Phân tích mã nguồn ConcurrentHashMap

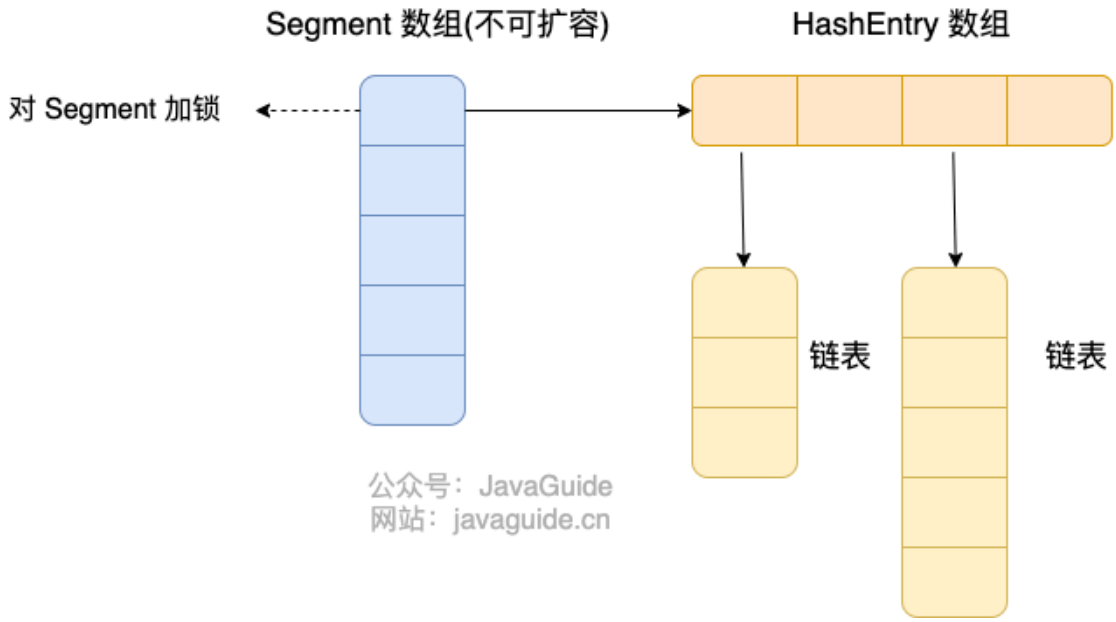
Hướng dẫn Java Bộ sưu tập Java Khoảng 4548 từ Khoảng 15 phút

> Bài viết này được trích từ tài khoản chính thức: The Last Read Code, địa chỉ gốc là: https://mp.weixin.qq.com/s/AHWzboztt53ZfFZmsSnMSw .

Bài viết trước đã giới thiệu mã nguồn HashMap, vốn đã nhận được rất nhiều phản hồi tích cực và nhiều sinh viên đã bày tỏ ý kiến. Lần này, nó trở lại, lần này `ConcurrentHashMap` với HashMap an toàn luồng, cũng được sử dụng thường xuyên. Vậy cấu trúc lưu trữ và nguyên lý triển khai của nó là gì?

1. ConcurrentHashMap 1.7

1. Cấu trúc lưu trữ



`ConcurrentHashMap` Cấu trúc lưu trữ của trong Java 7 được thể hiện trong hình trên. Nó bao gồm `ConcurrnetHashMap` nhiều `Segment` , và mỗi , `Segment` là một `HashMap` cấu trúc tương tự như , nên dung lượng lưu trữ bên trong của mỗi , `HashMap` có thể được mở rộng. Tuy nhiên `Segment` , sau khi **khởi tạo, số lượng , không thể thay đổi** . `Segment` Số lượng , mặc định là 16, có thể được coi là `ConcurrentHashMap` hỗ trợ tối đa 16 luồng đồng thời theo mặc định.

