ConcurrentHashMap 的出现的原因和一些原理

概述

历史出现原因和 CAS ABA

ConcurrentHashMap 这个类在 java.lang.current 包中，这个包中的类都是线程安全的。
ConcurrentHashMap 底层存储数据的结构与 1.8 的 HashMap 是一样的，都是数组+链表（或红黑树）的结构。
在日常的开发中，我们最长用到的键值对存储结构的是 HashMap，但是我们知道，这个类是非线程安全的，在高并发的场景下，在进行 put 操作的时候有可能进入死循环从而使服务器的 cpu 使用率达到 100%；
sun 公司因此也给出了与之对应的线程安全的类。在 jdk1.5 以前，使用的是 HashTable，这个类为了保证线程安全，在每个类中都添加了 synchronized 关键字，而想而知在高并发的情景下相率是非常低下的。
为了解决 HashTable 效率低下的问题，官网在 jdk1.5 后推出了 ConcurrentHashMap 来替代饱受诟病的 HashTable。
jdk1.5 后 ConcurrentHashMap 使用了分段锁的技术。在整个数组中被分为多个 segment，每次 get，put，remove 操作时就锁住目标元素所在的 segment 中，因此 segment 与 segment 之前是可以并发操作的，
上述就是 jdk1.5 后实现线程安全的大致思想。
但是，从描述中可以看出一个问题，就是如果出现比较极端的情况，所有的数据都集中在一个 segment 中的话，在并发的情况下相当于锁住了全表，这种情况下其实是和 HashTable 的效率出不多的，
但总体来说相较于 HashTable，效率还是有了很大的提升。
jdk1.8 后，ConcurrentHashMap 摒弃了 segment 的思想，转而使用 cas+synchronized 组合的方式来实现并发下的线程安全的，这种实现方式比 1.5 的效率又有了比较大的提升。

https://zhuanlan.zhihu.com/p/63629645

三、重要成员变量

1、ziseCtr：在多个方法中出现过这个变量，该变量主要是用来控制数组的初始化和扩容的，默认值为 0，可以概括一下 4 种状态：

- a、sizeCtr=0：默认值；
- b、sizeCtr=-1：表示 Map 正在初始化中；
- c、sizeCtr=-N：表示正在有 N-1 个线程进行扩容操作；
- d、sizeCtr>0: 未初始化则表示初始化 Map 的大小，已初始化则表示下次进行扩容操作的阈值；

2、table：**用于存储链表或红黑数的数组，初始值为 null，在第一次进行 put 操作的时候进行初始化，默认值为 16；

3、nextTable：**在扩容时新生成的数组，其大小为当前 table 的 2 倍，用于存放 table 转移过来的值；

4、Node：**该类存储数据的核心，以 key-value 形式来存储；

5、ForwardingNode：**这是一个特殊 Node 节点，仅在进行扩容时用作占位符，表示当前位置已被移动或者为 null，该 node 节点的 hash 值为-1；

四、put 操作

先把源码摆上来:

```
/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, Boolean onlyIfAbsent) {
    //key和value不能为空
    if (key == null || value == null) throw new NullPointerException();
    //通过key来计算获得hash值
    int hash = spread(key.hashCode());
    //用于计算数组位置上存放的node的节点数量
    //在put完成后会对这个参数判断是否需要转换成红黑树或链表
    int binCount = 0;
    //使用自旋的方式放入数据
    //这个过程是非阻塞的，放入失败会一直循环尝试，直至成功
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f;
        int n, i, fh;
        //第一次put操作，对数组进行初始化，实现懒加载
        if (tab == null || (n = tab.length) == 0)
                        //初始化
        tab = initTable();
        //数组已初始化完成后
        //使用cas来获取插入元素所在的数组的下标的位置，该位置为空的话就直接放进去 else if ((f = tabAt(tab, i = (n - 1)
            if (casTabAt(tab, i, null,
                                        new Node<K,V>(hash, key, value, null)))
                                break;
            // no lock when adding to empty bin
        }
        //hash=-1,表明该位置正在进行扩容操作，让当前线程也帮助该位置上的扩容，并发扩容提高扩容的速度 else if ((fh = f.ha
                        //帮助扩容
        tab = helpTransfer(tab, f);
        //插入到该位置已有数据的节点上，即用hash冲突
        //在这里为保证线程安全，会对当前数组位置上的第一个节点进行加锁，因此其他位置上
        //仍然可以进行插入，这里就是jdk1.8相较于之前版本使用segment作为锁性能要高效的地方 else {
            V oldVal = null;
            synchronized (f) {
                //再一次判断f节点是否为第一个节点，防止其他线程已修改f节点
                if (tabAt(tab, i) == f) {
                    //为链表
                    if (fh >= 0) {
                        binCount = 1;
                        //将节点放入链表中
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                                                ((ek = e.key) == key ||
                                                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                                                        e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
```

```
                            pred.next = new Node<K,V>(hash, key,
                                                      value, null);
                            break;
                        }
                    }
                }
                //为红黑树 else if (f instanceof TreeBin) {
                    Node<K,V> p;
                    binCount = 2;
                    //将节点插入红黑树中
                    if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                          value)) != null) {
                        oldVal = p.val;
                        if (!onlyIfAbsent)
                                                    p.val = value;
                    }
                }
            }
        }
        //插入成功后判断插入数据所在位置上的节点数量，
        //如果数量达到了转化红黑树的阈值，则进行转换
        if (binCount != 0) {
            if (binCount >= TREEIFY_THRESHOLD)
                                //由链表转换成红黑树
            treeifyBin(tab, i);
            if (oldVal != null)
                                return oldVal;
            break;
        }
    }
}
//使用cas统计数量增加1，同时判断是否满足扩容需求，进行扩容
addCount(1L, binCount);
return null;
}
```

