Java 理论与实践: 构建一个更好的 HashMap

ConcurrentHashMap 如何在不损失线程安全的同时提供更高的并发性

ConcurrentHashMap 是 Doug Lea 的 util.concurrent 包的一部分，它提供比 Hashtable 或者 synchronizedMap 更高程度的并发性。而且，对于大多数成功的 get() 操作它会设法避免完全锁定，其结果就是使得并发应用程序有着非常好的吞吐量。这个月，Brian Goetz 仔细分析了 ConcurrentHashMap 的代码，并探讨 Doug Lea 是如何在不损失线程安全的情况下取得这么骄人成绩的。请在 讨论论坛 上与作者及其他读者共享您对本文的一些想法（也可以在文章的顶部或底部点击 讨论 来访问论坛）。

Brian Goetz 是一名软件顾问，在过去的 15 年里，他一直是一名专业软件开发人员。他是 Quiotix 的首席顾问，Quiotix 是一家位于加尼福利亚州洛斯拉图斯（Los Altos）市的软件开发与咨询公司。请在业界流行的出版物中查阅 Brian 的 已出版和即将出版的文章。

2003 年 8 月 29 日

在7月份的那期 *Java理论与实践*（“并发集合类”）中，我们简单地回顾了可伸缩性的瓶颈，并讨论了怎么用共享数据结构的方法获得更高的并发性和吞吐量。有时候学习的最好方法是分析专家的成果，所以这个月我们将分析 Doug Lea 的 util.concurrent 包中的 ConcurrentHashMap 的实现。JSR 133 将指定 ConcurrentHashMap 的一个版本，该版本针对 Java 内存模型（JMM）作了优化，它将包含在 JDK 1.5 的 java.util.concurrent 包中。util.concurrent 中的版本在老的和新的内存模型中都已通过线程安全审核。

针对吞吐量进行优化

ConcurrentHashMap 使用了几个技巧来获得高程度的并发以及避免锁定，包括为不同的 hash bucket（桶）使用多个写锁和使用 JMM 的不确定性来最小化锁被保持的时间——或者根本避免获取锁。对于大多数一般用法来说它是经过优化的，这些用法往往会检索一个很可能在 map 中已经存在的值。事实上，多数成功的 get() 操作根本不需要任何锁定就能运行。（警告：不要自己试图这样做！想比 JMM 聪明不像看上去的那么容易。util.concurrent 类是由并发专家编写的，并且在 JMM 安全性方面经过了严格的同行评审。）

多个写锁

我们可以回想一下，Hashtable（或者替代方案 Collections.synchronizedMap）的可伸缩性的主要障碍是它使用了一个 map 范围（map-wide）的锁，为了保证插入、删除或者检索操作的完整性必须保持这样一个锁，而且有时候甚至还要为了保证迭代遍历操作的完整性保持这样一个锁。这样一来，只要锁被保持，就从根本上阻止了其他线程访问 Map，即使处理器有空闲也不能访问，这样大大地限制了并发性。

ConcurrentHashMap 摒弃了单一的 map 范围的锁，取而代之的是由 32 个锁组成的集合，其中每个锁负责保护 hash bucket 的一个子集。锁主要由变化性操作（put() 和 remove()）使用。具有 32 个独立的锁意味着最多可以有 32 个线程可以同时修改 map。这并不一定是说在并发地对 map 进行写操作的线程数少于 32 时，另外的写操作不会被阻塞——32 对于写线程来说是理论上的并发限制数目，但是实际上可能达不到这个值。但是，32 依然比 1 要好得多，而且对于运行于目前这一代的计算机系统上的大多数应用程序来说已经足够了。

map 范围的操作

有 32 个独立的锁，其中每个锁保护 hash bucket 的一个子集，这样需要独占访问 map 的操作就必须获得所有32个锁。一些 map 范围的操作，比如说size() 和 isEmpty()，也许能够不用一次锁整个 map（通过适当地限定这些操作的语义），但是有些操作，比如 map 重排（扩大 hash bucket 的数量，随着 map 的增长重新分布元素），则必须保证独占访问。Java 语言不提供用于获取可变大小的锁集合的简便方法。必须这么做的情况很少见，一旦碰到这种情况，可以用递归方法来实现。

JMM概述

在进入 `put()`、`get()` 和 `remove()` 的实现之前，让我们先简单地看一下 JMM。JMM 掌管着一个线程对内存的动作 （读和写）影响其他线程对内存的动作的方式。由于使用处理器寄存器和预处理 cache 来提高内存访问速度带来的性能提升，Java 语言规范（JLS）允许一些内存操作并不对于所有其他线程立即可见。有两种语言机制可用于保证跨线程内存操作的一致性——`synchronized` 和 `volatile`。

