【线上问题备忘2020-11-12】2020-11-12前css2c站点cpu持续占用不释放问题备忘

由 韩那松创建, 最后修改于十一月 23, 2020

【问题现象】

2020-11-06 ： 活动和网关反应线上查询昵称接口存在持续超时的情况。

经过检查ump和ulp日志，发现css2c其中一台机器105.95 cpu比较高，经过重启后业务恢复。

根据ump分析，从11-03开始，每天凌晨3点有一台机器cpu升高，11-6日3点有两台机器升高，其中105.95第二次选中，故cpu占用更为明显，也开始出现接口超时现象。

【问题影响】

每天凌晨3点会使其中一台机器cpu上升近20%，cpu打满会导致此台机器对外的响应变慢，表现在对外的接口出现请求超时的情况，需要重启才能释放。

【问题排查】

1、检查css2c站点三点时的接口调用情况，对比此时间段接口调用量上升的情况。

排查发现getBatchFriendNickName接口仅在3点时段发生4分钟左右的调用

2、检查11-03 css2c站点上线的功能，确认getBatchFriendNickName接口在本次上线项目的修改范围中，检查代码后发现，存在多线程操作HashMap的代码。

```
1873        //用户->朋友备注  单uid
1874        Map<Long, String> singleUidfriendRemarkMap = new HashMap<>();
1875        //用户->好友  单uid
1876        Map<Long, Long> userToSingleUidFriendMap = new HashMap<>();
1877
1878        //用户->好友  双uid
1879        Map<Long, Long> userToMsnyUidFriendMap = new HashMap<>();
1880
1881        Set<Long> finalManyUidSet = manyUidSet;
1882        friendList.parallelStream().forEach(f->{
1883            if (finalManyUidSet.contains(f.getFriendId())) {
1884                userToMsnyUidFriendMap.put(f.getCustomerId(), f.getFriendId());
1885            } else {
1886                userToSingleUidFriendMap.put(f.getCustomerId(), f.getFriendId());
1887                if (StringUtils.isNotBlank(f.getRemark())) {
1888                    singleUidfriendRemarkMap.put(f.getCustomerId(), f.getRemark());
1889                }
1890            }
1891        });
```

3、11-09 10点，在105.66重启前，使用jstack打印线程日志，用以检查线程情况，确认存在getBatchFriendNickName的线程正在执行。

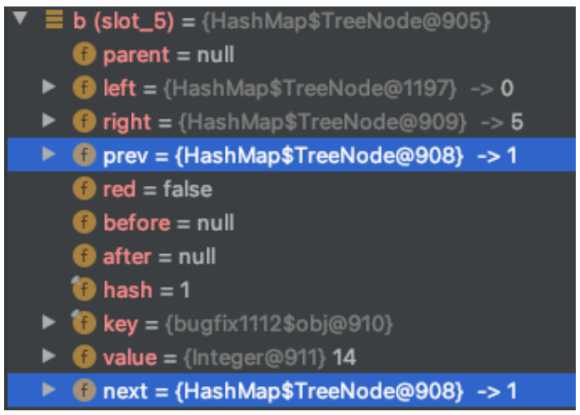
```
"ForkJoinPool.commonPool-worker-3" #3345432 daemon prio=5 os_prio=0 tid=0x00007fa384105000 nid=0xd4f runnable [0x00007fa2dae91000]
   java.lang.Thread.State: RUNNABLE
    at java.util.HashMap$TreeNode.balanceInsertion(HashMap.java:2221)
    at java.util.HashMap$TreeNode.treeify(HashMap.java:1930)
    at java.util.HashMap$TreeNode.split(HashMap.java:2162)
    at java.util.HashMap.resize(HashMap.java:713)
    at java.util.HashMap.putVal(HashMap.java:662)
    at java.util.HashMap.put(HashMap.java:611)
    at com.biyao.css2c.soa.service.role.impl.RoleToCInnerServiceImpl.lambda$getBatchFriendNickname$303(RoleToCInnerServiceImpl.java:1886)
    at com.biyao.css2c.soa.service.role.impl.RoleToCInnerServiceImpl$$Lambda$882/1532352791.accept(Unknown Source)
    at java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:183)
    at java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1359)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:512)
    at java.util.stream.ForEachOps$ForEachTask.compute(ForEachOps.java:290)
    at java.util.concurrent.CountedCompleter.exec(CountedCompleter.java:731)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:902)
    at java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1689)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1644)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)

   Locked ownable synchronizers:
    - None
```

