

17 | 别以为“自动挡”就不可能出现OOM

2020-04-18 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽

时长 17:52 大小 16.38M



你好，我是朱晔。今天，我要和你分享的主题是，别以为“自动挡”就不可能出现 OOM。

这里的“自动挡”，是我对 Java 自动垃圾收集器的戏称。的确，经过这么多年的发展，Java 的垃圾收集器已经非常成熟了。有了自动垃圾收集器，绝大多数情况下我们写程序时可以专注于业务逻辑，无需过多考虑对象的分配和释放，一般也不会出现 OOM。

但，内存空间始终是有限的，Java 的几大内存区域始终都有 OOM 的可能。相应地，Java 程序的常见 OOM 类型，可以分为堆内存的 OOM、栈 OOM、元空间 OOM、直接内存 OOM 等。几乎每一种 OOM 都可以使用几行代码模拟，市面上也有很多资料在堆、元空间、直接内存中分配超大对象或是无限分配对象，尝试创建无限个线程或是进行方法无限递归调用来模拟。

但值得注意的是，我们的业务代码并不会这么干。所以今天，我会从内存分配意识的角度通过一些案例，展示业务代码中可能导致 OOM 的一些坑。这些坑，或是因为我们意识不到对象的分配，或是因为不合理的资源使用，或是没有控制缓存的数据量等。

在 [第 3 讲](#) 介绍线程时，我们已经看到了两种 OOM 的情况，一是因为使用无界队列导致的堆 OOM，二是因为使用没有最大线程数量限制的线程池导致无限创建线程的 OOM。接下来，我们再一起看看，在写业务代码的过程中，还有哪些意识上的疏忽可能会导致 OOM。

太多份相同的对象导致 OOM

我要分享的第一个案例是这样的。有一个项目在内存中缓存了全量用户数据，在搜索用户时可以直接从缓存中返回用户信息。现在为了改善用户体验，需要实现输入部分用户名自动在下拉框提示补全用户名的功能（也就是所谓的自动完成功能）。

在 [第 10 讲](#) 介绍集合时，我提到对于这种快速检索的需求，最好使用 Map 来实现，会比直接从 List 搜索快得多。

为实现这个功能，我们需要一个 HashMap 来存放这些用户数据，Key 是用户姓名索引，Value 是索引下对应的用户列表。举一个例子，如果有两个用户 aa 和 ab，那么 Key 就有三个，分别是 a、aa 和 ab。用户输入字母 a 时，就能从 Value 这个 List 中拿到所有字母 a 开头的用户，即 aa 和 ab。

在代码中，在数据库存入 1 万个测试用户，用户名由 a~j 这 6 个字母随机构成，然后把每一个用户名的前 1 个字母、前 2 个字母以此类推直到完整用户名作为 Key 存入缓存中，缓存的 Value 是一个 UserDTO 的 List，存放的是所有相同的用户名索引，以及对应的用户信息：

 复制代码


```
1 //自动完成的索引，Key是用户输入的部分用户名，Value是对应的用户数据
2 private ConcurrentHashMap<String, List<UserDTO>> autoCompleteIndex = new Concu
3
4 @Autowired
5 private UserRepository userRepository;
6
7 @PostConstruct
8 public void wrong() {
9     //先保存10000个用户名随机的用户到数据库中
```

```

10     userRepository.saveAll(LongStream.rangeClosed(1, 10000).mapToObj(i -> new I
11
12     //从数据库加载所有用户
13     userRepository.findAll().forEach(userEntity -> {
14         int len = userEntity.getName().length();
15         //对于每一个用户，对其用户名的前N位进行索引，N可能是1~6六种长度类型
16         for (int i = 0; i < len; i++) {
17             String key = userEntity.getName().substring(0, i + 1);
18             autoCompleteIndex.computeIfAbsent(key, s -> new ArrayList<>())
19                 .add(new UserDTO(userEntity.getName()));
20         }
21     });
22     log.info("autoCompleteIndex size:{} count:{}", autoCompleteIndex.size(),
23             autoCompleteIndex.entrySet().stream().map(item -> item.getValue()).
24 }

```

对于每一个用户对象 UserDTO，除了有用户名，我们还加入了 10K 左右的数据模拟其用户信息：


 复制代码

```

1  @Data
2  public class UserDTO {
3      private String name;
4      @EqualsAndHashCode.Exclude
5      private String payload;
6
7      public UserDTO(String name) {
8          this.name = name;
9          this.payload = IntStream.rangeClosed(1, 10_000)
10              .mapToObj(__ -> "a")
11              .collect(Collectors.joining(""));
12      }
13  }

