



Hashtable, SynchronizedMap和ConcurrentHashMap线程安全实现的区别以及性能测试

java 线程安全 map 发布于 2019-11-28

Hashtable, Collections.SynchronizedMap和ConcurrentHashMap线程安全实现原理的区别以及性能测试

这三种 **Map** 都是 **Java** 中比较重要的集合类，虽然前两个不太常用，但是因为与多线程相关，所以关于这几种 **Map** 的对比已经成为了 **Java** 面试时的高频考点。首先要说明的是，其中每一个单独拎出来都足够支撑一篇长篇大论的技术文章，所以本文把重点放在了这三种集合类的线程安全实现原理的对比以及性能测试上，其他细节不做深入探讨。

一、线程安全原理对比

1. Hashtable

首先必须吐槽一下这个类名，作为官方工具类竟然不符合驼峰命名规则，怪不得被弃用了，开玩笑哈哈，主要原因还是性能低下，那 **Hashtable** 的性能为什么低下呢，这个嘛只需要看一下它的源码就一目了然了，以下是 **Hashtable** 中几个比较重要的方法：

```
public synchronized V put(K key, V value) {...}
public synchronized V get(Object key) {...}
public synchronized int size() {...}
public synchronized boolean remove(Object key, Object value) {...}
public synchronized boolean contains(Object value) {...}
... ..
```

查看源码后可以看出，**Hashtable** 实现线程安全的原理相当简单粗暴，直接在方法声明上使用 **synchronized** 关键字。这样一来，不管线程执行哪个方法，即便只是读取数据，都需要锁住整个 **Hashtable** 对象，可想而知其并发性能必然不会太好。

2. Collections.SynchronizedMap

SynchronizedMap 是 **Collections** 集合类的私有静态内部类，其定义和构造方法如下：

```
private static class SynchronizedMap<K,V> implements Map<K,V>, Serializable {
    private static final long serialVersionUID = 1978198479659022715L;
    // 用于接收传入的Map对象，也是类方法操作的对象
    private final Map<K,V> m;
    // 锁对象
    final Object mutex;

    // 以下是SynchronizedMap的两个构造方法
    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m);
        mutex = this;
    }
    SynchronizedMap(Map<K,V> m, Object mutex) {
        this.m = m;
        this.mutex = mutex;
    }
}
```

- **SynchronizedMap** 一共有三个成员变量，序列化ID抛开不谈，另外两个分别是 **Map** 类型的实例变量 **m**，用于接收构造方法中传入的 **Map** 参数，以及 **Object** 类型的实例变量 **mutex**，作为锁对象使用。
- 再来看构造方法，**SynchronizedMap** 有两个构造方法。第一个构造方法需要传入一个 **Map** 类型的参数，这个参数会被传递给成员变量 **m**，接下来 **SynchronizedMap** 所有方法的操作都是针对 **m** 的操作，需要注意的是这个参数不能为空，否则会由 **Objects** 类的 **requireNonNull()** 方法抛出空指针异常，然后当前的 **SynchronizedMap** 对象 **this** 会被传递给 **mutex** 作为锁

对象；第二个构造方法有两个参数，第一个 **Map** 类型的参数会被传递给成员变量 **m**，第二个 **Object** 类型的参数会被传递给 **mutex** 作为锁对象。

- 最后来看 **SynchronizedMap** 的主要方法 (只选取了一部分)：

```
public int size() {
    synchronized (mutex) {return m.size();}
}
public boolean isEmpty() {
    synchronized (mutex) {return m.isEmpty();}
}
public boolean containsKey(Object key) {
    synchronized (mutex) {return m.containsKey(key);}
}
public V get(Object key) {
    synchronized (mutex) {return m.get(key);}
}
public V put(K key, V value) {
    synchronized (mutex) {return m.put(key, value);}
}
public V remove(Object key) {
    synchronized (mutex) {return m.remove(key);}
}
```

