比AtomicLong还高效的LongAdder 源码解析

接触到AtomicLong的原因是在看guava的LoadingCache相关代码时，关于LoadingCache，其实思路也非常简单清晰：用模板模式解决了缓存不命中时获取数据的逻辑，这个思路我早前也正好在项目中使用到。

言归正传，为什么说LongAdder引起了我的注意，原因有二：

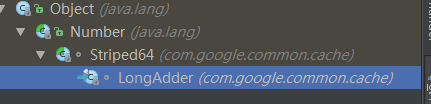
1. 作者是Doug lea ，地位实在举足轻重。

2. 他说这个比AtomicLong高效。

我们知道，AtomicLong已经是非常好的解决方案了，涉及并发的地方都是使用CAS操作，在硬件层次上去做 compare and set操作。效率非常高。

因此，我决定研究下，为什么LongAdder比AtomicLong高效。

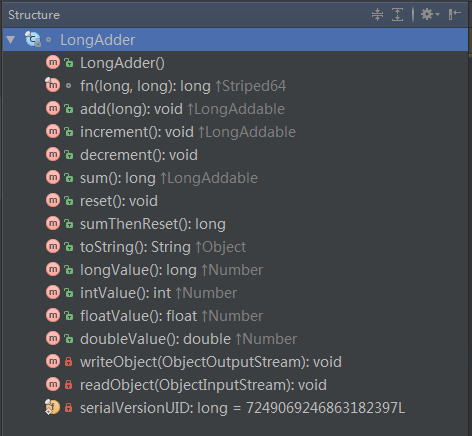
首先，看LongAdder的继承树：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la1.png)**

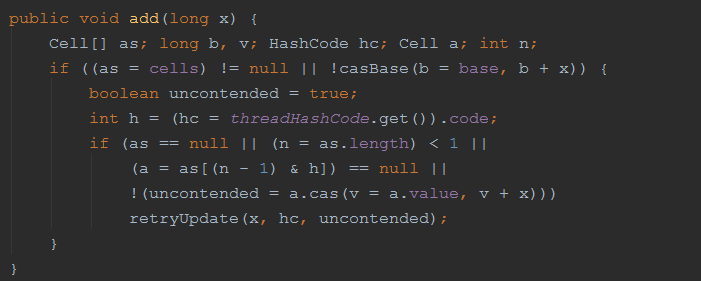
继承自Striped64，这个类包装了一些很重要的内部类和操作。稍候会看到。

**正式开始前，强调下，我们知道，AtomicLong的实现方式是内部有个value 变量，当多线程并发自增，自减时，均通过cas 指令从机器指令级别操作保证并发的原子性。**

**再看看LongAdder的方法：**

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la2.png)**

怪不得可以和AtomicLong作比较，连API都这么像。我们随便挑一个API入手分析，这个API通了，其他API都大同小异，因此，我选择了add这个方法。事实上,其他API也都依赖这个方法。

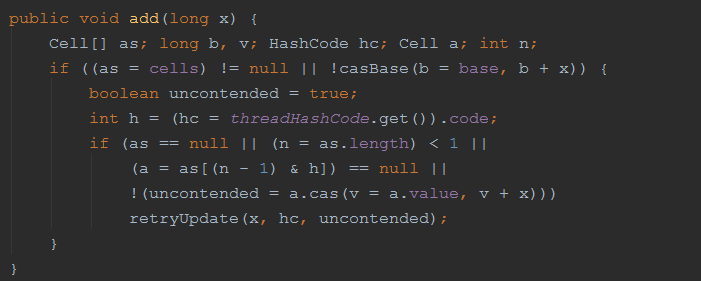
**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la3.png)**

LongAdder中包含了一个Cell 数组，Cell是Striped64的一个内部类，顾名思义，Cell 代表了一个最小单元，这个单元有什么用，稍候会说道。先看定义：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la4.png)**

Cell内部有一个非常重要的value变量，并且提供了一个cas更新其值的方法。

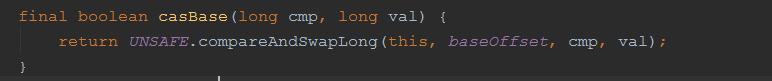
回到add方法：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la3.png)**

这里，我有个疑问，AtomicLong已经使用CAS指令，非常高效了（比起各种锁），LongAdder如果还是用CAS指令更新值，怎么可能比AtomicLong高效了？ 何况内部还这么多判断！！！