2. Khởi tạo

Khám phá quá trình khởi tạo của thông qua `ConcurrentHashMap` cấu trúc không có đối số của `ConcurrentHashMap` .

```java
    /**
     * Creates a new, empty map with a default initial capacity
(16),
     * load factor (0.75) and concurrencyLevel (16).
     */
    public ConcurrentHashMap() {
        this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR,
DEFAULT_CONCURRENCY_LEVEL);
    }
```

Cấu trúc tham số được gọi trong cấu trúc không tham số và các giá trị mặc định của ba tham số được truyền vào. Giá trị của chúng là.

```java
    /**
     * 默认初始化容量
     */
    static final int DEFAULT_INITIAL_CAPACITY = 16;

    /**
     * 默认负载因子
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /**
     * 默认并发级别
     */
    static final int DEFAULT_CONCURRENCY_LEVEL = 16;
```

Tiếp theo, chúng ta hãy xem xét logic triển khai bên trong của hàm tạo tham số này.

```java
@SuppressWarnings("unchecked")
public ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel) {
    // 参数校验
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    // 校验并发级别大小，大于 1<<16，重置为 65536
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // Find power-of-two sizes best matching arguments
    // 2的多少次方
    int sshift = 0;
    int ssize = 1;
    // 这个循环可以找到 concurrencyLevel 之上最近的 2的次方值
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    // 记录段偏移量
    this.segmentShift = 32 - sshift;
    // 记录段掩码
    this.segmentMask = ssize - 1;
    // 设置容量
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    // c = 容量 / ssize ，默认 16 / 16 = 1，这里是计算每个 Segment 中的类似于 HashMap 的容量
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    //Segment 中的类似于 HashMap 的容量至少是2或者2的倍数
    while (cap < c)
        cap <<= 1;
    // create segments and segments[0]
    // 创建 Segment 数组，设置 segments[0]
    Segment<K,V> s0 = new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                         (HashEntry<K,V>[])new HashEntry[cap]);
```

```
    Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
    UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of
segments[0]
    this.segments = ss;
}
```

Chúng ta hãy tóm tắt lại logic khởi tạo của ConcurrentHashMap trong Java 7.

1. Xác minh tham số bắt buộc.
2. Kiểm tra mức độ đồng thời `concurrencyLevel` . Nếu mức độ đồng thời lớn hơn giá trị tối đa, hãy đặt lại về giá trị tối đa. **Giá trị mặc định của hàm tạo không có đối số là 16.**
3. Tìm giá trị **lũy thừa gần nhất của 2** `concurrencyLevel` trên mức đồng thời làm dung lượng ban đầu. **Mặc định là 16** .
4. Ghi lại `segmentShift` độ lệch. Giá trị này là N trong [sức chứa = 2 mũ N] và sẽ được sử dụng để tính toán vị trí khi gạt. **Giá trị mặc định là 32 - sshift = 28** .
5. Record `segmentMask` , mặc định là ssize - 1 = 16 -1 = 15.
6. **Khởi tạo `segments[0]` , kích thước mặc định là 2 , hệ số tải là 0,75 , ngưỡng mở rộng là 2*0,75=1,5** và việc mở rộng chỉ xảy ra khi giá trị thứ hai được chèn vào.

3. đặt

Sau đó tiếp tục xem mã nguồn phương thức put dựa trên các tham số khởi tạo ở trên.