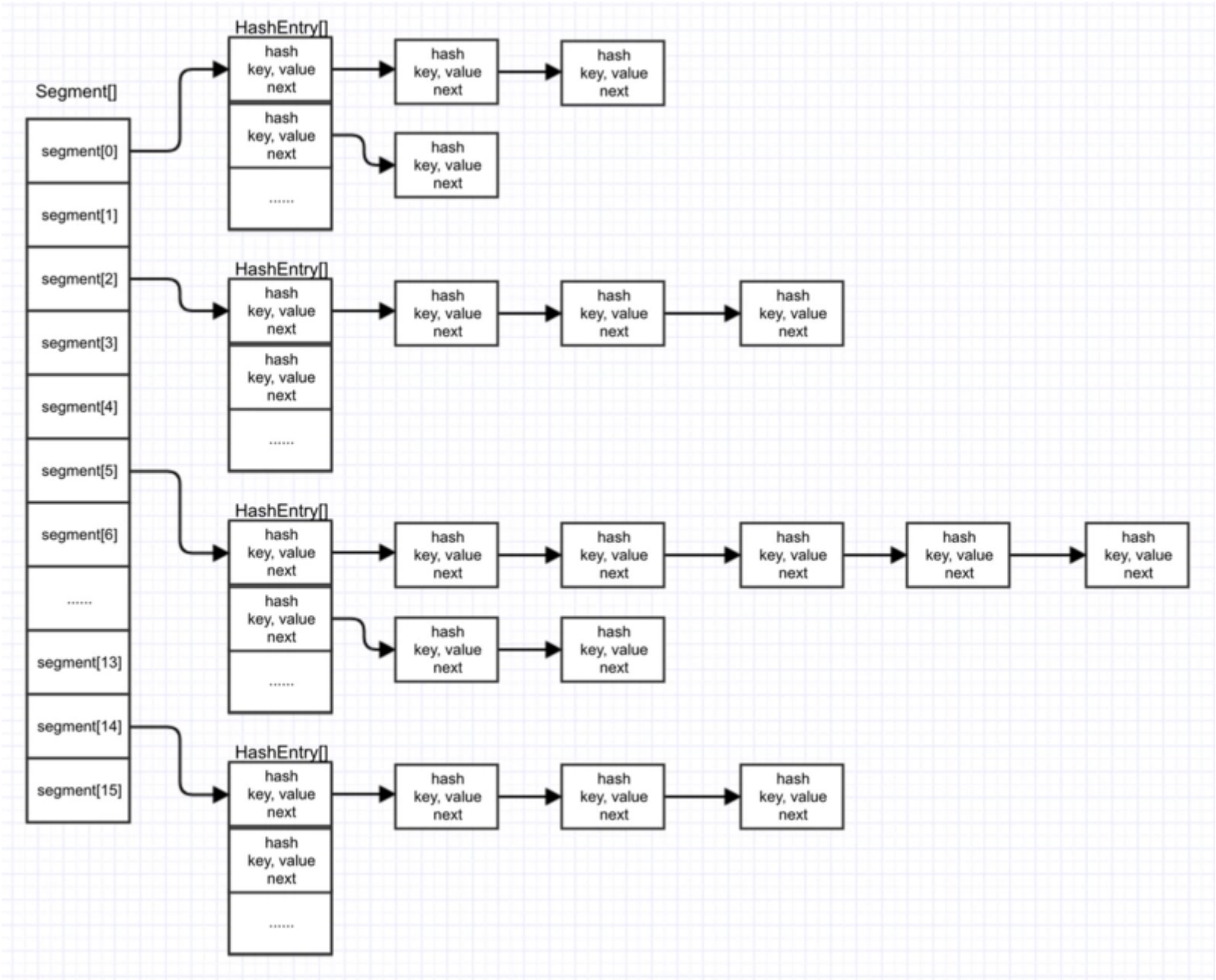
从源码可以看出，**SynchronizedMap** 实现线程安全的方法也是比较简单的，所有方法都是先对锁对象 **mutex** 上锁，然后再直接调用 **Map** 类型成员变量 **m** 的相关方法。这样一来，线程在执行方法时，只有先获得了 **mutex** 的锁才能对 **m** 进行操作。因此，跟 **Hashtable** 一样，在同一个时间点，只能有一个线程对 **SynchronizedMap** 对象进行操作，虽然保证了线程安全，却导致了性能低下。这么看来，连 **Hashtable** 都被弃用了，那性能同样低下的 **SynchronizedMap** 还有什么存在的必要呢？别忘了，后者的构造方法需要传入一个 **Map** 类型的参数，也就是说它可以将非线程安全的 **Map** 转化为线程安全的 **Map**，而这正是其存在的意义，以下是 **SynchronizedMap** 的用法示例 (这里并没有演示多线程操作)：

```
Map<String, Integer> map = new HashMap<>();
// 非线程安全操作
map.put("one", 1);
Integer one = map.get("one");
Map<String, Integer> synchronizedMap = Collections.synchronizedMap(map);
// 线程安全操作
one = synchronizedMap.get("one");
synchronizedMap.put("two", 2);
Integer two = synchronizedMap.get("two");
```