按照 JLS 的说法，“在没有显式同步的情况下，一个实现可以自由地更新主存，更新时所采取的顺序可能是出人意料的。”其意思是说，如果没有同步的话，在一个给定线程中某种顺序的写操作对于另外一个不同的线程来说可能呈现出不同的顺序， 并且对内存变量的更新从一个线程传播到另外一个线程的时间是不可预测的。

虽然使用同步最常见的原因是保证对代码关键部分的原子访问，但实际上同步提供三个独立的功能——原子性、可见性和顺序性。原子性非常简单——同步实施一个可重入的（reentrant）互斥，防止多于一个的线程同时执行由一个给定的监视器保护的代码块。不幸的是，多数文章都只关注原子性方面，而忽略了其他方面。但是同步在 JMM 中也扮演着很重要的角色，会引起 JVM 在获得和释放监视器的时候执行内存壁垒（memory barrier）。

一个线程在获得一个监视器之后，它执行一个*读屏障（read barrier）*——使得缓存在线程局部内存（比如说处理器缓存或者处理器寄存器）中的所有变量都失效，这样就会导致处理器重新从主存中读取同步代码块使用的变量。与此类似，在释放监视器时，线程会执行一个*写屏障（write barrier）*——将所有修改过的变量写回主存。互斥独占和内存壁垒结合使用意味着只要您在程序设计的时候遵循正确的同步法则（也就是说，每当写一个后面可能被其他线程访问的变量，或者读取一个可能最后被另一个线程修改的变量时，都要使用同步），每个线程都会得到它所使用的共享变量的正确的值。

如果在访问共享变量的时候没有同步的话，就会发生一些奇怪的事情。一些变化可能会通过线程立即反映出来，而其他的则需要一些时间（这由关联缓存的本质所致）。结果，如果没有同步您就不能保证内存内

容必定一致（相关的变量相互间可能会不一致），或者不能得到当前的内存内容（一些值可能是过时的）。避免这种危险情况的常用方法（也是推荐使用的方法）当然是正确地使用同步。然而在有些情况下，比如说在像 ConcurrentHashMap 之类的一些使用非常广泛的库类中，在开发过程当中还需要一些额外的专业技能和努力（可能比一般的开发要多出很多倍）来获得较高的性能。

ConcurrentHashMap 实现

如前所述，ConcurrentHashMap 使用的数据结构与 Hashtable 或 HashMap 的实现类似，是 hash bucket 的一个可变数组，每个 ConcurrentHashMap 都由一个 Map.Entry 元素链构成，如清单1所示。与 Hashtable 和 HashMap 不同的是，ConcurrentHashMap 没有使用单一的集合锁（collection lock），而是使用了一个固定的锁池，这个锁池形成了bucket 集合的一个分区。

清单1. ConcurrentHashMap 使用的 Map.Entry 元素

```
protected static class Entry implements Map.Entry {
protected final Object key;
protected volatile Object value;
protected final int hash;
protected final Entry next;
...
}
```

不用锁定遍历数据结构

与 Hashtable 或者典型的锁池 Map 实现不同，ConcurrentHashMap.get() 操作不一定需要获取与相关 bucket 相关联的锁。如果不使用锁定，那么实现必须有能力处理它用到的所有变量的过时的或者不一致的值，比如说列表头指针和 Map.Entry 元素的域（包括组成每个 hash bucket 条目的链表的链接指针）。

大多并发类使用同步来保证独占式访问一个数据结构（以及保持数据结构的一致性）。ConcurrentHashMap 没有采用独占性和一致性，它使用的链表是经过精心设计的，所以其实现可以*检测*到它的列表是否一致或者已经过时。如果它检测到它的列表出现不一致或者过时，或者干脆就找不到它要找的条目，它就会对适当的 bucket 锁进行同步并再次搜索整个链。这样做在一般的情况下可以优化查找，所谓的一般情况是指大多数检索操作是成功的并且检索的次数多于插入和删除的次数。

使用不变性

不一致性的一个重要来源是可以避免得，其方法是使 Entry 元素接近不变性——除了值字段（它们是易变的）之外，所有字段都是 final 的。这就意味着不能将元素添加到 hash 链的中间或末尾，或者从 hash 链的中间或末尾删除元素——而只能从 hash 链的开头添加元素，并且删除操作包括克隆整个链或链的一部分并更新列表的头指针。所以说只要有对某个 hash 链的一个引用，即使可能不知道有没有对列表头节点的引用，您也可以知道列表的其余部分的结构不会改变。而且，因为值字段是易变的，所以能够立即看到对值字段的更新，从而大大简化了编写能够处理内存潜在过时的 Map 的实现。