这是我开始时最大的疑问，所以，我猜想，难道有比CAS指令更高效的方式出现了？ 带着这个疑问，继续。

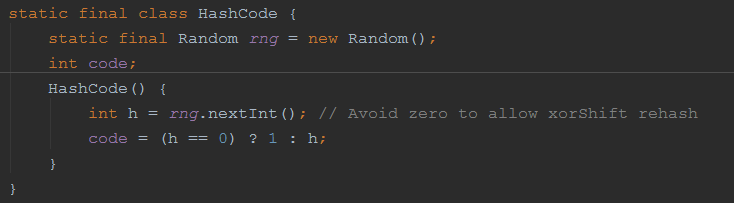
第一if 判断，第一次调用的时候cells数组肯定为null,因此，进入casBase方法：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la5.png)**

原子更新base没啥好说的，如果更新成功，本地调用开始返回，否则进入分支内部。

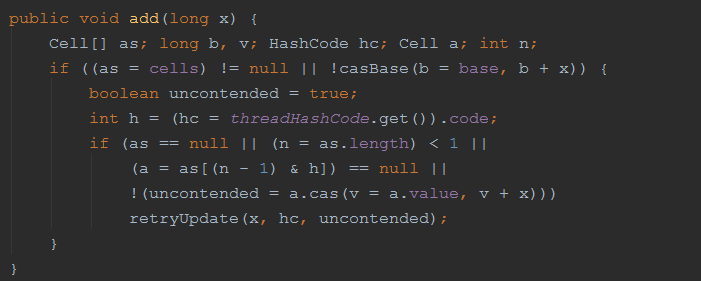
什么时候会更新失败？ 没错，并发的时候，好戏开始了，AtomicLong的处理方式是死循环尝试更新，直到成功才返回，而LongAdder则是进入这个分支。

分支内部，通过一个Threadlocal变量threadHashCode 获取一个HashCode对象，该HashCode对象依然是Striped64类的内部类，看定义：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la6.png)**

有个code变量，保存了一个非0的随机数随机值。

回到add方法：

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la3.png)**

拿到该线程相关的HashCode对象后，获取它的code变量，as[(n-1)h] 这句话相当于对h取模，只不过比起取摸，因为是 与 的运算所以效率更高。

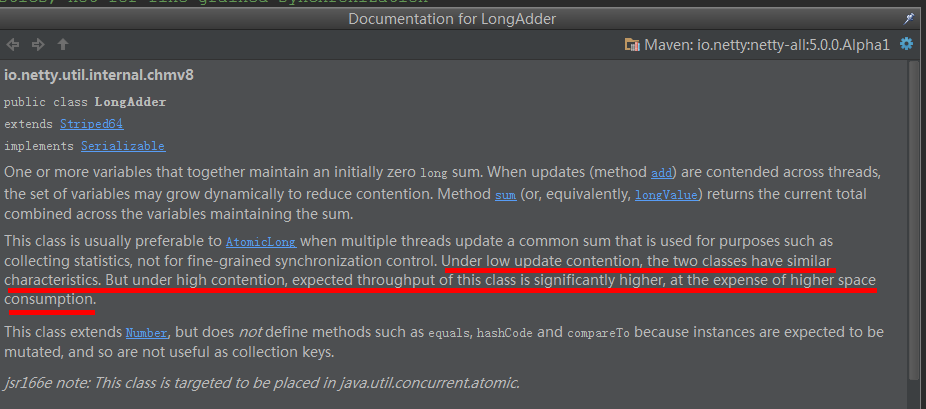
计算出一个在Cells 数组中当先线程的HashCode对应的 索引位置，并将该位置的Cell 对象拿出来更新cas 更新它的value值。

当然，如果as 为null 并且更新失败，才会进入retryUpdate方法。

看到这里我想应该有很多人明白为什么LongAdder会比AtomicLong更高效了，没错，唯一会制约AtomicLong高效的原因是高并发，高并发意味着CAS的失败几率更高， 重试次数更多，越多线程重试，CAS失败几率又越高，变成恶性循环，AtomicLong效率降低。 那怎么解决？ LongAdder给了我们一个非常容易想到的解决方案: 减少并发，将单一value的更新压力分担到多个value中去，降低单个value的 “热度”，分段更新！！！   这样，线程数再多也会分担到多个value上去更新，只需要增加value就可以降低 value的 “热度”  AtomicLong中的 恶性循环不就解决了吗？ cells 就是这个 “段” cell中的value 就是存放更新值的， 这样，当我需要总数时，把cells 中的value都累加一下不就可以了么！！

