[**Java集合---ConcurrentHashMap原理分析**](http://www.cnblogs.com/ITtangtang/p/3948786.html)

**一、背景：**

**线程不安全的HashMap**

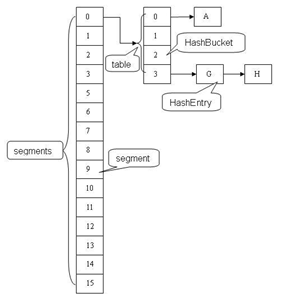
    因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

**效率低下的HashTable容器**

     HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

**锁分段技术**

    HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

**二、应用场景**

    当有一个大数组时需要在多个线程共享时就可以考虑是否把它给分层多个节点了，避免大锁。并可以考虑通过hash算法进行一些模块定位。

其实不止用于线程，当设计数据表的事务时（事务某种意义上也是同步机制的体现），可以把一个表看成一个需要同步的数组，如果操作的表数据太多时就可以考虑事务分离了（这也是为什么要避免大表的出现），比如把数据进行字段拆分，水平分表等.

**三、源码解读**

    ConcurrentHashMap(1.7及之前)中主要实体类就是三个：ConcurrentHashMap（整个Hash表）,Segment（桶），HashEntry（节点），对应上面的图可以看出之间的关系

/\*\*

\* The segments, each of which is a specialized hash table

\*/

final Segment<K,V>[] segments;

**不变(Immutable)和易变(Volatile)**

    ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。HashEntry代表每个hash链中的一个节点，其结构如下所示：

[复制代码](javascript:void(0);)

static final class HashEntry<K,V> {

final K key;

final int hash;

volatile V value;

final HashEntry<K,V> next;

}

[复制代码](javascript:void(0);)

　　可以看到除了value不是final的，其它值都是final的，这意味着不能从hash链的中间或尾部添加或删除节点，因为这需要修改next 引用值，所有的节点的修改只能从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除结点的下一个结点。这在讲解删除操作时还会详述。为了确保读操作能够看到最新的值，将value设置成volatile，这避免了加锁。

**其它**

　　为了加快定位段以及段中hash槽的速度，每个段hash槽的的个数都是2^n，这使得通过位运算就可以定位段和段中hash槽的位置。当并发级别为默认值16时，也就是段的个数，hash值的高4位决定分配在哪个段中。但是我们也不要忘记《算法导论》给我们的教训：hash槽的的个数不应该是 2^n，这可能导致hash槽分配不均，这需要对hash值重新再hash一次。（这段似乎有点多余了 ）

**定位操作：**

final Segment<K,V> segmentFor(int hash) {

return segments[(hash >>> segmentShift) & segmentMask];

}

　　既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到Segment。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再哈希。

再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的Segment上，从而提高容器的存取效率。假如哈希的质量差到极点，那么所有的元素都在一个Segment中，不仅存取元素缓慢，分段锁也会失去意义。我做了一个测试，不通过再哈希而直接执行哈希计算。

System.out.println(Integer.parseInt("0001111", 2) & 15);

System.out.println(Integer.parseInt("0011111", 2) & 15);

System.out.println(Integer.parseInt("0111111", 2) & 15);

System.out.println(Integer.parseInt("1111111", 2) & 15);

计算后输出的哈希值全是15，通过这个例子可以发现如果不进行再哈希，哈希冲突会非常严重，因为只要低位一样，无论高位是什么数，其哈希值总是一样。我们再把上面的二进制数据进行再哈希后结果如下，为了方便阅读，不足32位的高位补了0，每隔四位用竖线分割下。

0100｜0111｜0110｜0111｜1101｜1010｜0100｜1110

1111｜0111｜0100｜0011｜0000｜0001｜1011｜1000

0111｜0111｜0110｜1001｜0100｜0110｜0011｜1110

1000｜0011｜0000｜0000｜1100｜1000｜0001｜1010

可以发现每一位的数据都散列开了，通过这种再哈希能让数字的每一位都能参加到哈希运算当中，从而减少哈希冲突。ConcurrentHashMap通过以下哈希算法定位segment。

默认情况下segmentShift为28，segmentMask为15，再哈希后的数最大是32位二进制数据，向右无符号移动28位，意思是让高4位参与到hash运算中， (hash >>> segmentShift) & segmentMask的运算结果分别是4，15，7和8，可以看到hash值没有发生冲突。

final Segment<K,V> segmentFor(int hash) {

return segments[(hash >>> segmentShift) & segmentMask];

}

**数据结构**

　　所有的成员都是final的，其中segmentMask和segmentShift主要是为了定位段，参见上面的segmentFor方法。

　　关于Hash表的基础数据结构，这里不想做过多的探讨。Hash表的一个很重要方面就是如何解决hash冲突，ConcurrentHashMap 和HashMap使用相同的方式，都是将hash值相同的节点放在一个hash链中。与HashMap不同的是，ConcurrentHashMap使用多个子Hash表，也就是段(Segment)。

每个Segment相当于一个子Hash表，它的数据成员如下：

[复制代码](javascript:void(0);)

static final class Segment<K,V> extends ReentrantLock implements Serializable {

/\*\*

\* The number of elements in this segment's region.