在代码上写注释可能看得不是很清晰，那么我就使用文字再来描述一下插入数据的整个流程：

所以，put 操作流程可以简单的概括为上面的六个步骤，其中一些具体的操作会在下面进行详细的说明，不过，值得注意的是：

- ConcurrentHashMap 不可以存储 key 或 value 为 null 的数据，有别于 HashMap；
- ConcurrentHashMap 使用了懒加载的方式初始化数据，把 table 的初始化放在第一次 put 数据的时候，而不是在 new 的时候；
- 扩容时是支持并发扩容，这将有助于减少扩容的时间，因为每次扩容都需要对每个节点进行重 hash，从一个 table 转移到新的 table 中，这个过程会耗费大量的时间和 cpu 资源。
- 插入数据操作锁住的是表头，这是并发效率高于 jdk1.7 的地方；

I、hash 计算的 spread 方法

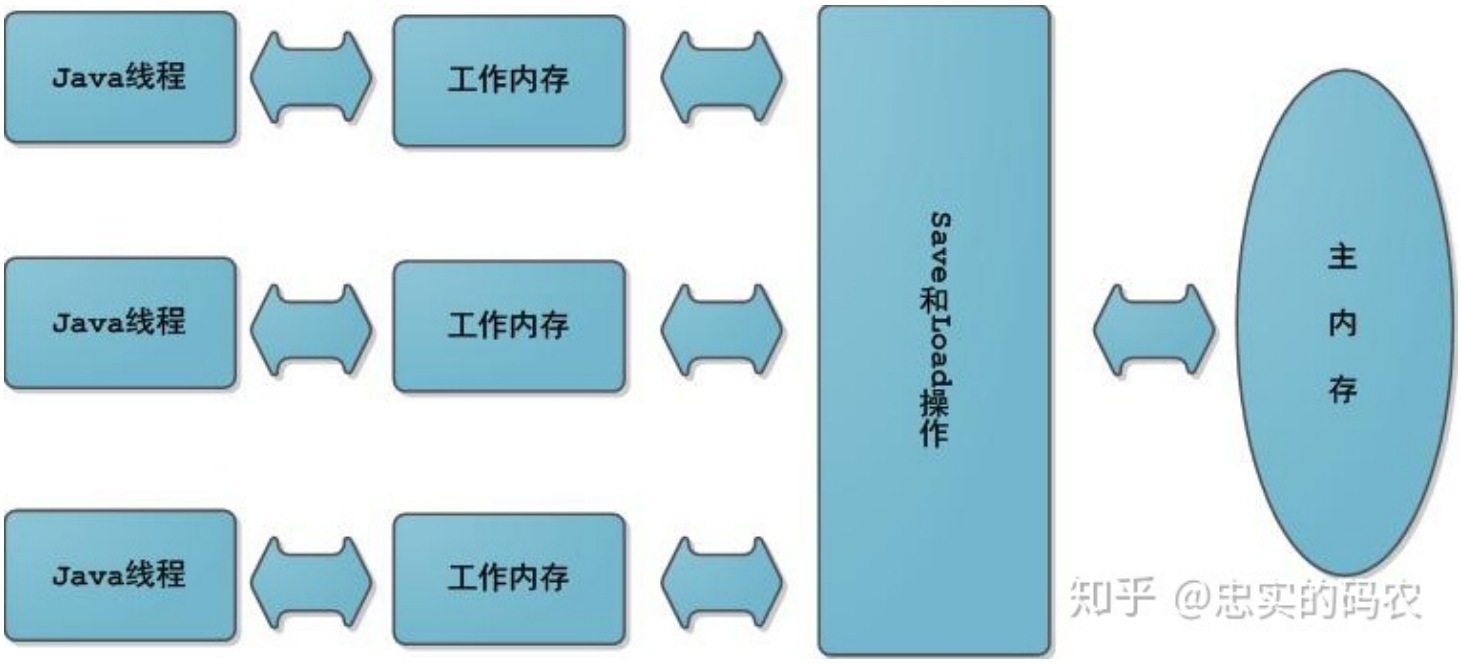
```
/**
     * Spreads (XORs) higher bits of hash to lower and also forces top
     * bit to 0. Because the table uses power-of-two masking, sets of
     * hashes that vary only in bits above the current mask will
     * always collide. (Among known examples are sets of Float keys
     * holding consecutive whole numbers in small tables.)  So we
     * apply a transform that spreads the impact of higher bits
     * downward. There is a tradeoff between speed, utility, and
     * quality of bit-spreading. Because many common sets of hashes
     * are already reasonably distributed (so don't benefit from
     * spreading), and because we use trees to handle large sets of
     * collisions in bins, we just XOR some shifted bits in the
     * cheapest possible way to reduce systematic lossage, as well as
     * to incorporate impact of the highest bits that would otherwise
     * never be used in index calculations because of table bounds.
     */
static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
```