**当然，聪明之处远远不仅仅这里，在看看add方法中的代码，casBase方法可不可以不要，直接分段更新,上来就计算 索引位置，然后更新value？**

**答案是不好，不是不行，因为，casBase操作等价于AtomicLong中的cas操作，要知道，LongAdder这样的处理方式是有坏处的，分段操作必然带来空间上的浪费，可以空间换时间，但是，能不换就不换，看空间时间都节约~！ 所以，casBase操作保证了在低并发时，不会立即进入分支做分段更新操作，因为低并发时，casBase操作基本都会成功，只有并发高到一定程度了，才会进入分支，所以，Doug Lead对该类的说明是： 低并发时LongAdder和AtomicLong性能差不多，高并发时LongAdder更高效！**

**[](http://www.liuinsect.com/wp-content/uploads/2014/04/la7.png)**

但是，Doung Lea 还是没这么简单，聪明之处还没有结束……

如此，retryUpdate中做了什么事，也基本略知一二了，因为cell中的value都更新失败(说明该索引到这个cell的线程也很多，并发也很高时) 或者cells数组为空时才会调用retryUpdate,

因此，retryUpdate里面应该会做两件事：

1. 扩容，将cells数组扩大，降低每个cell的并发量，同样，这也意味着cells数组的rehash动作。

2. 给空的cells变量赋一个新的Cell数组。

是不是这样呢？ 继续看代码：

代码比较长，变成文本看看，为了方便大家看if else 分支，对应的  { } 我用相同的颜色标注出来

可以看到，这个时候Doug Lea才愿意使用死循环保证更新成功~！

final void retryUpdate**(**long x**,** HashCode hc**,** boolean wasUncontended**)** **{**

int h **=** hc**.**code**;**

boolean collide **=** **false;** // True if last slot nonempty

**for** **(;;)** **{**

Cell**[]** as**;** Cell a**;** int n**;** long v**;**

**if** **((**as **=** cells**)** **!=** **null** **&&** **(**n **=** as**.**length**)** **>** 0**)** **{**// 分支1

**if** **((**a **=** as**[(**n **-** 1**)** **&** h**])** **==** **null)** **{**

**if** **(**busy **==** 0**)** **{** // Try to attach new Cell

Cell r **=** **new** Cell**(**x**);** // Optimistically create

**if** **(**busy **==** 0 **&&** casBusy**())** **{**

boolean created **=** **false;**

**try** **{** // Recheck under lock

Cell**[]** rs**;** int m**,** j**;**

**if** **((**rs **=** cells**)** **!=** **null** **&&**

**(**m **=** rs**.**length**)** **>** 0 **&&**

rs**[**j **=** **(**m **-** 1**)** **&** h**]** **==** **null)** **{**

rs**[**j**]** **=** r**;**

created **=** **true;**

**}**

**}** **finally** **{**

busy **=** 0**;**

**}**

**if** **(**created**)**

**break;**

**continue;** // Slot is now non-empty

**}**

**}**

collide **=** **false;**

**}**

**else** **if** **(!**wasUncontended**)** // CAS already known to fail

wasUncontended **=** **true;** // Continue after rehash

**else** **if** **(**a**.**cas**(**v **=** a**.**value**,** fn**(**v**,** x**)))**

**break;**

**else** **if** **(**n **>=** NCPU **||** cells **!=** as**)**

collide **=** **false;** // At max size or stale

**else** **if** **(!**collide**)**

collide **=** **true;**

**else** **if** **(**busy **==** 0 **&&** casBusy**())** **{**

**try** **{**

**if** **(**cells **==** as**)** **{** // Expand table unless stale

Cell**[]** rs **=** **new** Cell**[**n **<<** 1**];**

**for** **(**int i **=** 0**;** i **<** n**;** **++**i**)**

rs**[**i**]** **=** as**[**i**];**

cells **=** rs**;**

**}**

**}** **finally** **{**

busy **=** 0**;**

**}**

collide **=** **false;**

**continue;** // Retry with expanded table

**}**

h **^=** h **<<** 13**;** // Rehash h ^= h >>> 17;