java

```
/**
 * Maps the specified key to the specified value in this table.
 * Neither the key nor the value can be null.
 *
 * <p> The value can be retrieved by calling the <tt>get</tt>
method
 * with a key that is equal to the original key.
 *
 * @param key key with which the specified value is to be
associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with <tt>key</tt>, or
 *         <tt>null</tt> if there was no mapping for <tt>key</tt>
 * @throws NullPointerException if the specified key or value is
null
 */
public V put(K key, V value) {
```

```
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    // hash 值无符号右移 28位（初始化时获得），然后与 segmentMask=15 做与运
算
    // 其实也就是把高4位与segmentMask（1111）做与运算
    int j = (hash >>> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject          //
nonvolatile; recheck
         (segments, (j << SSHIFT) + SBASE)) == null) //  in
ensureSegment
        // 如果查找到的 Segment 为空，初始化
        s = ensureSegment(j);
    return s.put(key, hash, value, false);
}

/**
 * Returns the segment for the given index, creating it and
 * recording in segment table (via CAS) if not already present.
 *
 * @param k the index
 * @return the segment
 */
@SuppressWarnings("unchecked")
private Segment<K,V> ensureSegment(int k) {
    final Segment<K,V>[] ss = this.segments;
    long u = (k << SSHIFT) + SBASE; // raw offset
    Segment<K,V> seg;
    // 判断 u 位置的 Segment 是否为null
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) ==
null) {
        Segment<K,V> proto = ss[0]; // use segment 0 as prototype
        // 获取0号 segment 里的 HashEntry<K,V> 初始化长度
        int cap = proto.table.length;
        // 获取0号 segment 里的 hash 表里的扩容负载因子，所有的 segment
的 loadFactor 是相同的
        float lf = proto.loadFactor;
        // 计算扩容阀值
        int threshold = (int)(cap * lf);
        // 创建一个 cap 容量的 HashEntry 数组
        HashEntry<K,V>[] tab = (HashEntry<K,V>[])new
HashEntry[cap];
```

```
        if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
== null) { // recheck
            // 再次检查 u 位置的 Segment 是否为null，因为这时可能有其他线
程进行了操作
            Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);
            // 自旋检查 u 位置的 Segment 是否为null
            while ((seg =
(Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
                   == null) {
                // 使用CAS 赋值，只会成功一次
                if (UNSAFE.compareAndSwapObject(ss, u, null, seg =
s))
                    break;
            }
        }
    }
    return seg;
}
```

Mã nguồn trên phân tích `ConcurrentHashMap` luồng xử lý khi nhập dữ liệu. Sau đây là tóm tắt về quy trình cụ thể.

1. Tính toán vị trí của chìa khóa cần đặt và lấy vị trí đã chỉ định `Segment` .
2. Nếu vị trí được chỉ định `Segment` trống, hãy khởi tạo giá trị này `Segment` .

   Khởi tạo quy trình phân đoạn:

   1. `Segment` Kiểm tra xem vị trí được tính toán có phải là giá trị null hay không.
   2. Nếu null, tiếp tục khởi tạo và `Segment[0]` tạo một `HashEntry` mảng có dung lượng và hệ số tải là .
   3. `Segment` Kiểm tra lại xem giá trị tính toán tại vị trí chỉ định có phải là null không.
   4. `HashEntry` Khởi tạo Phân đoạn này bằng cách sử dụng mảng đã tạo .
   5. Spin kiểm tra xem giá trị tính toán ở vị trí chỉ định có phải `Segment` là null hay không và sử dụng CAS để gán giá trị ở vị trí này `Segment` .
3. `Segment.put` Chèn khóa, giá trị.

Phần trên khám phá các thao tác lấy `Segment` và khởi tạo các phân đoạn `Segment` . Phương thức put ở dòng cuối cùng `Segment` vẫn chưa được xem xét, vì vậy hãy tiếp tục phân tích.

```java
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 获取 ReentrantLock 独占锁，获取不到，scanAndLockForPut 获取。
    HashEntry<K,V> node = tryLock() ? null : scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        // 计算要put的数据位置
        int index = (tab.length - 1) & hash;
        // CAS 获取 index 坐标的值
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                // 检查是否 key 已经存在，如果存在，则遍历链表寻找位置，找到后替换 value
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
                        e.value = value;
                        ++modCount;
                    }
                    break;
                }
                e = e.next;
            }
            else {
                // first 有值没说明 index 位置已经有值了，有冲突，链表头插法。
                if (node != null)
                    node.setNext(first);
                else
                    node = new HashEntry<K,V>(hash, key, value, first);
                int c = count + 1;
                // 容量大于扩容阀值，小于最大容量，进行扩容
                if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                    rehash(node);
                else
                    // index 位置赋值 node，node 可能是一个元素，也可能是一个链表的表头
                    setEntryAt(tab, index, node);
```

```
                ++modCount;
                count = c;
                oldValue = null;
                break;
            }
        }
    } finally {
        unlock();
    }
    return oldValue;
}
```

