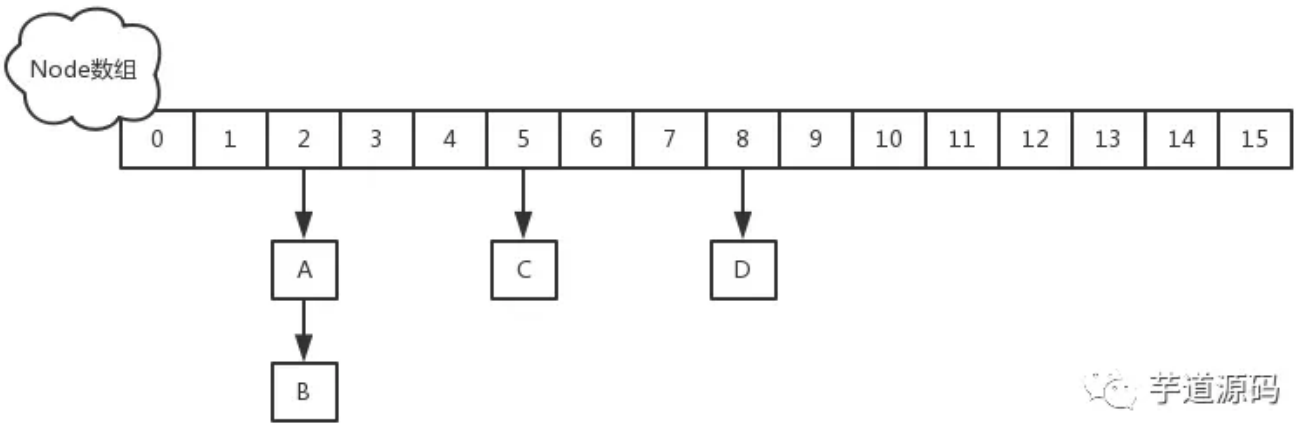
经典面试题：为什么 ConcurrentHashMap 的读操作不需要加锁?

我们知道，ConcurrentHashmap(1.8)这个并发集合框架是线程安全的，当你看到源码的get操作时，会发现get操作全程是没有加任何锁的，这也是这篇博文讨论的问题——为什么它不需要加锁呢?

ConcurrentHashMap的简介

> 我想有基础的同学知道在jdk1.7中是采用Segment + HashEntry + ReentrantLock的方式进行实现的，而1.8中放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现。

- JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）
- JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了
- JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档



img

get操作源码

- 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
- 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
- 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

```
//会发现源码中没有一处加了锁
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) {//读取首节点的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=-1，说明该节点是一个ForwardingNode，正在迁移，此时调用ForwardingNode的find方法去nextT
        //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
```

```
            //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;
    }
```

> get没有加锁的话，ConcurrentHashMap是如何保证读到的数据不是脏数据的呢?

volatile登场

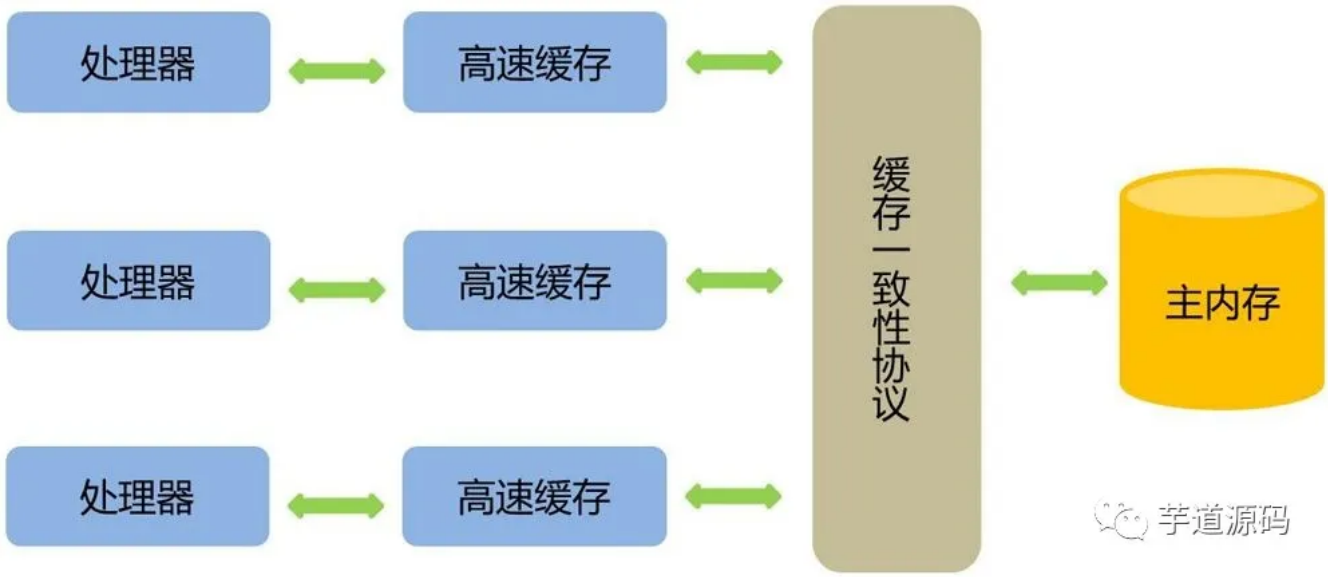
对于可见性，Java提供了volatile关键字来保证可见性、有序性。但不保证原子性。