3. ConcurrentHashMap

接下来是数据结构和线程安全原理都最复杂的 **ConcurrentHashMap**。首先必须要感叹一下，这个类的结构之复杂（包含53个内部类），设计之精妙（不知道怎么形容，反正就是很精妙），真是令人叹为观止。说实话，要想彻底理解 **ConcurrentHashMap** 的各个细节，还是需要相当扎实的基础并花费大量精力的。本文对于 **ConcurrentHashMap** 线程安全的原理只是做了宏观的介绍，想要深入理解的同学，可以去文末的传送门。另外，本文着重介绍 JDK 1.8 版本的 **ConcurrentHashMap**，不过会对 JDK 1.7 版本做个简单的回顾。

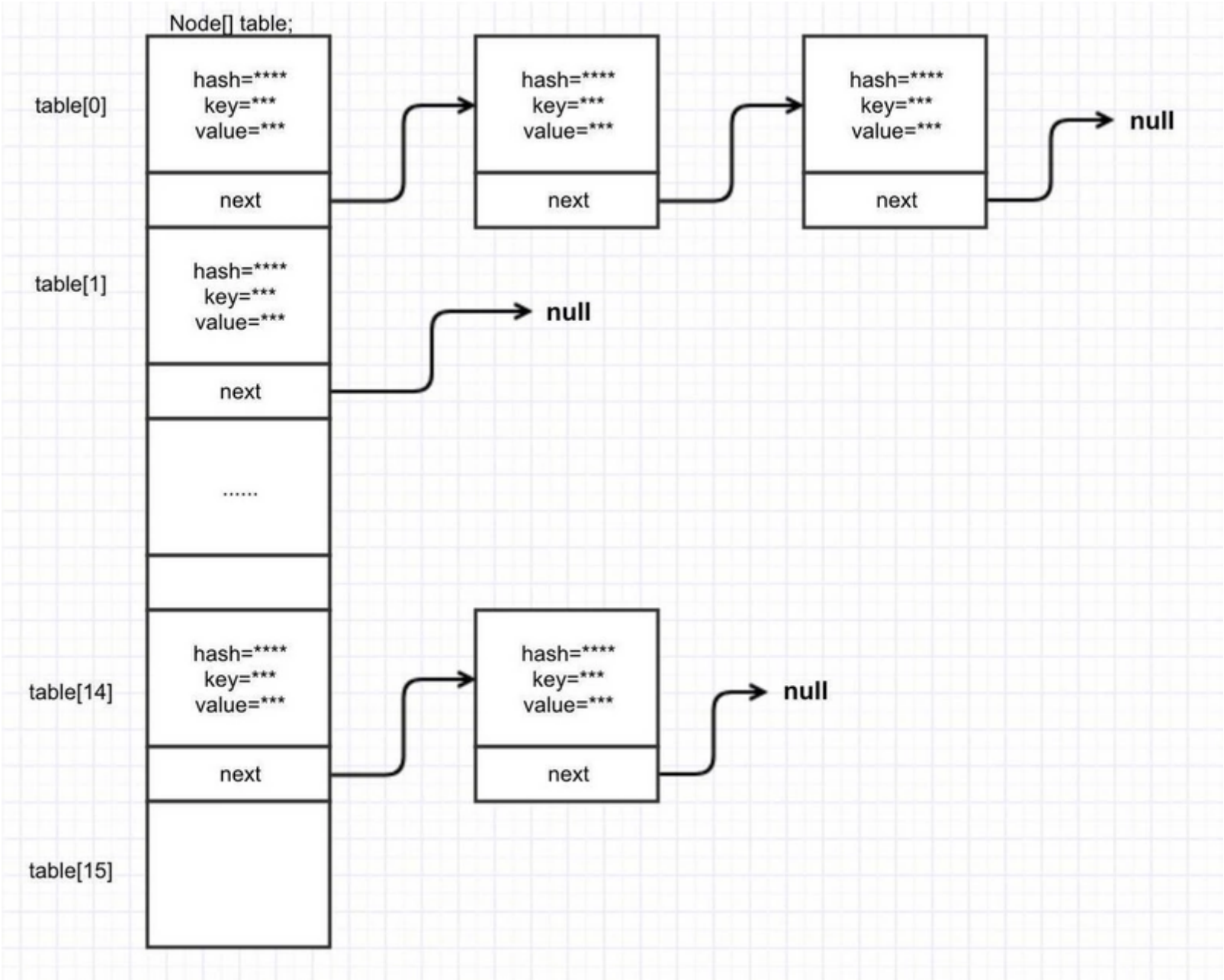
3.1 JDK 1.7 ConcurrentHashMap锁实现原理回顾



Java7 ConcurrentHashMap结构示意图

如果你有一定基础的话，应该会知道分段锁这个概念。没错，这是JDK 1.7版本的 **ConcurrentHashMap** 实现线程安全的主要手段，具体一点就是 **Segment + HashEntry + ReentrantLock**。简单来说，**ConcurrentHashMap** 是一个 **Segment** 数组（默认长度为16），每个 **Segment** 又包含了一个 **HashEntry** 数组，所以可以看做一个 **HashMap**，**Segment** 通过继承 **ReentrantLock** 来进行加锁，所以每次需要加锁的操作锁住的是一个 **Segment**，这样只要保证每个 **Segment** 是线程安全的，也就实现了全局的线程安全。

3.2 JDK 1.8 ConcurrentHashMap 线程安全原理详解



Java8 ConcurrentHashMap结构示意图

JDK 1.8 版本摒弃了之前版本中较为臃肿的 **Segment** 分段锁设计，取而代之的是 **Node 数组 + CAS + synchronized + volatile** 的新设计。这样一来，**ConcurrentHashMap** 不仅数据结构变得更简单了（与JDK 1.8 的HashMap类似），锁的粒度也更小了，锁的单位从 **Segment** 变成了 **Node** 数组中的桶（科普：桶就是指数组中某个下标位置上的数据集合，这里可能是链表，也可能是红黑树）。说到红黑树，必须提一下，在JDK 1.8 的 **HashMap** 和 **ConcurrentHashMap** 中，如果某个数组位置上的链表长度过长（大于等于8），就会转化为红黑树以提高查询效率，不过这不是本文的重点。以下是 **ConcurrentHashMap** 线程安全原理的详细介绍：