问题点：

1、存在使用parallelStream多线程操作HashMap的情况。HashMap是线程不安全的集合，并行操作时，会出现死循环、抛出异常、数据丢失、栈溢出等情况。

主要原因是多线程下操作同一对象时，对象内部属性的不一致性导致的，本地测试复现形成cpu不释放问题场景如下：

红黑树TreeNode成环，在split和find时会形成死循环



split方法是在HashMap扩容时调用，扩容时会对红黑树节点进行拆分，拆分后的两个红黑树会分布在"原索引位置"和"原索引位置+oldCap"两处。当TreeNode节点成环后，2126行进入死循环

```
2120        final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit) {
2121            TreeNode<K,V> b = this;
2122            // Relink into lo and hi lists, preserving order
2123            TreeNode<K,V> loHead = null, loTail = null;
2124            TreeNode<K,V> hiHead = null, hiTail = null;
2125            int lc = 0, hc = 0;
2126            for (TreeNode<K,V> e = b, next; e != null; e = next) {
2127                next = (TreeNode<K,V>)e.next;
2128                e.next = null;
2129                if ((e.hash & bit) == 0) {
2130                    if ((e.prev = loTail) == null)
2131                        loHead = e;
2132                    else
2133                        loTail.next = e;
2134                    loTail = e;
2135                    ++lc;
2136                }
2137                else {
2138                    if ((e.prev = hiTail) == null)
2139                        hiHead = e;
2140                    else
2141                        hiTail.next = e;
2142                    hiTail = e;
2143                    ++hc;
2144                }
2145            }
2146
2147            if (loHead != null) {
2148                if (lc <= UNTREEIFY_THRESHOLD)
2149                    tab[index] = loHead.untreeify(map);
2150                else {
2151                    tab[index] = loHead;
2152                    if (hiHead != null) // (else is already treeified)
2153                        loHead.treeify(tab);
2154                }
2155            }
2156            if (hiHead != null) {
2157                if (hc <= UNTREEIFY_THRESHOLD)
2158                    tab[index + bit] = hiHead.untreeify(map);
2159                else {
2160                    tab[index + bit] = hiHead;
2161                    if (loHead != null)
2162                        hiHead.treeify(tab);
2163                }
2164            }
2165        }
```

find方法是红黑树节点的查找，当TreeNode节点成环后，会进入死循环，异常在1861行。

```
1837        /**
1838         * Finds the node starting at root p with the given hash and key.
1839         * The kc argument caches comparableClassFor(key) upon first use
1840         * comparing keys.
1841         */
1842    @   final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
1843            TreeNode<K,V> p = this;
1844            do {
1845                int ph, dir; K pk;
1846                TreeNode<K,V> pl = p.left, pr = p.right, q;
1847                if ((ph = p.hash) > h)
1848                    p = pl;
1849                else if (ph < h)
1850                    p = pr;
1851                else if ((pk = p.key) == k || (k != null && k.equals(pk)))
1852                    return p;
1853                else if (pl == null)
1854                    p = pr;
1855                else if (pr == null)
1856                    p = pl;
1857                else if ((kc != null ||
1858                          (kc = comparableClassFor(k)) != null) &&
1859                         (dir = compareComparables(kc, k, pk)) != 0)
1860                    p = (dir < 0) ? pl : pr;
1861                else if ((q = pr.find(h, k, kc)) != null)
1862                    return q;
1863                else
1864                    p = pl;
1865            } while (p != null);
1866            return null;
1867        }
```

【问题修复】

调整此段代码逻辑，不允许使用并发操作HashMap，需要并发时，请选择ConcurrentHashMap：

1、代码去掉parallelStream，此处代码为并发执行map put操作，耗时不高，不需要并行处理。

2020-11-12：修复代码上线，跟踪一周cpu正常。

【后续改进】

1、parallelStream适合没有线程安全问题、较单纯的数据处理任务，使用前要特别注意。在后续CR过程中要关注此类并发的问题。

2、不允许使用并发操作HashMap，需要并发时，请选择ConcurrentHashMap

3、死循环表现在cpu占用高，但不会打印错误日志，定位此种问题需要检查下游站点的接口超时情况，同时需要jstack日志验证。