Do tính `Segment` kế thừa `ReentrantLock` , `Segment` việc lấy khóa nội bộ rất thuận tiện và quá trình put sử dụng hàm này.

1. `tryLock()` Lấy khóa. Nếu không lấy được khóa, hãy sử dụng **`scanAndLockForPut`** phương pháp này để tiếp tục lấy khóa.
2. Tính toán vị trí chỉ mục nơi dữ liệu put cần được đặt, sau đó lấy chỉ mục tại vị trí này `HashEntry` .
3. Tại sao chúng ta cần duyệt và thêm phần tử mới? Bởi vì phần tử thu được ở đây `HashEntry` có thể là một phần tử rỗng hoặc có thể đã tồn tại trong danh sách liên kết, nên chúng ta cần xử lý nó theo cách khác.

   Nếu vị trí **`HashEntry` không tồn tại** :

   1. Nếu công suất hiện tại lớn hơn ngưỡng mở rộng nhưng nhỏ hơn công suất tối đa thì **quá trình mở rộng sẽ được thực hiện** .
   2. Cắm trực tiếp.

   `HashEntry` Nếu tồn tại ở vị trí này :

   1. Xác định xem khóa và giá trị băm của phần tử hiện tại trong danh sách liên kết có nhất quán với khóa và giá trị băm cần thêm vào hay không. Nếu chúng nhất quán, hãy thay thế giá trị.
   2. Nếu có bất kỳ sự không nhất quán nào, hãy lấy nút tiếp theo trong danh sách liên kết cho đến khi tìm thấy và thay thế được giá trị giống hệt hoặc không có giá trị giống hệt trong danh sách liên kết.
      1. Nếu công suất hiện tại lớn hơn ngưỡng mở rộng nhưng nhỏ hơn công suất tối đa thì **quá trình mở rộng sẽ được thực hiện** .
      2. Phương pháp chèn đầu liên kết trực tiếp.

4. Nếu vị trí cần chèn đã tồn tại, giá trị cũ sẽ được trả về sau khi thay thế, nếu không thì trả về giá trị null.

Các thao tác trong bước đầu tiên `scanAndLockForPut` không được mô tả ở đây. Phương pháp này chỉ đơn giản là quay `tryLock()` để lấy khóa. Khi số vòng quay vượt quá số lượng quy định, nó sẽ sử dụng `lock()` chặn để lấy khóa. Trong quá trình quay, nó sẽ truy xuất vị trí băm trong bảng `HashEntry` .

```java
private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value)
{
    HashEntry<K,V> first = entryForHash(this, hash);
    HashEntry<K,V> e = first;
    HashEntry<K,V> node = null;
    int retries = -1; // negative while locating node
    // 自旋获取锁
    while (!tryLock()) {
        HashEntry<K,V> f; // to recheck first below
        if (retries < 0) {
            if (e == null) {
                if (node == null) // speculatively create node
                    node = new HashEntry<K,V>(hash, key, value,
null);
                retries = 0;
            }
            else if (key.equals(e.key))
                retries = 0;
            else
                e = e.next;
        }
        else if (++retries > MAX_SCAN_RETRIES) {
            // 自旋达到指定次数后，阻塞等到只到获取到锁
            lock();
            break;
        }
        else if ((retries & 1) == 0 &&
                 (f = entryForHash(this, hash)) != first) {
            e = first = f; // re-traverse if entry changed
            retries = -1;
        }
    }
    return node;
}
```