从源码中可以看到，jdk1.8 计算 hash 的方法是先获取到 key 的 hashCode，然后对 hashCode 进行高 16 位和低 16 位异或运算，然后再与 0x7fffffff 进行与运算。高低位异或运算可以保证 haahCode 的每一位都可以参与运算，从而使运算的结果更加均匀的分布在不同的区域，在计算 table 位置时可以减少冲突，提高效率，我们知道 Map 在 put 操作时大部分性能都耗费在解决 hash 冲突上面。得出运算结果后再和 0x7fffffff 与运算，其目的是保证每次运算结果都是一个正数。对于 java 位运算不了解的同学，建议百度自行了解相关内容。

II、java 内存模型和 cas 操作

这里我只是简单的说一下 java 的内存模型和 cas，因为这篇文章的主角的 ConcurrentHashMap。

java 内存模型：在 java 中线程之间的通讯是通过共享内存（即我们在变成时声明的成员变量或叫全局变量）的来实现的。Java 内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存（可以与前面将的处理器的高速缓存类比），线程的工作内存中保存了该线程使用到的变量到主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成，线程、主内存和工作内存的交互关系如下图所示，和上图很类似。



举一个非常简单的例子，就是我们常用的 i++的操作，这个操作看起来只有一行，然而在编译器中这一行代码会被编译成 3 条指令，分别是读取、更新和写入，所以 i++并不是一个原子操作，在多线程环境中是有问题了。其原因在于（我们假设当前 i 的值为 1）当一条线程向主内存中读取数据时，还没来得及把更新后的值刷新到主内存中，另一个线程就已经开始向主内存中读取了数据，而此时内存中的值仍然为 1，两个线程执行+1 操作后得到的结果都为 2，然后将结果刷新到主内存中，整个 i++操作结果，最终得到的结果为 2，但是我们预想的结果应该是 3，这就出现了线程安全的问题了。

cas： cas 的全名称是 Compare And Swap 即比较交换。cas 算法在不需要加锁的情况也可以保证多线程安全。核心思想是： cas 中有三个变量，要更新的变量 V，预期值 E 和新值 N,首先先读取 V 的值，然后进行相关的操作，操作完成后再向主存中读取一次取值为 E，当且仅当 V == E 时才将 N 赋值给 V，否则再走一遍上诉的流程，直至更新成功为止。就拿上面的 i++的操作来做说明，假设当前 i=1，两个线程同时对 i 进行+1 的操作，线程 A 中 V = 1，E = 1，N = 2；线程 B 中 V = 1，E = 1，N = 2；假设线程 A 先执行完整个操作，此时线程 A 发现 V = E = 1，所以线程 A 将 N 的值赋值给 V，那么此时 i 的值就变成了 2； 线程 B 随后也完成了操作，向主存中读取 i 的值，此时 E = 2，V = 1，V！ = E，发现两个并不相等，说明 i 已经被其他线程修改了，因此不执行更新操作，而是从新读取 V 的值 V = 2，执行+1 后 N = 3，完成后再读取主存中 i 的值，因为此时没有其他线程修改 i 的值了，所以 E = 2，V = E = 2，两个值相等，因此执行赋值操作，将 N 的值赋值给 i，最终得到的结果为 3。在整过过程中始终没有使用到锁，却实现的线程的安全性。