新的 JMM 为 final 型变量提供初始化安全，而老的 JMM 不提供，这意味着另一个线程看到的可能是 final 字段的默认值，而不是对象的构造方法提供的值。实现必须能够同时检测到这一点，这是通过保证 Entry 中每个字段的默认值不是有效值来实现的。这样构造好列表之后，如果任何一个 Entry 字段有其默认值（零或空），搜索就会失败，提示同步 get() 并再次遍历链。

检索操作

检索操作首先为目标 bucket 查找头指针（是在不锁定的情况下完成的，所以说可能是过时的），然后在不获取 bucket 锁的情况下遍历 bucket 链。如果它不能发现要查找的值，就会同步并试图再次查找条目，如清单2 所示：

清单2. ConcurrentHashMap.get() 实现

```
public Object get(Object key) {
int hash = hash(key); // throws null pointer exception if key is null

// Try first without locking...
Entry[] tab = table;
int index = hash & (tab.length - 1);
Entry first = tab[index];
Entry e;

for (e = first; e != null; e = e.next) {
  if (e.hash == hash && eq(key, e.key)) {
Object value = e.value;
// null values means that the element has been removed
if (value != null)
  return value;
else
  break;
```

```
  }
}

// Recheck under synch if key apparently not there or interference
Segment seg = segments[hash & SEGMENT_MASK];
synchronized(seg) {
  tab = table;
  index = hash & (tab.length - 1);
  Entry newFirst = tab[index];
  if (e != null || first != newFirst) {
for (e = newFirst; e != null; e = e.next) {
  if (e.hash == hash && eq(key, e.key))
return e.value;
}
  }
  return null;
}
  }
```

删除操作

因为一个线程可能看到 hash 链中链接指针的过时的值，简单地从链中删除一个元素不足以保证其他线程在进行查找的时候不继续看到被删除的值。相反，从清单3我们可以看到，删除操作分两个过程——首先找到适当的 `Entry` 对象并把其值字段设为 `null`，然后对链中从头元素到要删除的元素的部分进行克隆，再连接到要删除的元素之后的部分。因为值字段是易变的，如果另外一个线程正在过时的链中查找那个被删除的元素，它会立即看到一个空值，并知道使用同步重新进行检索。最终，原始 hash 链中被删除的元素将会被垃圾收集。

清单3. ConcurrentHashMap.remove() 实现

```
protected Object remove(Object key, Object value) {
/*
  Find the entry, then
1. Set value field to null, to force get() to retry
2. Rebuild the list without this entry.
   All entries following removed node can stay in list, but
   all preceding ones need to be cloned.  Traversals rely
   on this strategy to ensure that elements will not be
  repeated during iteration.
*/

int hash = hash(key);
Segment seg = segments[hash & SEGMENT_MASK];

synchronized(seg) {
  Entry[] tab = table;
  int index = hash & (tab.length-1);
  Entry first = tab[index];
```

```
  Entry e = first;

  for (;;) {
if (e == null)
  return null;
if (e.hash == hash && eq(key, e.key))
  break;
e = e.next;
  }

  Object oldValue = e.value;
  if (value != null && !value.equals(oldValue))
return null;

  e.value = null;

  Entry head = e.next;
  for (Entry p = first; p != e; p = p.next)
head = new Entry(p.hash, p.key, p.value, head);
  tab[index] = head;
  seg.count--;
  return oldValue;
}
  }
```

图1为删除一个元素之前的 hash 链：

图1. Hash链

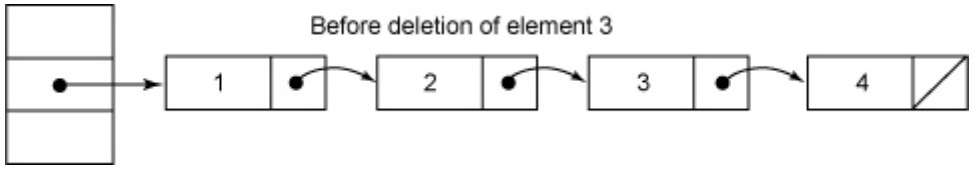
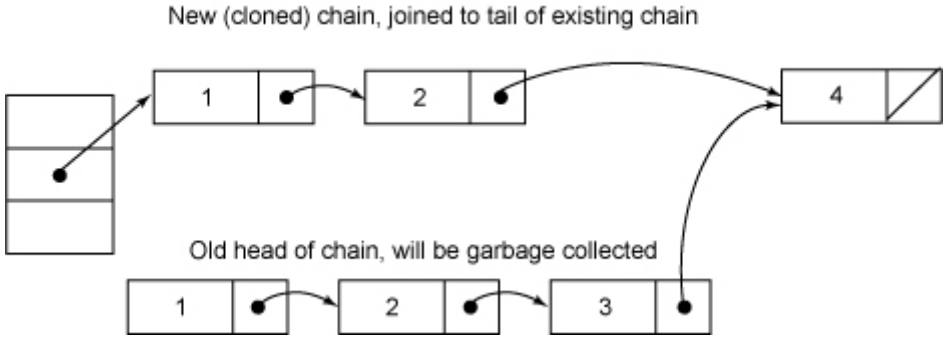