◆ 3.2.1 get 操作过程

可以发现发现源码中完全没有加锁的操作，后面会说明原因

1. 首先计算hash值，定位到该table索引位置，如果是首节点符合就返回
2. 如果遇到扩容的时候，会调用标志正在扩容节点ForwardingNode的find方法，查找该节点，匹配就返回
3. 以上都不符合的话，就往下遍历节点，匹配就返回，否则最后就返回null


```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode()); //计算hash
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) { // 读取头节点的Node元素
        if ((eh = e.hash) == h) { // 如果该节点就是首节点就返回
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        //hash值为负值表示正在扩容，这个时候查的是ForwardingNode的find方法来定位到nextTable来
        //eh代表头节点的hash值，eh=-1，说明该节点是一个ForwardingNode，正在迁移，此时调用ForwardingNode的find方法去nextTable里找。
        //eh=-2，说明该节点是一个TreeBin，此时调用TreeBin的find方法遍历红黑树，由于红黑树有可能正在旋转变色，所以find里会有读写锁。
        //eh>=0，说明该节点下挂的是一个链表，直接遍历该链表即可。
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) { // 既不是首节点也不是ForwardingNode，那就往下遍历
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

可能有同学会提出疑问：为什么 **get** 操作不需要加锁呢？这个嘛，也需要看一下源码：

```
/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
 * 这是ConcurrentHashMap的成员变量，用 volatile修饰的Node数组，保证了数组在扩容时对其他线程的可见性
 * 另外需要注意的是，这个数组是延迟初始化的，会在第一次put元素时进行初始化，后面还会用到这个知识点
 */
transient volatile Node<K,V>[] table;

