二

29 HashMap 为什么是线程不安全的?

本课时我们主要讲解为什么 HashMap 是线程不安全的？而对于 HashMap，相信你一定并不陌生，HashMap 是我们平时工作和学习中用得非常非常多的一个容器，也是 Map 最主要的实现类之一，但是它自身并不具备线程安全的特点，可以从多种情况中体现出来，下面我们就对此进行具体的分析。

源码分析

第一步，我们来看一下 HashMap 中 put 方法的源码：

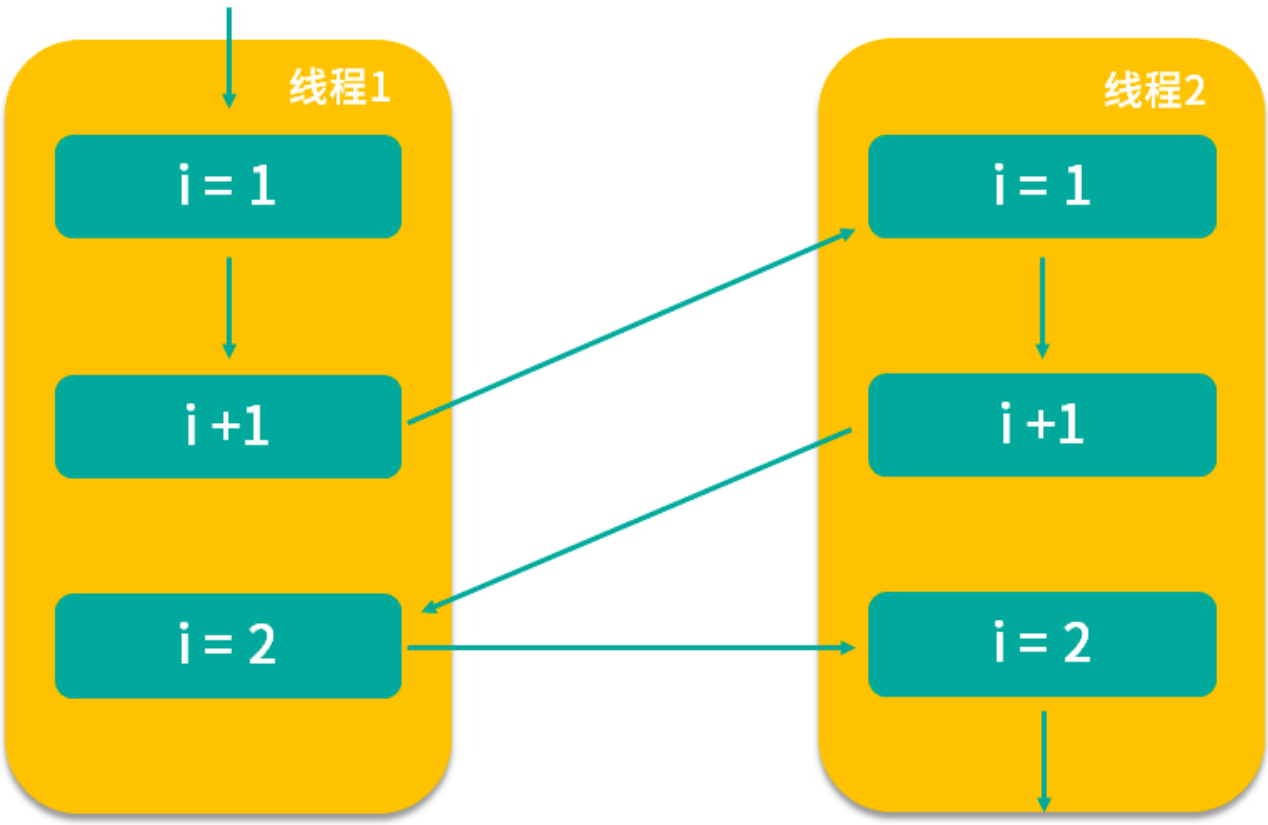
```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    //modCount++ 是一个复合操作
    modCount++;
```

```
        addEntry(hash, key, value, i);

        return null;

    }
```