4. Mở rộng và làm lại

`ConcurrentHashMap` Việc mở rộng sẽ chỉ tăng gấp đôi dung lượng ban đầu. Khi dữ liệu trong mảng cũ được chuyển sang mảng mới, vị trí sẽ không thay đổi hoặc trở thành `index+ oldSize` . Nút trong tham số sẽ được chèn vào vị trí đã chỉ định sau khi mở rộng bằng **phương pháp chèn đầu** danh sách liên kết .

```java
private void rehash(HashEntry<K,V> node) {
    HashEntry<K,V>[] oldTable = table;
    // 老容量
    int oldCapacity = oldTable.length;
    // 新容量，扩大两倍
    int newCapacity = oldCapacity << 1;
    // 新的扩容阀值
    threshold = (int)(newCapacity * loadFactor);
    // 创建新的数组
    HashEntry<K,V>[] newTable = (HashEntry<K,V>[]) new
HashEntry[newCapacity];
    // 新的掩码，默认2扩容后是4，-1是3，二进制就是11。
    int sizeMask = newCapacity - 1;
    for (int i = 0; i < oldCapacity ; i++) {
        // 遍历老数组
        HashEntry<K,V> e = oldTable[i];
        if (e != null) {
            HashEntry<K,V> next = e.next;
            // 计算新的位置，新的位置只可能是不变或者是老的位置+老的容量。
            int idx = e.hash & sizeMask;
            if (next == null)   //  Single node on list
                // 如果当前位置还不是链表，只是一个元素，直接赋值
                newTable[idx] = e;
            else { // Reuse consecutive sequence at same slot
                // 如果是链表了
                HashEntry<K,V> lastRun = e;
                int lastIdx = idx;
                // 新的位置只可能是不变或者是老的位置+老的容量。
                // 遍历结束后，lastRun 后面的元素位置都是相同的
                for (HashEntry<K,V> last = next; last != null;
= last.next) {
                    int k = last.hash & sizeMask;
                    if (k != lastIdx) {
```

```
                        lastIdx = k;
                        lastRun = last;
                    }
                }
                // , lastRun 后面的元素位置都是相同的，直接作为链表赋值到新
位置。
                newTable[lastIdx] = lastRun;
                // Clone remaining nodes
                for (HashEntry<K,V> p = e; p != lastRun; p =
p.next) {
                    // 遍历剩余元素，头插法到指定 k 位置。
                    V v = p.value;
                    int h = p.hash;
                    int k = h & sizeMask;
                    HashEntry<K,V> n = newTable[k];
                    newTable[k] = new HashEntry<K,V>(h, p.key, v,
n);
                }
            }
        }
    }
    // 头插法插入新的节点
    int nodeIndex = node.hash & sizeMask; // add the new node
    node.setNext(newTable[nodeIndex]);
    newTable[nodeIndex] = node;
    table = newTable;
}
```

Một số học viên có thể bị nhầm lẫn bởi hai vòng lặp for cuối cùng. Vòng lặp for đầu tiên tìm một nút có vị trí mới giống nhau cho tất cả các nút tiếp theo. Nút này sau đó được sử dụng như một danh sách liên kết và được gán cho vị trí mới. Vòng lặp for thứ hai chèn các phần tử còn lại vào vị trí đã chỉ định của danh sách liên kết bằng phương pháp chèn đầu. ~~Việc triển khai này có thể dựa trên thống kê xác suất. Học viên có nghiên cứu chuyên sâu có thể chia sẻ ý kiến của mình.~~

`for` Vòng lặp thứ hai bên trong `new HashEntry<K,V>(h, p.key, v, n)` tạo ra một vòng lặp mới `HashEntry` thay vì sử dụng lại vòng lặp trước đó vì nếu vòng lặp trước đó được sử dụng lại, `get` luồng đang duyệt (chẳng hạn như thực thi phương thức) sẽ không thể duyệt tiếp do con trỏ đã bị sửa đổi. Như bình luận đã nói:

> Các nút được thay thế sẽ được thu gom rác khi chúng không còn được tham chiếu bởi bất kỳ luồng đọc nào có thể đang duyệt bảng đồng thời.