```

运行程序后，日志输出如下：

 复制代码

```

1  [11:11:22.982] [main] [INFO ] [.t.c.o.d.UsernameAutoCompleteService:37 ] - au

```

可以看到，一共有 26838 个索引（也就是所有用户名的 1 位、2 位一直到 6 位有 26838 个组合），HashMap 的 Value，也就是 List 一共有 1 万个用户 * 6 = 6 万个 UserDTO 对

象。

使用内存分析工具 MAT 打开堆 dump 发现，6 万个 UserDTO 占用了约 1.2GB 的内存：

i Overview Histogram Retained by 'selection of 'UserDTO" X			
Class Name	Objects ^	Shallow Heap	
<Regex>	<Numeric>	<Numeric>	
org.geekbang.time.commonmistakes.oom.demo4.UserDTO	60,000	1,440,000	
char[]	60,062	1,200,961,984	
java.lang.String	60,062	1,441,488	
Σ Total: 3 entries	180,124	1,203,843,472	

看到这里发现，**虽然真正的用户只有 1 万个，但因为使用部分用户名作为索引的 Key，导致缓存的 Key 有 26838 个，缓存的用户信息多达 6 万个。**如果我们的用户名不是 6 位而是 10 位、20 位，那么缓存的用户信息可能就是 10 万、20 万个，必然会产生堆 OOM。

尝试调大用户名的最大长度，重启程序可以看到类似如下的错误：

复制代码

```
1 [17:30:29.858] [main] [ERROR] [ringframework.boot.SpringApplication:826] - Ap
2 org.springframework.beans.factory.BeanCreationException: Error creating bean w
```

我们可能会想当然地认为，数据库中有 1 万个用户，内存中也应该只有 1 万个 UserDTO 对象，但实现的时候每次都会 new 出来 UserDTO 加入缓存，当然在内存中都是新对象。在实际的项目中，用户信息的缓存可能是随着用户输入增量缓存的，而不是像这个案例一样在程序初始化的时候全量缓存，所以问题暴露得不会这么早。

知道原因后，解决起来就比较简单了。把所有 UserDTO 先加入 HashSet 中，因为 UserDTO 以 name 来标识唯一性，所以重复用户名会被过滤掉，最终加入 HashSet 的 UserDTO 就不足 1 万个。

有了 HashSet 来缓存所有可能的 UserDTO 信息，我们再构建自动完成索引 autoCompleteIndex 这个 HashMap 时，就可以直接从 HashSet 获取所有用户信息来构建了。这样一来，同一个用户名前缀的不同组合（比如用户名为 abc 的用户，a、ab 和 abc 三个 Key）关联到 UserDTO 是同一份：

```
1 @PostConstruct
2 public void right() {
3     ...
4
5     HashSet<UserDTO> cache = userRepository.findAll().stream()
6         .map(item -> new UserDTO(item.getName()))
7         .collect(Collectors.toCollection(HashSet::new));
8
9
10    cache.stream().forEach(userDTO -> {
11        int len = userDTO.getName().length();
12        for (int i = 0; i < len; i++) {
13            String key = userDTO.getName().substring(0, i + 1);
14            autoCompleteIndex.computeIfAbsent(key, s -> new ArrayList<>())
15                .add(userDTO);
16        }
17    });
18    ...
19 }
```

再次分析堆内存，可以看到 UserDTO 只有 9945 份，总共占用的内存不到 200M。这才是我们真正想要的结果。

i Overview	Histogram	Retained by 'selection of 'UserDTO'		
Class Name		Objects ^	Shallow Heap	
<Regex>		<Numeric>	<Numeric>	
char[]		9,945	199,059,120	
java.lang.String		9,945	238,680	
org.geekbang.time.commonmistakes.oom.demo4.UserDTO		9,945	238,680	
Σ Total: 3 entries		29,835	199,536,480	

修复后的程序，不仅相同的 UserDTO 只有一份，总副本数变为了原来的六分之一；而且因为 HashSet 的去重特性，双重节约了内存。

值得注意的是，我们虽然清楚数据总量，但却忽略了每一份数据在内存中可能有多份。我之前还遇到一个案例，一个后台程序需要从数据库加载大量信息用于数据导出，这些数据在数据库中占用 100M 内存，但是 1GB 的 JVM 堆却无法完成导出操作。

我来和你分析下原因吧。100M 的数据加载到程序内存中，变为 Java 的数据结构就已经占用了 200M 堆内存；这些数据经过 JDBC、MyBatis 等框架其实是加载了 2 份，然后领域