/**
 * 这是ConcurrentHashMap静态内部类Node的定义，可见其成员变量val和next都使用volatile修饰，可保证
 * 在多线程环境下线程A修改结点的val或者新增节点的时候是对线程B可见的
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
}
```

使用 **volatile** 关键字已经足以保证线程在读取数据时不会读取到脏数据，所以没有加锁的必要。

◆ 3.2.2 put 操作过程

1. 第一次 **put** 元素会初始化 **Node** 数组 (initTable)
2. **put** 操作又分为 **key** (**hash** 碰撞) 存在时的插入和 **key** 不存在时的插入
3. **put** 操作可能会引发数组扩容 (tryPresize) 和链表转红黑树 (treeifyBin)
4. 扩容会使用到数据迁移方法 (transfer)

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 得到 hash 值
    int hash = spread(key.hashCode());
    // 用于记录相应链表的长度
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果数组"空", 进行数组初始化
        if (tab == null || (n = tab.length) == 0)
            // 表的初始化, 这里不展开了, 核心思想是使用sizeCtl的变量和CAS操作进行控制, 保证数组在扩容时
            // 不会创建出多余的表
            tab = initTable();
        // 找该 hash 值对应的数组下标, 得到第一个节点 f
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 如果数组该位置为空, 用一次 CAS 操作将这个新值放入其中即可, 这个 put 操作差不多就结束了
            // 如果 CAS 失败, 那就是有并发操作, 进到下一个循环就好了(循环的意思是 CAS 在执行失败后会进行
            // 重试)
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
                break; // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            // 帮助数据迁移,
            tab = helpTransfer(tab, f);
        else { // 到这里就是说, f 是该位置的头结点, 而且不为空
            V oldVal = null;
```

以上是 **put** 方法的源码和分析，其中涉及到的其他方法，比如 **initTable**，**helpTransfer**，**treeifyBin** 和 **tryPresize** 等方法不再一一展开，有兴趣的同学可以去文末传送门看详细解析。