\*/

transient volatileint count;

/\*\*

\* Number of updates that alter the size of the table. This is

\* used during bulk-read methods to make sure they see a

\* consistent snapshot: If modCounts change during a traversal

\* of segments computing size or checking containsValue, then

\* we might have an inconsistent view of state so (usually)

\* must retry.

\*/

transient int modCount;

/\*\*

\* The table is rehashed when its size exceeds this threshold.

\* (The value of this field is always <tt>(int)(capacity \*

\* loadFactor)</tt>.)

\*/

transient int threshold;

/\*\*

\* The per-segment table.

\*/

transient volatile HashEntry<K,V>[] table;

/\*\*

\* The load factor for the hash table. Even though this value

\* is same for all segments, it is replicated to avoid needing

\* links to outer object.

\* @serial

\*/

final float loadFactor;

}

[复制代码](javascript:void(0);)

　　count用来统计该段数据的个数，它是volatile，它用来协调修改和读取操作，以保证读取操作能够读取到几乎最新的修改。协调方式是这样的，每次修改操作做了结构上的改变，如增加/删除节点(修改节点的值不算结构上的改变)，都要写count值，每次读取操作开始都要读取count的值。这利用了 Java 5中对volatile语义的增强，对同一个volatile变量的写和读存在happens-before关系。modCount统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变，在讲述跨段操作时会还会详述。threashold用来表示需要进行rehash的界限值。table数组存储段中节点，每个数组元素是个hash链，用HashEntry表示。table也是volatile，这使得能够读取到最新的 table值而不需要同步。loadFactor表示负载因子。

**删除操作remove(key)**

public V remove(Object key) {

hash = hash(key.hashCode());

return segmentFor(hash).remove(key, hash, null);

}

   整个操作是先定位到段，然后委托给段的remove操作。当多个删除操作并发进行时，只要它们所在的段不相同，它们就可以同时进行。

下面是Segment的remove方法实现：

[复制代码](javascript:void(0);)

V remove(Object key, int hash, Object value) {

lock();

try {

int c = count - 1;

HashEntry<K,V>[] tab = table;

int index = hash & (tab.length - 1);

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while (e != null && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue = null;

if (e != null) {

V v = e.value;

if (value == null || value.equals(v)) {

oldValue = v;

// All entries following removed node can stay

// in list, but all preceding ones need to be

// cloned.

++modCount;

HashEntry<K,V> newFirst = e.next;

\*for (HashEntry<K,V> p = first; p != e; p = p.next)

\*newFirst = new HashEntry<K,V>(p.key, p.hash,

newFirst, p.value);

tab[index] = newFirst;

count = c; // write-volatile

}

}

return oldValue;

} finally {

unlock();

}

}

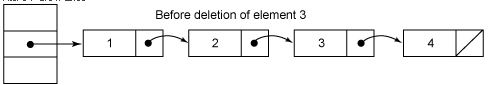
[复制代码](javascript:void(0);)

　　整个操作是在持有段锁的情况下执行的，空白行之前的行主要是定位到要删除的节点e。接下来，如果不存在这个节点就直接返回null，否则就要将e前面的结点复制一遍，尾结点指向e的下一个结点。e后面的结点不需要复制，它们可以重用。

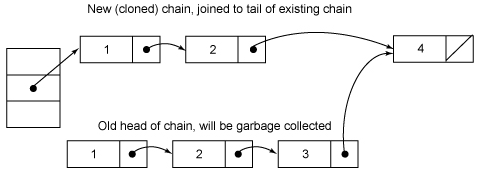
中间那个for循环是做什么用的呢？（\*号标记）从代码来看，就是将定位之后的所有entry克隆并拼回前面去，但有必要吗？每次删除一个元素就要将那之前的元素克隆一遍？这点其实是由entry的不变性来决定的，仔细观察entry定义，发现除了value，其他所有属性都是用final来修饰的，这意味着在第一次设置了next域之后便不能再改变它，取而代之的是将它之前的节点全都克隆一次。至于entry为什么要设置为不变性，这跟不变性的访问不需要同步从而节省时间有关

下面是个示意图

**删除元素之前：**



**删除元素3之后：**



　　第二个图其实有点问题，复制的结点中应该是值为2的结点在前面，值为1的结点在后面，也就是刚好和原来结点顺序相反，还好这不影响我们的讨论。

整个remove实现并不复杂，但是需要注意如下几点。第一，当要删除的结点存在时，删除的最后一步操作要将count的值减一。这必须是最后一步操作，否则读取操作可能看不到之前对段所做的结构性修改。第二，remove执行的开始就将table赋给一个局部变量tab，这是因为table是 volatile变量，读写volatile变量的开销很大。编译器也不能对volatile变量的读写做任何优化，直接多次访问非volatile实例变量没有多大影响，编译器会做相应优化。

**get操作**

ConcurrentHashMap的get操作是直接委托给Segment的get方法，直接看Segment的get方法：

[复制代码](javascript:void(0);)