模型、DTO 再进行转换可能又加载了 2 次；最终，占用的内存达到了 $200M \times 4 = 800M$ 。

所以，在进行容量评估时，我们不能认为一份数据在程序内存中也是一份。

使用 WeakHashMap 不等于不会 OOM

对于上一节实现快速检索的案例，为了防止缓存中堆积大量数据导致 OOM，一些同学可能会想到使用 WeakHashMap 作为缓存容器。

WeakHashMap 的特点是 Key 在哈希表内部是弱引用的，当没有强引用指向这个 Key 之后，Entry 会被 GC，即使我们无限往 WeakHashMap 加入数据，只要 Key 不再使用，就不会 OOM。

说到了强引用和弱引用，我先和你回顾下 Java 中引用类型和垃圾回收的关系：

垃圾回收器不会回收有强引用的对象；

在内存充足时，垃圾回收器不会回收具有软引用的对象；

垃圾回收器只要扫描到了具有弱引用的对象就会回收，WeakHashMap 就是利用了这个特点。

不过，我要和你分享的第二个案例，恰巧就是不久前我遇到的一个使用 WeakHashMap 却最终 OOM 的案例。我们暂且不论使用 WeakHashMap 作为缓存是否合适，先分析一下这个 OOM 问题。

声明一个 Key 是 User 类型、Value 是 UserProfile 类型的 WeakHashMap，作为用户数据缓存，往其中添加 200 万个 Entry，然后使用 ScheduledThreadPoolExecutor 发起一个定时任务，每隔 1 秒输出缓存中的 Entry 个数：

 复制代码

```
1 private Map<User, UserProfile> cache = new WeakHashMap<>();
2
3 @GetMapping("wrong")
4 public void wrong() {
5     String userName = "zhuye";
6     //间隔1秒定时输出缓存中的条目数
7     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(
8         () -> log.info("cache size:{}", cache.size()), 1, 1, TimeUnit.SECONDS);
9 }
```

```

9      LongStream.rangeClosed(1, 2000000).forEach(i -> {
10          User user = new User(userName + i);
11          cache.put(user, new UserProfile(user, "location" + i));
12      });
13 }

```

执行程序后日志如下：

[复制代码](#)

```

1 [10:30:28.509] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
2 [10:30:29.507] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
3 [10:30:30.509] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController

```

可以看到，输出的 cache size 始终是 200 万，即使我们通过 jvisualvm 进行手动 GC 还是这样。这就说明，这些 Entry 无法通过 GC 回收。如果你把 200 万改为 1000 万，就可以在日志中看到如下的 OOM 错误：

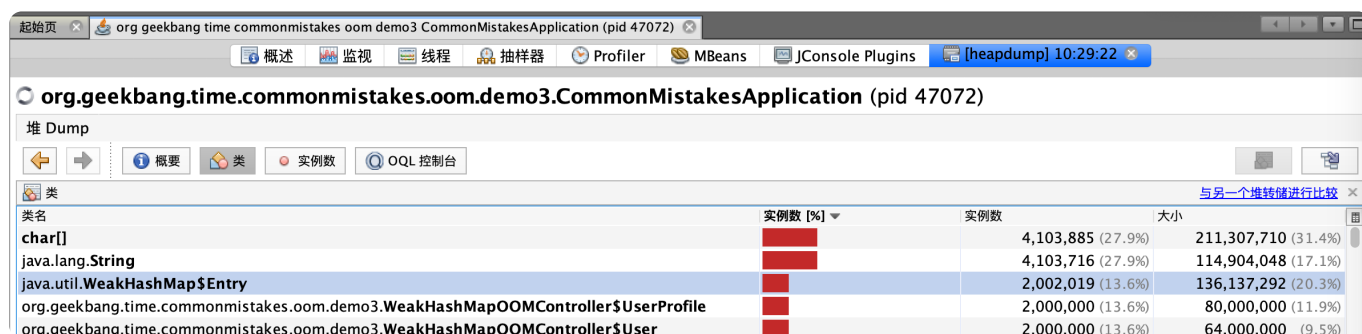
[复制代码](#)

```

1 Exception in thread "http-nio-45678-exec-1" java.lang.OutOfMemoryError: GC ove
2 Exception in thread "Catalina-utility-2" java.lang.OutOfMemoryError: GC overhe