普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

- volatile关键字对于基本类型的修改可以在随后对多个线程的读保持一致，但是对于引用类型如数组，实体bean，仅仅保证引用的可见性，但并不保证引用内容的可见性。。
- 禁止进行指令重排序。

背景：为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存（L1，L2或其他）后再进行操作，但操作完不知道何时会写到内存。

- 如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条指令，将这个变量所在缓存行的数据写回到系统内存。但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。
- 在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，当某个CPU在写数据时，如果发现操作的变量是共享变量，则会通知其他CPU告知该变量的缓存行是无效的，因此其他CPU在读取该变量时，发现其无效会重新从主存中加载数据。



img

总结下来：

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量的缓存行无效，所以线程1再次读取变量的值时会去主存读取。

是加在数组上的volatile吗?

```
/**
     * The array of bins. Lazily initialized upon first insertion.
     * Size is always a power of two. Accessed directly by iterators.
     */
    transient volatile Node<K,V>[] table;
```

我们知道volatile可以修饰数组的，只是意思和它表面上看起来的样子不同。举个栗子，volatile int array[10]是指array的地址是volatile的而不是数组元素的值是volatile的.

用volatile修饰的Node

get操作可以无锁是由于Node的元素val和指针next是用volatile修饰的，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;
    //可以看到这些都用了volatile修饰
    volatile V val;
    volatile Node<K,V> next;

    Node(int hash, K key, V val, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.val = val;
        this.next = next;
    }

    public final K getKey()       { return key; }
    public final V getValue()     { return val; }
    public final int hashCode()   { return key.hashCode() ^ val.hashCode(); }
    public final String toString(){ return key + "=" + val; }
    public final V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    public final boolean equals(Object o) {
        Object k, v, u; Map.Entry<?,?> e;
        return ((o instanceof Map.Entry) &&
                (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
                (v = e.getValue()) != null &&
                (k == key || k.equals(key)) &&
                (v == (u = val) || v.equals(u)));
    }

    /**
     * Virtualized support for map.get(); overridden in subclasses.
     */
    Node<K,V> find(int h, Object k) {
        Node<K,V> e = this;
        if (k != null) {
            do {
                K ek;
                if (e.hash == h &&
                    ((ek = e.key) == k || (ek != null && k.equals(ek))))
                    return e;
```

```
            } while ((e = e.next) != null);
        }
        return null;
    }
}
```

> 既然volatile修饰数组对get操作没有效果那加在数组上的volatile的目的是什么呢?

其实就是为了使得Node数组在扩容的时候对其他线程具有可见性而加的volatile

总结

- 在1.8中ConcurrentHashMap的get操作全程不需要加锁，这也是它比其他并发集合比如hashtable、用Collections.synchronizedMap()包装的hashmap;安全效率高的原因之一。
- get操作全程不需要加锁是因为Node的成员val是用volatile修饰的和数组用volatile修饰没有关系。
- 数组用volatile修饰主要是保证在数组扩容的时候保证可见性。
