrentHashMap如何实现高效地线程安全？ - 极客时间

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% 等一些问題，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！

```
private static class SynchronizedMap<K,V>
    implements Map<K,V>, Serializable {
    private final Map<K,V> m;        // Backing Map
    final Object      mutex;          // Object on which to synchronize
    // ...
    public int size() {
        synchronized (mutex) {return m.size();}
    }
    // ...
}
```

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

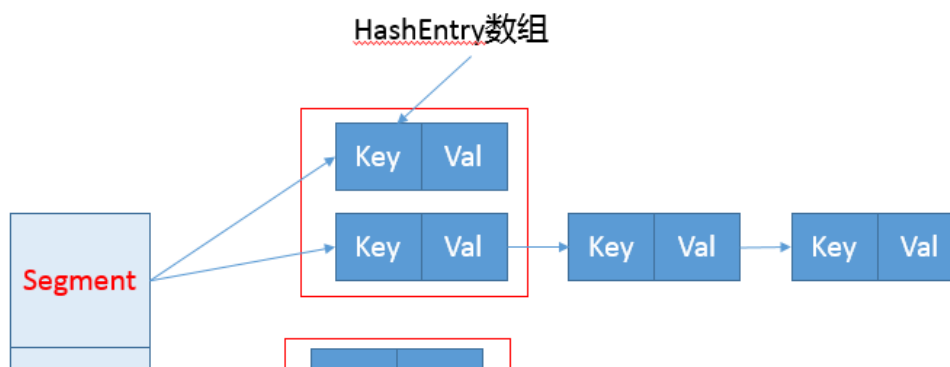
我们再来看看 ConcurrentHashMap 是如何设计实现的，为什么它能大大提高并发效率。

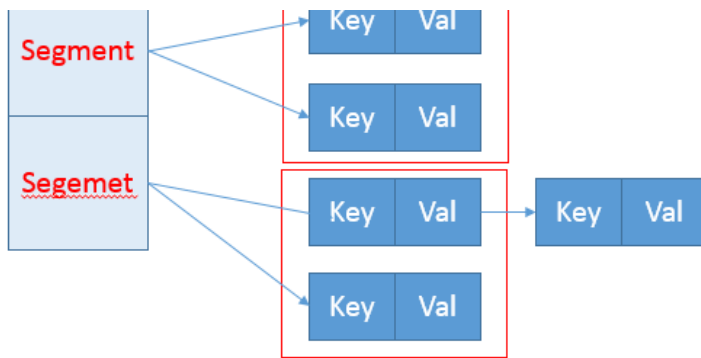
首先，我这里强调，**ConcurrentHashMap 的设计实现其实一直在演化**，比如在 Java 8 中就发生了非常大的变化（Java 7 其实也有不少更新），所以，我这里将比较分析结构、实现机制等方面，对比不同版本的主要区别。

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。

```

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

```

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：

```

public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    // 二次哈希，以保证数据的分散性，避免哈希冲突
    int hash = hash(key.hashCode());
    int j = (hash >>> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject(segments, (j << SSHIFT) + SBASE)) == null) // nonvolatile; recheck
        // in ensureSegment

```

```
        s = ensureSegment(j);  
        return s.put(key, hash, value, false);  
    }
```

其核心逻辑实现在下面的内部方法中：

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {  
    // scanAndLockForPut会去查找是否有key相同Node  
    // 无论如何，确保获取锁  
    HashEntry<K,V> node = tryLock() ? null :  
        scanAndLockForPut(key, hash, value);  
    V oldValue;  
    try {  
        HashEntry<K,V>[] tab = table;  
        int index = (tab.length - 1) & hash;  
        HashEntry<K,V> first = entryAt(tab, index);  
        for (HashEntry<K,V> e = first;;) {  
            if (e != null) {  
                K k;  
                // 更新已有value...  
            }  
            else {  
                // 放置HashEntry到特定位置，如果超过阈值，进行rehash  
                // ...  
            }  
        }  
    } finally {  
        unlock();  
    }  
    return oldValue;  
}
```

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。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    // ...
}
```

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。

```
final V putVal(K key, V value, boolean onlyIfAbsent) { if (key == null || value ==
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh; K fk; V fv;
```

```

    if (tab == null || (n = tab.length) == 0)
        tab = initTable();
    else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
        // 利用CAS去进行无锁线程安全操作，如果bin是空的
        if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))
            break;
    }
    else if ((fh = f.hash) == MOVED)
        tab = helpTransfer(tab, f);
    else if (onlyIfAbsent // 不加锁，进行检查
            && fh == hash
            && ((fk = f.key) == key || (fk != null && key.equals(fk)))
            && (fv = f.val) != null)
        return fv;
    else {
        V oldVal = null;
        synchronized (f) {
            // 细粒度的同步修改操作...
        }
    }
    // Bin超过阈值，进行树化
    if (binCount != 0) {
        if (binCount >= TREEIFY_THRESHOLD)
            treeifyBin(tab, i);
        if (oldVal != null)
            return oldVal;
        break;
    }
}
addCount(1L, binCount);
return null;
}

```

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：

```

private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        // 如果发现冲突，进行spin等待
        if ((sc = sizeCtl) < 0)
            Thread.yield();
        // CAS成功返回true，则进入真正的初始化逻辑
        else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")

```



```

        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
        table = tab = nt;
        sc = n - (n >>> 2);
    }
    } finally {
        sizeCtl = sc;
    }
    break;
}
}
return tab;
}

```

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。

```

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

```

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？

```

final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}

```

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？


```
static final class CounterCell {  
    volatile long value;  
    CounterCell(long x) { value = x; }  
}
```

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率的方法，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。

[上一页](#)

[下一页](#)