```

我们来分析一下这个问题。进行堆转储后可以看到，堆内存中有 200 万个 UserProfile 和 User：



类名	实例数 [%]	实例数	大小
char[]		4,103,885 (27.9%)	211,307,710 (31.4%)
java.lang.String		4,103,716 (27.9%)	114,904,048 (17.1%)
java.util.WeakHashMap\$Entry		2,002,019 (13.6%)	136,137,292 (20.3%)
org.geekbang.time.commonmistakes.oom.demo3.WeakHashMapOOMController\$UserProfile		2,000,000 (13.6%)	80,000,000 (11.9%)
org.geekbang.time.commonmistakes.oom.demo3.WeakHashMapOOMController\$User		2,000,000 (13.6%)	64,000,000 (9.5%)

如下是 User 和 UserProfile 类的定义，需要注意的是，WeakHashMap 的 Key 是 User 对象，而其 Value 是 UserProfile 对象，持有了 User 的引用：

[复制代码](#)

```

1 @Data


```

```

2  @AllArgsConstructor
3  @NoArgsConstructor
4  class User {
5      private String name;
6  }
7
8
9  @Data
10 @AllArgsConstructor
11 @NoArgsConstructor
12 public class UserProfile {
13     private User user;
14     private String location;
15 }

```

没错，这就是问题的所在。分析一下 WeakHashMap 的源码，你会发现 WeakHashMap 和 HashMap 的最大区别，是 Entry 对象的实现。接下来，我们暂且忽略 HashMap 的实现，来看下 Entry 对象：

 复制代码

```

1  private static class Entry<K,V> extends WeakReference<Object> ...
2  /**
3   * Creates new entry.
4   */
5  Entry(Object key, V value,
6        ReferenceQueue<Object> queue,
7        int hash, Entry<K,V> next) {
8      super(key, queue);
9      this.value = value;
10     this.hash = hash;
11     this.next = next;
12 }

```

Entry 对象继承了 WeakReference，Entry 的构造函数调用了 super (key,queue)，这是父类的构造函数。其中，key 是我们执行 put 方法时的 key；queue 是一个 ReferenceQueue。如果你了解 Java 的引用就会知道，被 GC 的对象会被丢进这个 queue 里面。

再来看看对象被丢进 queue 后是如何被销毁的：

 复制代码

```

1  public V get(Object key) {

```



```

2     Object k = maskNull(key);
3     int h = hash(k);
4     Entry<K,V>[] tab = getTable();
5     int index = indexFor(h, tab.length);
6     Entry<K,V> e = tab[index];
7     while (e != null) {
8         if (e.hash == h && eq(k, e.get()))
9             return e.value;
10        e = e.next;
11    }
12    return null;
13 }
14
15 private Entry<K,V>[] getTable() {
16     expungeStaleEntries();
17     return table;
18 }
19
20 /**
21  * Expunges stale entries from the table.
22  */
23 private void expungeStaleEntries() {
24     for (Object x; (x = queue.poll()) != null; ) {
25         synchronized (queue) {
26             @SuppressWarnings("unchecked")
27             Entry<K,V> e = (Entry<K,V>) x;
28             int i = indexFor(e.hash, table.length);
29
30             Entry<K,V> prev = table[i];
31             Entry<K,V> p = prev;
32             while (p != null) {
33                 Entry<K,V> next = p.next;
34                 if (p == e) {
35                     if (prev == e)
36                         table[i] = next;
37                     else
38                         prev.next = next;
39                     // Must not null out e.next;
40                     // stale entries may be in use by a HashIterator
41                     e.value = null; // Help GC
42                     size--;
43                     break;
44                 }
45                 prev = p;
46                 p = next;
47             }
48         }
49     }
50 }

```

从源码中可以看到，每次调用 get、put、size 等方法时，都会从 queue 里拿出所有已经被 GC 掉的 key 并删除对应的 Entry 对象。我们再回顾下这个逻辑：

put 一个对象进 Map 时，它的 key 会被封装成弱引用对象；

发生 GC 时，弱引用的 key 被发现并放入 queue；


调用 get 等方法时，扫描 queue 删除 key，以及包含 key 和 value 的 Entry 对象。

WeakHashMap 的 Key 虽然是弱引用，但是其 Value 却持有 Key 中对象的强引用，Value 被 Entry 引用，Entry 被 WeakHashMap 引用，最终导致 Key 无法回收。解决方案就是让 Value 变为弱引用，使用 WeakReference 来包装 UserProfile 即可：