V get(Object key, int hash) {

if (count != 0) { // read-volatile 当前桶的数据个数是否为0

HashEntry<K,V> e = getFirst(hash); 得到头节点

while (e != null) {

if (e.hash == hash && key.equals(e.key)) {

V v = e.value;

if (v != null)

return v;

return readValueUnderLock(e); // recheck

}

e = e.next;

}

}

returnnull;

}

[复制代码](javascript:void(0);)

get操作不需要锁。

　　除非读到的值是空的才会加锁重读，我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile

　　第一步是访问count变量，这是一个volatile变量，由于所有的修改操作在进行结构修改时都会在最后一步写count 变量，通过这种机制保证get操作能够得到几乎最新的结构更新。对于非结构更新，也就是结点值的改变，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。

　　接下来就是根据hash和key对hash链进行遍历找到要获取的结点，如果没有找到，直接访回null。对hash链进行遍历不需要加锁的原因在于链指针next是final的。但是头指针却不是final的，这是通过getFirst(hash)方法返回，也就是存在 table数组中的值。这使得getFirst(hash)可能返回过时的头结点，例如，当执行get方法时，刚执行完getFirst(hash)之后，另一个线程执行了删除操作并更新头结点，这就导致get方法中返回的头结点不是最新的。这是可以允许，通过对count变量的协调机制，get能读取到几乎最新的数据，虽然可能不是最新的。要得到最新的数据，只有采用完全的同步。

　　最后，如果找到了所求的结点，判断它的值如果非空就直接返回，否则在有锁的状态下再读一次。这似乎有些费解，理论上结点的值不可能为空，这是因为 put的时候就进行了判断，如果为空就要抛NullPointerException。空值的唯一源头就是HashEntry中的默认值，因为 HashEntry中的value不是final的，非同步读取有可能读取到空值。仔细看下put操作的语句：tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry构造函数中对value的赋值以及对tab[index]的赋值可能被重新排序，这就可能导致结点的值为空。这里当v为空时，可能是一个线程正在改变节点，而之前的get操作都未进行锁定，根据bernstein条件，读后写或写后读都会引起数据的不一致，所以这里要对这个e重新上锁再读一遍，以保证得到的是正确值。

[复制代码](javascript:void(0);)

V readValueUnderLock(HashEntry<K,V> e) {

lock();

try {

return e.value;

} finally {

unlock();

}

}

[复制代码](javascript:void(0);)

　　如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景

**put操作**

同样地put操作也是委托给段的put方法。下面是段的put方法：

[复制代码](javascript:void(0);)

V put(K key, int hash, V value, boolean onlyIfAbsent) {

lock();

try {

int c = count;

if (c++ > threshold) // ensure capacity

rehash();

HashEntry<K,V>[] tab = table;

int index = hash & (tab.length - 1);

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

while (e != null && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

if (e != null) {

oldValue = e.value;

if (!onlyIfAbsent)

e.value = value;

}

else {

oldValue = null;

++modCount;

tab[index] = new HashEntry<K,V>(key, hash, first, value);

count = c; // write-volatile

}

return oldValue;

} finally {

unlock();

}

}

[复制代码](javascript:void(0);)

　　该方法也是在持有段锁(锁定整个segment)的情况下执行的，这当然是为了并发的安全，修改数据是不能并发进行的，必须得有个判断是否超限的语句以确保容量不足时能够rehash。接着是找是否存在同样一个key的结点，如果存在就直接替换这个结点的值。否则创建一个新的结点并添加到hash链的头部，这时一定要修改modCount和count的值，同样修改count的值一定要放在最后一步。put方法调用了rehash方法，reash方法实现得也很精巧，主要利用了table的大小为2^n，这里就不介绍了。而比较难懂的是这句int index = hash & (tab.length - 1)，原来segment里面才是真正的hashtable，即每个segment是一个传统意义上的hashtable,如上图，从两者的结构就可以看出区别，这里就是找出需要的entry在table的哪一个位置，之后得到的entry就是这个链的第一个节点，如果e!=null，说明找到了，这是就要替换节点的值（onlyIfAbsent == false），否则，我们需要new一个entry，它的后继是first，而让tab[index]指向它，什么意思呢？实际上就是将这个新entry插入到链头，剩下的就非常容易理解了

　　由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

* 是否需要扩容。在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。
* 如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

另一个操作是containsKey，这个实现就要简单得多了，因为它不需要读取值：

[复制代码](javascript:void(0);)

boolean containsKey(Object key, int hash) {

if (count != 0) { // read-volatile

HashEntry<K,V> e = getFirst(hash);

while (e != null) {

if (e.hash == hash && key.equals(e.key))

returntrue;

e = e.next;

}

}

returnfalse;

}

[复制代码](javascript:void(0);)

**size()操作**

　　如果我们要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，我们是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是拿到之后可能累加前使用的count发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计size的时候把所有Segment的put，remove和clean方法全部锁住，但是这种做法显然非常低效。

　　因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

　　那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put , remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。