> Các nút mà chúng thay thế sẽ có thể được thu gom rác ngay khi chúng không còn được tham chiếu bởi bất kỳ luồng trình đọc nào có thể đang trong quá trình duyệt bảng đồng thời

Tại sao chúng ta cần sử dụng một `for` vòng lặp khác để tìm nó `lastRun` ? Thực tế, nó nhằm mục đích giảm số lượng đối tượng được tạo ra, như đã nêu trong chú thích:

> Theo thống kê, theo ngưỡng mặc định, khi dung lượng bảng tăng gấp đôi, chỉ cần sao chép khoảng một phần sáu số nút.
> Theo thống kê, ở ngưỡng mặc định, chỉ có khoảng một phần sáu trong số chúng cần được sao chép khi một bảng tăng gấp đôi.

5. nhận được

Ở đây rất đơn giản, phương thức get chỉ cần hai bước.

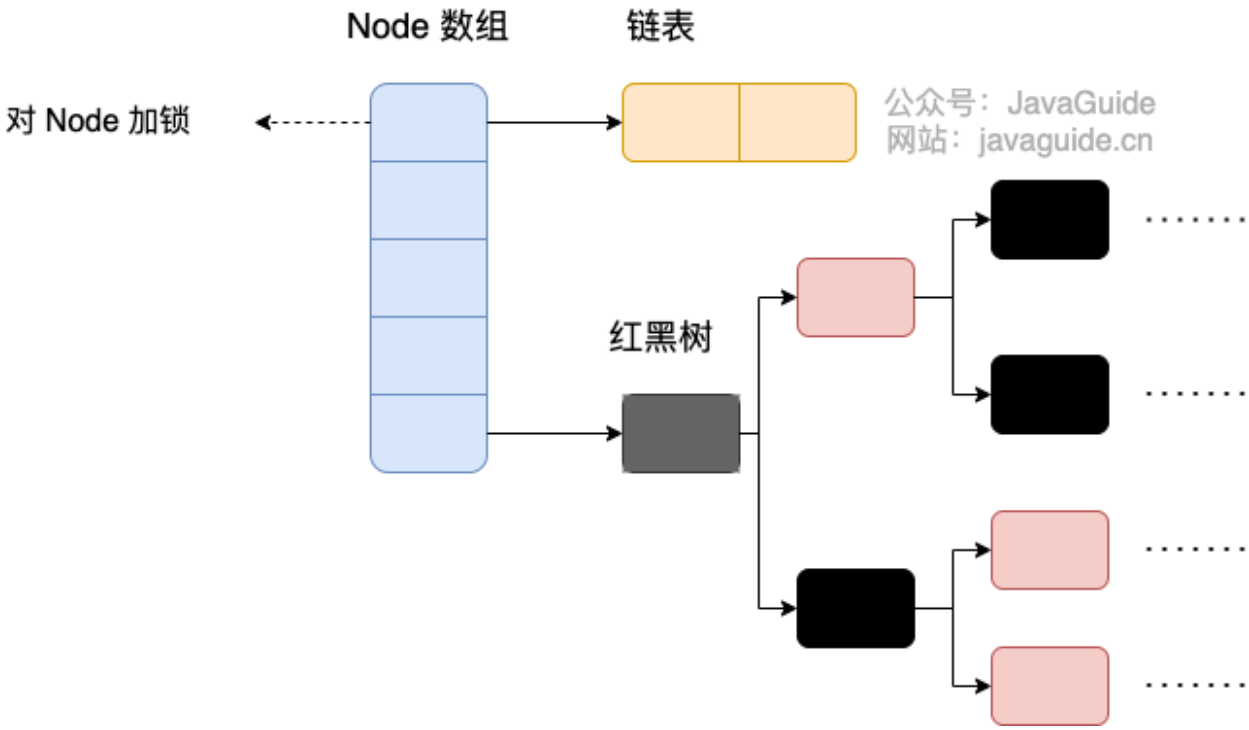
1. Tính toán vị trí lưu trữ của khóa.
2. Duyệt qua vị trí đã chỉ định để tìm giá trị của cùng một khóa.

```java
public V get(Object key) {
    Segment<K,V> s; // manually integrate access methods to reduce overhead
    HashEntry<K,V>[] tab;
    int h = hash(key);
    long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    // 计算得到 key 的存放位置
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
             e != null; e = e.next) {
            // 如果是链表，遍历查找到相同 key 的 value。
            K k;
            if ((k = e.key) == key || (e.hash == h && key.equals(k)))
```

```
                return e.value;
            }
        }
        return null;
    }
```