 复制代码

```
1 private Map<User, WeakReference<UserProfile>> cache2 = new WeakHashMap<>();
2
3 @GetMapping("right")
4 public void right() {
5     String userName = "zhuye";
6     //间隔1秒定时输出缓存中的条目数
7     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(
8         () -> log.info("cache size:{}", cache2.size()), 1, 1, TimeUnit.SECONDS);
9     LongStream.rangeClosed(1, 2000000).forEach(i -> {
10         User user = new User(userName + i);
11         //这次，我们使用弱引用来包装UserProfile
12         cache2.put(user, new WeakReference(new UserProfile(user, "location" + i)));
13     });
14 }
```

重新运行程序，从日志中观察到 cache size 不再是固定的 200 万，而是在不断减少，甚至在手动 GC 后所有的 Entry 都被回收了：

 复制代码

```
1 [10:40:05.792] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
2 [10:40:05.795] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
3 [10:40:06.773] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
4 ...
5 [10:40:20.742] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
6 [10:40:22.862] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
7 [10:40:22.865] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
8 [10:40:23.779] [pool-3-thread-1] [INFO] [t.c.o.demo3.WeakHashMapOOMController
9 //手动进行GC
10 [t.c.o.demo3.WeakHashMapOOMController:40] - cache size:0
```

当然，还有一种办法就是，让 Value 也就是 UserProfile 不再引用 Key，而是重新 new 出一个新的 User 对象赋值给 UserProfile：

复制代码

```
1 @GetMapping("right2")
2 public void right2() {
3     String userName = "zhuye";
4     ...
5     User user = new User(userName + i);
6     cache.put(user, new UserProfile(new User(user.getName()), "location" +
7 });
8 }
```

此外，Spring 提供的 [ConcurrentReferenceHashMap](#) 类可以使用弱引用、软引用做缓存，Key 和 Value 同时被软引用或弱引用包装，也能解决相互引用导致的数据不能释放问题。与 WeakHashMap 相比，ConcurrentReferenceHashMap 不但性能更好，还可以确保线程安全。你可以自己做实验测试下。

Tomcat 参数配置不合理导致 OOM

我们再来看看第三个案例。有一次运维同学反馈，有个应用在业务量大的情况下会出现假死，日志中也有大量 OOM 异常：

复制代码

```
1 [13:18:17.597] [http-nio-45678-exec-70] [ERROR] [ache.coyote.http11.Http11NioP
2 java.lang.OutOfMemoryError: Java heap space
```

于是，我让运维同学进行生产堆 Dump。通过 MAT 打开 dump 文件后，我们一眼就看到 OOM 的原因是，有接近 1.7GB 的 byte 数组分配，而 JVM 进程的最大堆内存我们只配置了 2GB：

Overview	Histogram			
Class Name	Objects	Shallow Heap	Retained Heap	
<Regex>	<Numeric>	<Numeric>	<Numeric>	
byte[]	6,994	1,624,814,592		
char[]	94,887	12,492,336		

通过查看引用可以发现，大量引用都是 Tomcat 的工作线程。大部分工作线程都分配了两个 10M 左右的数组，100 个左右工作线程吃满了内存。第一个红框是 Http11InputBuffer，其 buffer 大小是 10008192 字节；而第二个红框的 Http11OutputBuffer 的 buffer，正好占用 10000000 字节：