从上面的过程知道，cas 会面临着两个问题，一个是当线程一直更新不成功的话，那么这个线程就一直处于死循环中，这样会非常耗费 cpu 的资源；另一种是 ABA 的问题，即对 i =1 进行+1 操作后，再-1，那么此时 i 的值仍为 1，而另外一个线程获取的 E 的值也是 1，认为其他线程没有修改过 i，然后进行的更新操作，事实上已经有其他线程修改过了这个值了，这个就是 A ---> B ---> A 的问题；

CAS 缺点以及解决办法

首先明确 CAS 操作是基于多个 CPU 的情况。在没有线程竞争的情况下使用线程偏向锁。
在轻度到中度(轻量级锁+(自适应)自旋锁)的争用情况下，非阻塞算法的性能会超越阻塞算法，因为 CAS 的多数时间都在第一次尝试时就成功，而发生争用时的开销也不涉及线程挂起和上下文切换，只多了几个循环迭代。
没有争用的 CAS 要比没有争用的锁开销小得多（这句话肯定是真的，因为没有争用的锁涉及 CAS 加上额外的处理），而争用的 CAS 比争用的锁获取涉及更短的延迟。
在高度争用(重量级锁)的情况下（即有多个线程不断争用一个内存位置的时候），基于锁的算法开始提供比非阻塞算法更好的吞吐率，因为当线程阻塞时，它就会停止争用，耐心地等候轮到自己，从而避免了进一步争用。
但是，这么高的争用程度并不常见，因为多数时候，线程会把线程本地的计算与争用共享数据的操作分开，从而给其他线程使用共享数据的机会。

对于 ABA 问题，在深入理解 java 虚拟机一书中，作者的看法。J.U.C 包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference",它可以通过控制变量值的版本来保证 CAS 操作的正确性。
不过这个目前来说比较鸡肋，大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会更加高效。

III、获取 table 对应的索引元素的位置

通过（n-1）& hash 的算法来获得对应的 table 的下标的位置，如果对于这条公式不是很理解的同学可以到： jdk1.8 源码分析-hashMap 博客中了解。

tabAt(Node

casTabAt(Node

IV、扩容

- 如果新增节点之后，所在的链表的元素个数大于等于 8，则会调用 treeifyBin 把链表转换为红黑树。在转换结构时，若 tab 的长度小于 MIN_TREEIFY_CAPACITY，默认值为 64，则会将数组长度扩大到原来的两倍，并触发 transfer，重新调整节点位置。（只有当 tab.length >= 64, ConcurrentHashMap 才会使用红黑树。）
- 新增节点后，addCount 统计 tab 中的节点个数大于阈值（sizeCtl），会触发 transfer，重新调整节点位置。

```
/**
     * Adds to count, and if table is too small and not already
     * resizing, initiates transfer. If already resizing, helps
     * perform transfer if work is available.  Rechecks occupancy
     * after a transfer to see if another resize is already needed
     * because resizings are lagging additions.
     *
     * @param x the count to add
     * @param check if <0, don't check resize, if <= 1 only check if uncontended
     */
private final void addCount(long x, int check) {
    CounterCell[] as;
    long b, s;
    if ((as = counterCells) != null ||
                !U.compareAndSwaplong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a;
        long v;
        int m;
        Boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
                        (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
                        !(uncontended =
                          U.compareAndSwaplong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
                        return;
        s = sumCount();
    }
    if (check >= 0) {
        Node<K,V>[] tab, nt;
        int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
                           (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                                        sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                                        transferIndex <= 0)
                                        break;
                if (U.compareAndSwapint(this, SIZECTL, sc, sc + 1))
                                        transfer(tab, nt);
            } else if (U.compareAndSwapint(this, SIZECTL, sc,
                                                                (rs << RESIZE_STAMP_SHIFT) + 2))
                                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

五、get 操作

get 操作中没有使用到同步的操作，所以相对来说比较简单一点。通过 key 的 hashCode 计算获得相应的位置，然后在遍历该位置上的元素，找到需要的元素，然后返回，如果没有则返回 null：

```
/**
     * Returns the value to which the specified key is mapped,
     * or {@code null} if this map contains no mapping for the key.
     *
     * <p>More formally, if this map contains a mapping from a key
     * {@code k} to a value {@code v} such that {@code key.equals(k)},
     * then this method returns {@code v}; otherwise it returns
     * {@code null}.  (There can be at most one such mapping.)
     *
     * @throws NullPointerException if the specified key is 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());
    if ((tab = table) != null && (n = tab.length) > 0 &&
                (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                                return e.val;
        } else if (eh < 0)
                        return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
            if (e.hash == h &&
                                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                                return e.val;
        }
    }
    return null;
}
```