在 HashMap 的 put() 方法中，可以看出里面进行了很多操作，那么在这里，我们把目光聚焦到标记出来的 modCount++ 这一行代码中，相信有经验的小伙伴一定发现了，这相当于是典型的“i++”操作，正是我们在 06 课时讲过的线程不安全的“运行结果错误”的情况。从表面上看 i++ 只是一行代码，但实际上它并不是一个原子操作，它的执行步骤主要分为三步，而且在每步操作之间都有可能被打断。

- 第一个步骤是读取；
- 第二个步骤是增加；
- 第三个步骤是保存。

那么我们接下来具体看一下如何发生的线程不安全问题。



我们根据箭头指向依次看，假设线程 1 首先拿到 i=1 的结果，然后进行 i+1 操作，但此时 i+1 的结果并没有保存下来，线程 1 就被切换走了，于是 CPU 开始执行线程 2，它所做的事情和线程 1 是一样的 i++ 操作，但此时我们想一下，它拿到的 i 是多少？实际上和线程 1 拿到的 i 的结果一样都是 1，为什么呢？因为线程 1 虽然对 i 进行了 +1 操作，但结果没有

保存，所以线程 2 看不到修改后的结果。

然后假设等线程 2 对 i 进行 +1 操作后，又切换到线程 1，让线程 1 完成未完成的操作，即将 i + 1 的结果 2 保存下来，然后又切换到线程 2 完成 i = 2 的保存操作，虽然两个线程都执行了对 i 进行 +1 的操作，但结果却最终保存了 i = 2 的结果，而不是我们期望的 i = 3，这样就发生了线程安全问题，导致了数据结果错误，这也是最典型的线程安全问题。

所以，从源码的角度，或者说从理论上来讲，这完全足以证明 HashMap 是线程非安全的了。因为如果有多个线程同时调用 put() 方法的话，它很有可能会把 modCount 的值计算错（上述的源码分析针对的是 Java 7 版本的源码，而在 Java 8 版本的 HashMap 的 put 方法中会调用 putVal 方法，里面同样有 ++modCount 语句，所以原理是一样的）。

实验：扩容期间取出的值不准确

刚才我们分析了源码，你可能觉得不过瘾，下面我们就打开代码编辑器，用一个实验来证明 HashMap 是线程不安全的。

为什么说 HashMap 不是线程安全的呢？我们先来讲解下原理。HashMap 本身默认的容量不是很大，如果不停地往 map 中添加新的数据，它便会在合适的时机进行扩容。而在扩容期间，它会新建一个新的空数组，并且用旧的项填充到这个新的数组中去。那么，在这个填充的过程中，如果有线程获取值，很可能会取到 null 值，而不是我们所希望的、原来添加的值。所以我们程序就想演示这种情景，我们来看一下这段代码：

```
public class HashMapNotSafe {

    public static void main(String[] args) {

        final Map<Integer, String> map = new HashMap<>();

        final Integer targetKey = 0b1111_1111_1111_1111; // 65 535

        final String targetValue = "v";

        map.put(targetKey, targetValue);

        new Thread(() -> {

            IntStream.range(0, targetKey).forEach(key -> map.put(key, "someValue"))

        }).start();

        while (true) {

            if (null == map.get(targetKey)) {

                throw new RuntimeException("HashMap is not thread safe.");
```

```
            }
        }
    }
}
```

代码中首先建立了一个 HashMap，并且定义了 key 和 value， key 的值是一个二进制的 1111_1111_1111_1111，对应的十进制是 65535。之所以选取这样的值，就是为了让它在扩容往回填充数据的时候，尽量不要填充得太快，比便于我们能捕捉到错误的发生。而对应的 value 是无所谓的，我们随意选取了一个非 null 的 "v" 来表示它，并且把这个值放到了 map 中。