Class Name	Ref. Objects	Shallow Heap	Ref. Shallow Heap	Retained Heap
org.apache.tomcat.util.threads.TaskThread @ 0x786b8ad80 http-nio-45678-exec-77 Thread	20	128	30,049,736	5,192
org.apache.tomcat.util.threads.TaskThread @ 0x781cfd78 http-nio-45678-exec-4 Thread	20	128	30,049,736	5,264
org.apache.tomcat.util.threads.TaskThread @ 0x781eb1d28 http-nio-45678-exec-75 Thread	17	128	30,041,352	9,608
org.apache.tomcat.util.threads.TaskThread @ 0x781cf52a0 http-nio-45678-exec-13 Thread	1,105	128	20,072,256	6,568
org.apache.tomcat.util.threads.TaskThread @ 0x7861e6f88 http-nio-45678-exec-97 Thread	21	128	20,049,760	15,776
org.apache.tomcat.util.threads.TaskThread @ 0x781cea038 http-nio-45678-exec-31 Thread	40	128	20,042,096	7,552
org.apache.tomcat.util.threads.TaskThread @ 0x7861e96b0 http-nio-45678-exec-95 Thread	38	128	20,042,048	7,552
org.apache.tomcat.util.threads.TaskThread @ 0x781d2f358 http-nio-45678-exec-66 Thread	28	128	20,041,824	6,488
Java Local> org.apache.tomcat.util.net.NioChannel @ 0x753362390	1	32	10,008,208	32
appReadBufHandler org.apache.coyote.http11.Http11InputBuffer @ 0x774db8460	1	88	10,008,208	448
byteBuffer java.nio.HeapByteBuffer @ 0x774dc8b18	1	48	10,008,208	48
hb byte[10008192] @ 0x74c7f1198 GET /impropermaxheadersize/oom HTTP/1.1..host:localhost:45678...	1	10,008,208	10,008,208	10,008,208
Java Local> org.apache.coyote.Response @ 0x774dbab30	17	112	10,000,560	3,032
outputBuffer org.apache.coyote.http11.Http11OutputBuffer @ 0x774dbb5a8	3	56	10,000,072	10,000,632
headerBuffer java.nio.HeapByteBuffer @ 0x774dbb5e0	1	48	10,000,016	10,000,064
hb byte[10000000] @ 0x751dcda8 HTTP/1.1 200 ..Content-Type: text/plain;charset=UTF-8..Content-Leng	1	10,000,016	10,000,016	10,000,016
filterLibrary org.apache.coyote.http11.OutputFilter[4] @ 0x774dbb610	2	32	56	432
Total: 2 entries				
headers org.apache.tomcat.util.http.MimeHeaders @ 0x774dbaba0	14	24	488	2,744
Total: 2 entries				
Java Local> org.apache.tomcat.util.net.NioEndpoint\$NioSocketWrapper @ 0x7533622f0	2	160	16,416	600
Java Local> org.apache.catalina.connector.Request @ 0x774db9b28	2	168	8,240	31,144
Java Local> org.apache.catalina.connector.Response @ 0x774dbaaf0	2	64	8,232	25,440
Java Local> org.apache.coyote.Request @ 0x774db84b8	1	176	80	3,016
Java Local> sun.nio.ch.SelectionKeyImpl @ 0x7533622c8	2	40	64	40
threadLocals java.lang.ThreadLocal\$ThreadLocalMap @ 0x781d2f530	1	24	24	4,992
Total: 8 entries				
org.apache.tomcat.util.threads.TaskThread @ 0x7861f5f50 http-nio-45678-exec-85 Thread	29	128	20,041,792	13,152

我们先来看看第一个 Http11InputBuffer 为什么会占用这么多内存。查看 Http11InputBuffer 类的 init 方法注意到，其中一个初始化方法会分配 headerBufferSize+readBuffer 大小的内存：

复制代码

```
1 void init(SocketWrapperBase<?> socketWrapper) {
2
3     wrapper = socketWrapper;
4     wrapper.setAppReadBufHandler(this);
5
6     int bufLength = headerBufferSize +
7         wrapper.getSocketBufferHandler().getReadBuffer().capacity();
8     if (byteBuffer == null || byteBuffer.capacity() < bufLength) {
9         byteBuffer = ByteBuffer.allocate(bufLength);
10        byteBuffer.position(0).limit(0);
11    }
12 }
```

在 Tomcat 文档中有提到，这个 Socket 的读缓冲，也就是 readBuffer 默认是 8192 字节。显然，问题出在了 headerBufferSize 上：

socket.getAppReadBufSize


(int)Each connection that is opened up in Tomcat get associated with a read ByteBuffer. This attribute controls the size of this buffer. By default this read buffer is sized at 8192 bytes. For lower concurrency, you can increase this to buffer more data. For an extreme amount of keep alive connections, decrease this number or increase your heap size.

向上追溯初始化 Http11InputBuffer 的 Http11Processor 类，可以看到，传入的 headerBufferSize 配置的是 MaxHttpHeaderSize：

 复制代码

```
1 inputBuffer = new Http11InputBuffer(request, protocol.getMaxHttpHeaderSize(),
2     protocol.getRejectIllegalHeaderName(), httpParser);
```

Http11OutputBuffer 中的 buffer 正好占用了 10000000 字节，这又是为什么？通过 Http11OutputBuffer 的构造方法，可以看到它是直接根据 headerBufferSize 分配了固定大小的 headerBuffer：

 复制代码

```
1 protected Http11OutputBuffer(Response response, int headerBufferSize){
2     ...
3     headerBuffer = ByteBuffer.allocate(headerBufferSize);
4 }
```

那么我们就可以想到，一定是应用把 Tomcat 头相关的参数配置为 10000000 了，使得每一个请求对于 Request 和 Response 都占用了 20M 内存，最终在并发较多的情况下引起了 OOM。