图2为删除元素3之后的链：

图2. 一个元素的删除过程



插入和更新操作

put() 的实现很简单。像 remove() 一样，put() 会在执行期间保持 bucket 锁，但是由于 put() 并不是都需要获取锁，所以这并不一定会阻塞其他读线程的执行（也不会阻塞其他写线程访问别的 bucket）。它首先会在适当的 hash 链中搜索需要的键值。如果能够找到，value字段（易变的）就直接被更新。如果没有找到，新会创建一个用于描述新 map 的新 Entry 对象，然后插入到 bucket 列表的头部。

弱一致的迭代器

由 ConcurrentHashMap 返回的迭代器的语义又不同于 ava.util 集合中的迭代器；而且它又是 *弱一致的（weakly consistent）* 而非 *fail-fast* 的（所谓 fail-fast 是指，当正在使用一个迭代器的时候，如何底层的集合被修改，就会抛出一个异常）。当一个用户调用 keySet().iterator() 去迭代器中检索一组 hash 键的时候，实现就简单地使用同步来保证每个链的头指针是当前值。next()和 hasNext() 操作以一种明显的方式定义，即遍历每个链然后转到下一个链直到所有的链都被遍历。弱一致迭代器可能会也可能不会反映迭代器迭代过程中的插入操作，但是一定会反映迭代器还没有到达的键的更新或删除操作，并且对任何值最多返回一次。ConcurrentHashMap 返回的迭代器不会抛出 ConcurrentModificationException 异常。

动态调整大小

随着 map 中元素数目的增长，hash 链将会变长，因此检索时间也会增加。从某种意义上说，增加 bucket 的数目和重排其中的值是非常重要的。在有些像 Hashtable 之类的类中，这很简单，因为保持一个应用到整个 map 的独占锁是可能的。在 ConcurrentHashMap 中，每次一个条目插入的时候，如果链的长度超过了某个阈值，链就被标记为需要调整大小。当有足够多的链被标记为需要调整大小以后，ConcurrentHashMap 就使用递归获取每个 bucket 上的锁并重排每个 bucket 中的元素到一个新的、更大的 hash 表中。多数情况下，这是自动发生的，并且对调用者透明。

不锁定?

要说不用锁定就可以成功地完成 `get()` 操作似乎有点言过其实，因为 `Entry` 的 `value` 字段是易变的，这是用来检测更新和删除的。在机器级，易变的和同步的内容通常在最后会被翻译成相同的缓存一致原语，所以这里会有 *一些* 锁定，虽然只是细粒度的并且没有调度，或者没有获取和释放监视器的 JVM 开销。但是，除语义之外，在很多通用的情况下，检索的次数大于插入和删除的次数，所以说由 `ConcurrentHashMap` 取得的并发性是相当高的。

结束语

`ConcurrentHashMap` 对于很多并发应用程序来说是一个非常有用的类，而且对于理解 JMM 何以取得较高性能的微妙细节是一个很好的例子。`ConcurrentHashMap`是编码的经典，需要深刻理解并发和 JMM 才能够写得出。使用它，从中学到东西，享受其中的乐趣——但是除非您是 Java 并发方面的专家，否则的话您自己不应该这样试。

参考资料

参加本文的讨论论坛。（还可以点击本文顶部或底部的**讨论**进入论坛。）

阅读 Brian Goetz 撰写的 *Java理论和实践*的整个系列。其中尤其相关的有：

- “并发在一定程度上使一切变得简单”（2002年9月），介绍了 `util.concurrent` 包。
- “变还是不变？”（2003年2月），讨论了线程安全在不变性方面的优点。
- “并发集合类”（2003年7月），分析了可伸缩性的瓶颈以及如何在共享数据结构中获得较高的并发性和吞吐量。

Doug Lea 的 *Concurrent Programming in Java, Second Edition* 是一本围绕有关 Java 应用程序中多线程编程的复杂问题的专著。

Doug Lea 的书的摘录描述了同步的*真正*含义。

下载 util.concurrent 包。

ConcurrentHashMap 的 javadoc 页面非常详细地解释了 ConcurrentHashMap 和 Hashtable 的区别。

查看ConcurrentHashMap 的源代码。

JSR 166 正在标准化 JDK 1.5 的 util.concurrent 库。

Bill Pugh 在 Java Memory Model 上维护了一整套的资源。

JSR 133 发布了针对新的 Java 内存模型进行了优化的一个 ConcurrentHashMap 版本。

在 developerWorks Java 技术专区 上可以找到数百篇其他的 Java 参考资料。