h **^=** h **<<** 5**;**

**}**

**else** **if** **(**busy **==** 0 **&&** cells **==** as **&&** casBusy**())** **{**//分支2

boolean init **=** **false;**

**try** **{** // Initialize table

**if** **(**cells **==** as**)** **{**

Cell**[]** rs **=** **new** Cell**[**2**];**

rs**[**h **&** 1**]** **=** **new** Cell**(**x**);**

cells **=** rs**;**

init **=** **true;**

**}**

**}** **finally** **{**

busy **=** 0**;**

**}**

**if** **(**init**)**

**break;**

**}**

**else** **if** **(**casBase**(**v **=** base**,** fn**(**v**,** x**)))**

**break;** // Fall back on using base

**}**

hc**.**code **=** h**;** // Record index for next time

**}**

分支2中，为cells为空的情况，需要new 一个Cell数组。

分支1分支中，略复杂一点点：

注意，几个分支中都提到了busy这个方法，这个可以理解为一个cas实现的锁，只有在需要更新cells数组的时候才会更新该值为1，如果更新失败，则说明当前有线程在更新cells数组，当前线程需要等待。重试。

回到分支1中，这里首先判断当前cells数组中的索引位置的cell元素是否为空，如果为空，则添加一个cell到数组中。

否则更新 标示冲突的标志位wasUncontended 为 true ，重试。

否则，再次更新cell中的value,如果失败，重试。

。。。。。。。一系列的判断后，如果还是失败，下下下策，reHash,直接将cells数组扩容一倍，并更新当前线程的hash值，保证下次更新能尽可能成功。

**可以看到，LongAdder确实用了很多心思减少并发量，并且，每一步都是在”没有更好的办法“的时候才会选择更大开销的操作，从而尽可能的用最最简单的办法去完成操作。追求简单，但是绝对不粗暴。**

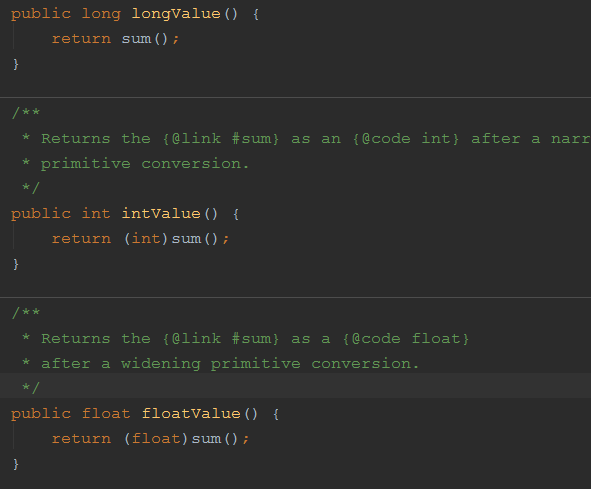
————————————————分割线——————————————————————-

昨天和左耳朵耗子简单讨论了下，发现左耳朵耗子对读者思维的引导还是非常不错的，在第一次发现这个类后，对里面的实现又提出了更多的问题，引导大家思考，值得学习。

我们 发现的问题有这么几个：

1. jdk 1.7中是不是有这个类？  
我确认后，结果如下：    jdk-7u51 版本上还没有  但是jdk-8u20版本上已经有了。代码基本一样 ，增加了对double类型的支持和删除了一些冗余的代码。有兴趣的同学可以去下载下JDK 1.8看看

2. base有没有参与汇总？  
base在调用intValue等方法的时候是会汇总的：

[](http://www.liuinsect.com/wp-content/uploads/2014/04/LA101.bmp)  
3. base的顺序可不可以调换?  
左耳朵耗子,提出了这么一个问题： 在add方法中，如果cells不会为空后，casBase方法一直都没有用了？

因此，我想可不可以调换add方法中的判断顺序，比如，先做casBase的判断，结果是 不调换可能更好，调换后每次都要CAS一下，在高并发时，失败几率非常高，并且是恶性循环，比起一次判断，后者的开销明显小很多，还没有副作用。因此，不调换可能会更好。

4. AtomicLong可不可以废掉？  
我的想法是可以废掉了，因为，虽然LongAdder在空间上占用略大，但是，它的性能已经足以说明一切了,无论是从节约空的角度还是执行效率上，AtomicLong基本没有优势了，具体看这个测试（感谢coolshell读者[Lemon](http://lianming.info/)的回复）:http://blog.palominolabs.com/2014/02/10/java-8-performance-improvements-longadder-vs-atomiclong/