2. ConcurrentHashMap 1.8

1. Cấu trúc lưu trữ



ConcurrentHashMap của Java 8 đã có những thay đổi đáng kể so với Java 7. Thay vì sử dụng **mảng Segment + mảng HashEntry + danh sách liên kết** trước đây , giờ đây nó sử dụng **mảng Node + danh sách liên kết/cây đỏ-đen** . Khi danh sách liên kết xung đột đạt đến một độ dài nhất định, danh sách liên kết sẽ được chuyển đổi thành cây đỏ-đen.

2. Khởi tạo initTable

```java
/**
 * Initializes table, using the size recorded in sizeCtl.
 */
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        //  如果 sizeCtl < 0 ,说明另外的线程执行CAS 成功，正在进行初始化。
        if ((sc = sizeCtl) < 0)
            // 让出 CPU 使用权
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

Từ mã nguồn, chúng ta có thể thấy quá trình khởi tạo được hoàn tất `ConcurrentHashMap` thông qua các phép toán **spin và CAS** `sizeCtl` . Điều quan trọng cần lưu ý là biến (viết tắt của sizeControl) xác định trạng thái khởi tạo hiện tại.

1. -1 cho biết quá trình khởi tạo đang diễn ra và các luồng khác cần phải quay và chờ.
2. -N biểu thị bảng đang được mở rộng. 16 bit trên biểu thị dấu nhận dạng mở rộng, và 16 bit dưới trừ 1 là số luồng đang được mở rộng.
3. 0 nghĩa là kích thước khởi tạo của bảng. Nếu bảng chưa được khởi tạo
4. >0 biểu thị ngưỡng mở rộng bảng nếu bảng đã được khởi tạo.

3. đặt

Chỉ cần xem qua mã nguồn đã đưa vào.

```java
public V put(K key, V value) {
    return putVal(key, value, false);
}

/** 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();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        // f = 目标位置元素
        Node<K,V> f; int n, i, fh;// fh 后面存放目标位置的元素 hash 值
        if (tab == null || (n = tab.length) == 0)
            // 数组桶为空，初始化数组桶（自旋+CAS）
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 桶内为空，CAS 放入，不加锁，成功了就直接 break 跳出
            if (casTabAt(tab, i, null,new Node<K,V>(hash, key,
value, null)))
                break;  // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            // 使用 synchronized 加锁加入节点
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    // 说明是链表
                    if (fh >= 0) {
                        binCount = 1;
                        // 循环加入新的或者覆盖节点
                        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;
                            }
                        }
                    }
                    else if (f instanceof TreeBin) {
                        // 红黑树
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash,
key,
                                                              value)) !=
null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
```