◆ 3.2.3 CAS 操作简要介绍

CAS 操作是新版本 **ConcurrentHashMap** 线程安全实现原理的精华所在，如果说其共享变量的读取全靠 **volatile** 实现线程安全的话，那么存储和修改过程除了使用少量的 **synchronized** 关键字外，主要是靠 **CAS** 操作实现线程安全的。不了解 **CAS** 操作的同学看这里 [JAVA CAS原理深度分析](#)

```
// CAS操作的提供者
private static final sun.misc.Unsafe U;

// 以下是put方法里用到CAS操作的代码片段
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null,
        new Node<K,V>(hash, key, value, null)))
        break;
}

// tabAt方法通过Unsafe.getObjectVolatile()的方式获取数组对应index上的元素, getObjectVolatile作用于对
// 应的内存偏移量上, 是具备volatile内存语义的。
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}

// 如果获取的是空, 尝试用CAS的方式在数组的指定index上创建一个新的Node。
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

在 **ConcurrentHashMap** 中，数组初始化、插入删除元素、扩容、数据迁移以及链表和红黑树的转换等过程都会涉及到线程安全问题，而相关的方法中实现线程安全的思想是一致的：对桶中的数据进行添加或修改操作时会用到 **synchronized** 关键字，也就是获得该位置上头节点对象的锁，保证线程安全，另外就是用到了大量的 **CAS** 操作。以上就是对这三种 **Map** 的线程安全原理的简要介绍。

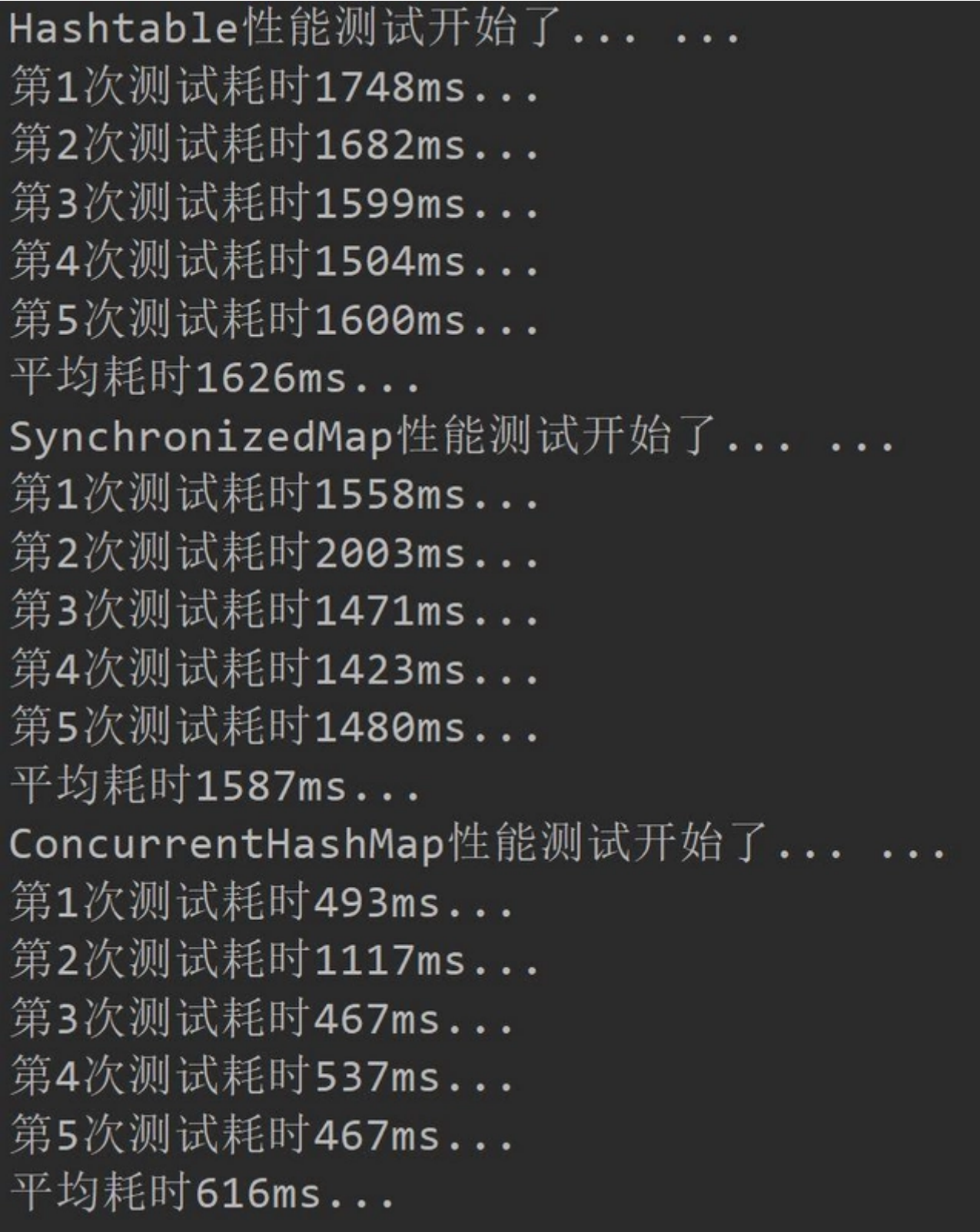
二、性能测试

直接上代码

```
public class MapPerformanceTest {
    private static final int THREAD_POOL_SIZE = 5;
    private static Map<String, Integer> hashtableObject = null;
    private static Map<String, Integer> concurrentHashMapObject = null;
    private static Map<String, Integer> synchronizedMap = null;