接下来，我们就用一个新的线程不停地往我们的 map 中去填入新的数据，我们先来看是怎么填入的。首先它用了一个 IntStream，这个 range 是从 0 到之前所讲过的 65535，这个 range 是一个左闭右开的区间，所以会从 0、1、2、3……一直往上加，并且每一次加的时候，这个 0、1、2、3、4 都会作为 key 被放到 map 中去。而它的 value 是统一的，都是 "someValue"，因为 value 不是我们所关心的。

然后，我们就会把这个线程启动起来，随后就进入一个 while 循环，这个 while 循环是关键，在 while 循环中我们会不停地检测之前放入的 key 所对应的 value 还是不是我们所期望的字符串 "v"。我们在 while 循环中会不停地从 map 中取 key 对应的值。如果 HashMap 是线程安全的，那么无论怎样它所取到的值都应该是我们最开始放入的字符串 "v"，可是如果取出来是一个 null，就会满足这个 if 条件并且随即抛出一个异常，因为如果取出 null 就证明它所取出来的值和我们一开始放入的值是不一致的，也就证明了它是线程不安全的，所以在此我们要抛出一个 RuntimeException 提示我们。

下面就让我们运行这个程序来看一看是否会抛出这个异常。一旦抛出就代表它是线程不安全的，这段代码的运行结果：

```
Exception in thread "main" java.lang.RuntimeException: HashMap is not thread safe.
at lesson29.HashMapNotSafe.main(HashMapNotSafe.java:25)
```

很明显，很快这个程序就抛出了我们所希望看到的 RuntimeException，并且我们把它描述为：HashMap is not thread safe，一旦它能进入到这个 if 语句，就已经证明它所取出来的值是 null，而不是我们期望的字符串 "v"。

通过以上这个例子，我们也证明了HashMap 是线程非安全的。

除了刚才的例子之外，还有很多种线程不安全的情况，例如：

同时 put 碰撞导致数据丢失

比如，有多个线程同时使用 put 来添加元素，而且恰好两个 put 的 key 是一样的，它们发生了碰撞，也就是根据 hash 值计算出来的 bucket 位置一样，并且两个线程又同时判断该位置是空的，可以写入，所以这两个线程的两个不同的 value 便会添加到数组的同一个位置，这样最终就只会保留一个数据，丢失一个数据。

可见性问题无法保证

我们再从可见性的角度去考虑一下。可见性也是线程安全的一部分，如果某一个数据结构声称自己是线程安全的，那么它同样需要保证可见性，也就是说，当一个线程操作这个容器的时候，该操作需要对另外的线程都可见，也就是其他线程都能感知到本次操作。可是 HashMap 对此是做不到的，如果线程 1 给某个 key 放入了一个新值，那么线程 2 在获取对应的 key 的值的时候，它的可见性是无法保证的，也就是说线程 2 可能可以看到这一次的更改，但也有可能看不到。所以从可见性的角度出发，HashMap 同样是线程非安全的。

死循环造成 CPU 100%

下面我们再举一个死循环造成 CPU 100% 的例子。HashMap 有可能会发生死循环并且造成 CPU 100% ，这种情况发生最主要的原因就是在扩容的时候，也就是内部新建新的 HashMap 的时候，扩容的逻辑会反转散列桶中的节点顺序，当有多个线程同时进行扩容的时候，由于 HashMap 并非线程安全的，所以如果两个线程同时反转的话，便可能形成一个循环，并且这种循环是链表的循环，相当于 A 节点指向 B 节点，B 节点又指回到 A 节点，这样一来，在下一次想要获取该 key 所对应的 value 的时候，便会在遍历链表的时候发生永远无法遍历结束的情况，也就发生 CPU 100% 的情况。

所以综上所述，HashMap 是线程不安全的，在多线程使用场景中如果需要使用 Map，应该尽量避免使用线程不安全的 HashMap。同时，虽然 Collections.synchronizedMap(new HashMap()) 是线程安全的，但是效率低下，因为内部用了很多的 synchronized，多个线程不能同时操作。推荐使用线程安全同时性能比较好的 ConcurrentHashMap。关于 ConcurrentHashMap 我们会在下一个课时中介绍。

上一页 下一页