```
            }
        }
        addCount(1L, binCount);
        return null;
    }
```

1. Tính toán mã băm dựa trên khóa.
2. Xác định xem có cần khởi tạo hay không.
3. Đây là nút được định vị bởi khóa hiện tại. Nếu nút này trống, điều đó có nghĩa là dữ liệu có thể được ghi vào vị trí hiện tại. CAS được sử dụng để thử ghi. Nếu không thành công, hãy xoay để đảm bảo thành công.
4. Nếu vị trí hiện tại là `hashcode == MOVED == -1`, thì cần phải mở rộng.
5. Nếu không đáp ứng được bất kỳ điều kiện nào, dữ liệu sẽ được ghi bằng khóa đồng bộ.
6. Nếu số lớn hơn `TREEIFY_THRESHOLD`, phương thức cây sẽ được thực thi. Trong `treeifyBin`, trước tiên, phương thức này sẽ xác định độ dài mảng hiện tại ≥ 64 trước khi danh sách liên kết được chuyển đổi thành cây đỏ-đen.

4. nhận được

Quá trình lấy dữ liệu khá đơn giản, chỉ cần xem trực tiếp mã nguồn.

```java
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // key 所在的 hash 位置
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 如果指定位置元素存在，头结点hash值相同
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null &&
key.equals(ek)))
                // key hash 值相等，key值相同，直接返回元素 value
                return e.val;
        }
        else if (eh < 0)
            // 头结点hash值小于0，说明正在扩容或者是红黑树，find查找
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
```

```
            // 是链表，遍历查找
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null &&
key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

Tóm tắt quá trình lấy:

1. Tính toán vị trí dựa trên giá trị băm.
2. Tìm vị trí đã chỉ định, nếu nút đầu là nút bạn đang tìm kiếm, hãy trả về giá trị của nút đó trực tiếp.
3. Nếu giá trị băm của nút đầu nhỏ hơn 0, điều đó có nghĩa là nút đó đang được mở rộng hoặc đó là cây đỏ đen. Hãy tìm kiếm nút đó.
4. Nếu đó là danh sách được liên kết, hãy duyệt qua và tìm kiếm trong đó.

Tóm tắt:

Nhìn chung, `ConcurrentHashMap` có khá nhiều thay đổi trong Java 8 so với Java 7.

3. Tóm tắt

Java 7 `ConcurrentHashMap` sử dụng khóa phân đoạn, nghĩa là chỉ một luồng có thể thao tác trên mỗi phân đoạn tại một thời điểm. Mỗi phân đoạn `Segment` là một cấu trúc tương tự `HashMap` như một mảng, có thể được mở rộng và các xung đột được chuyển đổi thành một danh sách liên kết. Tuy nhiên, `Segment` sau khi khởi tạo, số lượng phân đoạn không thể thay đổi.

Java 8 `ConcurrentHashMap` sử dụng `Synchronized` cơ chế khóa và CAS. Cấu trúc này đã phát triển từ **`Segment` mảng + `HashEntry` mảng + danh sách liên kết** của Java 7 thành **mảng nút + danh sách liên kết/cây đỏ-đen** . Một nút tương tự như một HashEntry. Khi số lượng xung đột đạt đến một mức nhất định, nó sẽ chuyển đổi thành cây đỏ-đen. Khi số lượng xung đột giảm xuống dưới một mức nhất định, nó sẽ trở lại thành danh sách liên kết.

Một số sinh viên có thể `Synchronized` nghi ngờ về hiệu suất của . Trên thực tế, `Synchronized` kể từ khi chiến lược nâng cấp khóa được giới thiệu, hiệu suất không còn là vấn đề nữa. Những sinh viên quan tâm có thể tự tìm hiểu về `Synchronized` việc **nâng**

cấp khóa .

Cập nhật gần đây2024/11/9 17:11

Người đóng góp: shuang.kou , hướng dẫn ,辉鸭蛋, tangj1992 , nicollchen , Itswag , Rentianle2022 , ReedZheng , Guide , Erzbir , Mr.Hope , Zzr-rr , HoeYeungHo