果不其然，查看项目代码发现配置文件中有这样的配置项：

 复制代码

```
1 server.max-http-header-size=10000000
```

翻看源码提交记录可以看到，当时开发同学遇到了这样的异常：

 复制代码

```
1 java.lang.IllegalArgumentException: Request header is too large
```

于是他就到网上搜索了一下解决方案，随意将 `server.max-http-header-size` 修改为了一个超大值，期望永远不会再出现类似问题。但，没想到这个修改却引起了这么大的问题。把这个参数改为比较合适的 20000 再进行压测，我们就可以发现应用的各项指标都比较稳定。

这个案例告诉我们，**一定要根据实际需求来修改参数配置，可以考虑预留 2 到 5 倍的量。容量类的参数背后往往代表了资源，设置超大的参数就有可能占用不必要的资源，在并发量大的时候因为资源大量分配导致 OOM。**

重点回顾

今天，我从内存分配意识的角度和你分享了 OOM 的问题。通常而言，Java 程序的 OOM 有如下几种可能。

一是，我们的程序确实需要超出 JVM 配置的内存上限的内存。不管是程序实现的不合理，还是因为各种框架对数据的重复处理、加工和转换，相同的数据在内存中不一定只占用一份空间。针对内存量使用超大的业务逻辑，比如缓存逻辑、文件上传下载和导出逻辑，我们在做容量评估时，可能还需要实际做一下 Dump，而不是进行简单的假设。

二是，出现内存泄露，其实就是我们认为没有用的对象最终会被 GC，但却没有。GC 并不会回收强引用对象，我们可能经常在程序中定义一些容器作为缓存，但如果容器中的数据无限增长，要特别小心最终会导致 OOM。使用 WeakHashMap 是解决这个问题的好办法，但值得注意的是，如果强引用的 Value 有引用 Key，也无法回收 Entry。

三是，不合理的资源需求配置，在业务量小的时候可能不会出现问题，但业务量一大可能很快就会撑爆内存。比如，随意配置 Tomcat 的 `max-http-header-size` 参数，会导致一个请求使用过多的内存，请求量大的时候出现 OOM。在进行参数配置的时候，我们要认识到，很多限制类参数限制的是背后资源的使用，资源始终是有限的，需要根据实际需求来合理设置参数。

最后我想说的是，在出现 OOM 之后，也不用过于紧张。我们可以根据错误日志中的异常信息，再结合 `jstat` 等命令行工具观察内存使用情况，以及程序的 GC 日志，来大致定位出现 OOM 的内存区块和类型。其实，我们遇到的 90% 的 OOM 都是堆 OOM，对 JVM 进程进行堆内存 Dump，或使用 `jmap` 命令分析对象内存占用排行，一般都可以很容易定位到问题。

这里，我建议你为生产系统的程序配置 JVM 参数启用详细的 GC 日志，方便观察垃圾收集器的行为，并开启 `HeapDumpOnOutOfMemoryError`，以便在出现 OOM 时能自动 Dump 留下第一问题现场。对于 JDK8，你可以这么设置：

 复制代码

```
1 XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=. -XX:+PrintGCDateStamps -XX:+l
```

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

思考与讨论

1. Spring 的 `ConcurrentReferenceHashMap`，针对 Key 和 Value 支持软引用和弱引用两种方式。你觉得哪种方式更适合做缓存呢？
2. 当我们需要动态执行一些表达式时，可以使用 Groovy 动态语言实现：new 出一个 `GroovyShell` 类，然后调用 `evaluate` 方法动态执行脚本。这种方式的问题是，会重复产生大量的类，增加 Metaspace 区的 GC 负担，有可能会引起 OOM。你知道如何避免这个问题吗？

针对 OOM 或内存泄露，你还遇到过什么案例吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 用好Java 8的日期时间类，少踩一些“老三样”的坑

下一篇 18 | 当反射、注解和泛型遇到OOP时，会有哪些坑？

精选留言 (10)

 写留言

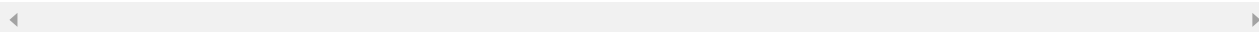


一个汉子~

2020-04-18

针对第二点，可以先compile，然后在内存中保存，脚本内容的hash作为key，compile结果作为value，用ConcurrentReferenceHashMap保存
同样的风险还出现在表达式框架aviator中

作者回复: 



 1

 3



Darren

2020-04-20

试着回到下问题：
第一个：

肯定是软引用，因为弱引用是只要GC执行，扫描到就被回收，缓存的作用是为了提高速度，要有一定的存在周期；如果是弱引用，每次GC执行，缓存被回收，缓存命中率超低，完全达不到缓存的作用，而又要维护缓存和DB的数据一致性问题，得不偿失。...