    private static void performanceTest(final Map<String, Integer> map) throws InterruptedException {
        System.out.println(map.getClass().getSimpleName() + "性能测试开始了... ");
        long totalTime = 0;
        // 进行五次性能测试, 每次开启五个线程, 每个线程对 map 进行500000次查询操作和500000次插入操作
        for (int i = 0; i < 5; i++) {
            long startTime = System.nanoTime();
            ExecutorService service = Executors.newFixedThreadPool(THREAD_POOL_SIZE);
            for (int j = 0; j < THREAD_POOL_SIZE; j++) {
                service.execute(() -> {
                    for (int k = 0; k < 500000; k++) {
                        Integer randomNumber = (int)Math.ceil(Math.random() * 500000);
                        // 从map中查找数据, 查找结果并不会用到, 这里不能用int接收返回值, 因为Integer可能是
                        // null, 赋值给int会引发空指针异常
                        Integer value = map.get(String.valueOf(randomNumber));
                        // 向map中添加元素
                        map.put(String.valueOf(randomNumber), randomNumber);
                    }
                });
            }
            totalTime += System.nanoTime() - startTime;
        }
        // 关闭线程池
    }
}
```

在这里说明一点，这段代码在不同环境下运行的结果会存在差别，但是结果的数量级对比应该是一致的，以下是我机子上的运行结果：



Map性能测试结果

从运行结果可以看出，在250万这个数量级别的数据存取上，**Hashtable** 和 **SynchronizedMap** 的性能表现几乎一致，毕竟它们的锁实现原理大同小异，而 **ConcurrentHashMap** 表现出了比较大的性能优势，耗时只有前两者的三分之一多一点儿。嗯... 以后面试再被问到相关问题，可以直接把数据甩给面试官... ..

三、结语

好了，以上就是本篇文章的全部内容了，完结撒花。第一次写博客，竟然唠叨了这么多，不足之处还请各位看官老爷不吝赐教。另外想要进一步深入了解 **ConcurrentHashMap** 原理的朋友可以看一下下面两篇文章，是我看过的讲的比较详细的。


[解读Java8中ConcurrentHashMap是如何保证线程安全的](#)

[Java7/8 中的 HashMap 和 ConcurrentHashMap 全解析](#)

阅读 908 • 更新于 2019-11-29

 赞 3

 收藏 3

 分享

本作品系 原创， 采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



lamHYN

 1k

关注作者

0 条评论

得票 • 时间



撰写评论 ...

提交评论

推荐阅读

Java核心（四）你不知道的数据集合

Java中的集合通常指的是Collection下的三个集合框架List、Set、Queue和Map集合，Map并不属于Collection的子集，而是和它平...

[Java中文社群](#) • 阅读 713 • 14 赞

HashMap ConcurrentHashMap

问题描述 翻翻别人的面试经历 这里在知乎上看到的，分享出了自己面试阿里Java岗的面试题。看了一下，除了Spring之外的其他...

[张喜硕](#) • 阅读 5.1k • 4 赞

java中ConcurrentHashMap的使用及在Java 8中的冲突方案

ConcurrentHashMap(简称CHM)是在Java 1.5作为Hashtable的替代选择新引入的，是concurrent包的重要成员。在Java 1.5之前，...

[lSherry](#) • 阅读 5.5k • 2 赞

Java并发编程实战笔记(1)-线程安全简介

转载请注明出处 [链接] 原文排版地址 点击获取更好阅读体验 多线程简介 进程出现的原因 资源利用率 程序在等待操作执行完成的...

[paraller](#) • 阅读 808 • 2 赞

面试必问的几种线程安全的 Map 解析

HashMap线程安全的吗？Java中平时用的最多的Map集合就是HashMap了，它是线程不安全的。看下面两个场景： 1、当用在方...

[Java技术栈](#) • 阅读 538 • 2 赞

JAVA集合框架的特点及实现原理简介

1.集合框架总体架构 集合大致分为Set、List、Queue、Map四种体系,其中List,Set,Queue继承自Collection接口，Map为独立接口...

Yangk · 阅读 393 · 1 赞

面试必问－几种线程安全的Map解析

HashMap线程安全的吗？ Java中平时用的最多的Map集合就是HashMap了，它是线程不安全的。 看下面两个场景： 1、当用在方...

Java技术栈 · 阅读 479 · 1 赞

面试题·HashMap和Hashtable的区别(转载再整理)

HashMap和Hashtable的比较是Java面试中的常见问题，用来考验程序员是否能够正确使用集合类以及是否可以随机应变使用多种...

cmlanche · 阅读 502 · 1 赞

产品

[热门问答](#)

[热门专栏](#)

[热门课程](#)

[最新活动](#)

[技术圈](#)

[酷工作](#)

[移动客户端](#)

课程

[Java 开发课程](#)

[PHP 开发课程](#)

[Python 开发课程](#)

[前端开发课程](#)

[移动开发课程](#)

资源

[每周精选](#)

[用户排行榜](#)

[徽章](#)

[帮助中心](#)

[声望与权限](#)

[社区服务中心](#)

合作

[关于我们](#)

[广告投放](#)

[职位发布](#)

[讲师招募](#)

[联系我们](#)

[合作伙伴](#)

关注

[产品技术日志](#)

[社区运营日志](#)

[市场运营日志](#)

[团队日志](#)

[社区访谈](#)

条款

[服务条款](#)

[隐私政策](#)

[下载 App](#)

Copyright © 2011-2020 SegmentFault.



浙ICP备 15005796号-2 浙公网安备 33010602002000号 杭州堆栈科技有限公司版权所有