展开 ▾

作者回复: 源码里我也有一个例子，思路是不要每次都evaluate脚本而是把脚本转变为一个方法parse后缓存起来这个Script，以后直接invokeMethod来使用

1 评论

2 点赞



2020-04-18

应该用哪种引用，首先考虑的肯定是四大引用的区别。

- 1.强引用：最常见的一种，只要该引用存在，就不会被GC。
- 2.软引用：内存空间不足时，进行回收。
- 3.弱引用：当JVM进行GC时，则进行回收，无论内存是否充足。
- 4.虚引用：这个不提了，因为我也完全不懂。...

展开 ▾

作者回复: 不错

1 评论

2 点赞



自由港

2020-04-18

关于第二个问题限制了metadata的堆大小，发现就可自动回收了

1 评论

2 点赞



一个汉子~

2020-04-18

之前还遇到一个，一个导出功能，拥有管理员权限的人几乎没有限制，造成了全表查，再加上框架禁止join，所以又把外键拉出来做了一次in查询，也是全表扫，大量的Bo对象和超长sql，直接把系统oom了

作者回复: 很常见的问题，还有包括参数未传导致mybatis条件没有拼接上去，导致全表查询的oom

1 评论

2 点赞



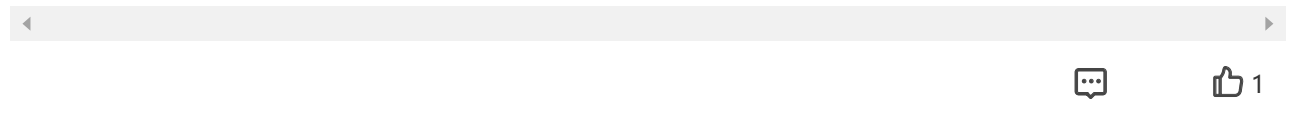
ddosyang

2020-04-20

老师想问一下，在WeakHashMap的那个例子里，可不可以直接用String name当作Key，而不是用User做Key。这样是不是也可以解决问题？

展开 ▾

作者回复: 是，不过这就改了设计了



pedro

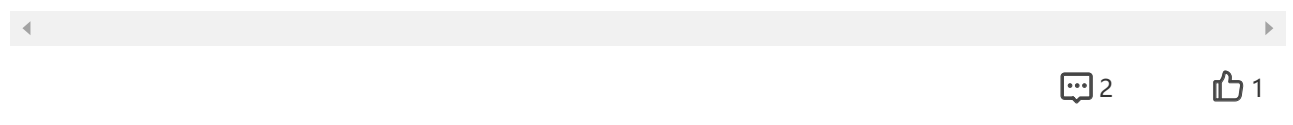
2020-04-18

问题一，弱引用是在内存不足时被 gc 掉，而软引用是只要 gc 就回收掉，自然就不能用来做缓存，否则动不动就缓存失效，数据库怕是要被玩坏哦，因为适合做缓存的是弱引用。

问题二，没用过 groovy，希望看到别人的解答。😊

展开 ▾

作者回复: 不太对，可以再查一些资料或做一些实验看下软和弱的区别



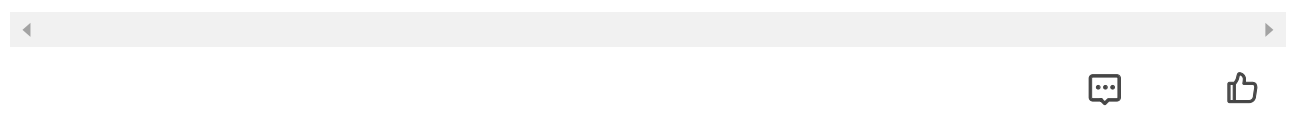
汝林外史

2020-04-20

1. 对于老师说的autoComplete的场景是不是Trie树更适合一些？
2. 这个WeakReference可能导致内存溢出的典型就是ThreadLocal，虽然ThreadLocalMap的entry的key是weakReference，但是value是强引用，当用线程池的时候，就会内存溢出，还是要自己remove才行。
3. 对于问题1，应该用软引用更好一些，用弱引用总是被gc回收就失去了缓存的意义。...

展开 ▾

作者回复: 1. 是的，这种场景字典树更合适，不过我这边都是拿着实际的案例整理成文的，主要是希望告知大家我们认为的一份数据在程序中不一定是一份这个误区



yang

2020-04-20

技术方案也够奇葩的-----怎么设计出来的-----~::~





Geek_3b1096

2020-04-18

周六第一件事跟上老师进度

展开 ▾

