二

30 GC 疑难情况问题排查与分析（下篇）

Weak、Soft 及 Phantom 引用

另一类影响 GC 的问题是程序中的 non-strong 引用。虽然这类引用在很多情况下可以避免出现 OutOfMemoryError，但过量使用也会对 GC 造成严重的影响，反而降低系统性能。

弱引用的缺点

首先，弱引用（weak reference）是可以被 GC 强制回收的。当垃圾收集器发现一个弱可达对象（weakly reachable，即指向该对象的引用只剩下弱引用）时，就会将其置入相应的 ReferenceQueue 中，变成可终结的对象。之后可能会遍历这个 reference queue，并执行相应的清理。典型的示例是清除缓存中不再引用的 KEY。

当然，在这个时候我们还可以将该对象赋值给新的强引用，在最后终结和回收前，GC 会再次确认该对象是否可以安全回收。因此，弱引用对象的回收过程是横跨多个 GC 周期的。

实际上弱引用使用的很多。大部分缓存框架都是基于弱引用实现的，所以虽然业务代码中没有直接使用弱引用，但程序中依然会大量存在。

其次，软引用（soft reference）比弱引用更难被垃圾收集器回收。回收软引用没有确切的时间点，由 JVM 自己决定。一般只会在即将耗尽可用内存时，才会回收软引用，以作最后手段。这意味着可能会有更频繁的 Full GC，暂停时间也比预期更长，因为老年代中的存活对象会很多。

最后，使用虚引用（phantom reference）时，必须手动进行内存管理，以标识这些对象是否可以安全地回收。表面上看起来很正常，但实际上并不是这样。javadoc 中写道：

> In order to ensure that a reclaimable object remains so, the referent of a phantom reference may not be retrieved: The get method of a phantom reference always returns null.
>
> 为了防止可回收对象的残留，虚引用对象不应该被获取：phantom reference 的 get 方法返回值永远是 null。

令人惊讶的是，很多开发者忽略了下一段内容（**这才是重点**）：

> Unlike soft and weak references，phantom references are not automatically cleared by the garbage collector as they are enqueued. An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable.
>
> 与软引用和弱引用不同，虚引用不会被 GC 自动清除，因为他们被存放到队列中。通过虚引用可达的对象会继续留在内存中，直到调用此引用的 clear 方法，或者引用自身变为不可达。

也就是说，我们必须手动调用 clear() 来清除虚引用，否则可能会造成 OutOfMemoryError 而导致 JVM 挂掉。使用虚引用的理由是，对于用编程手段来跟踪某个对象何时变为不可达对象，这是唯一的常规手段。和软引用/弱引用不同的是，我们不能“复活”虚可达（phantom-reachable）对象。

示例

让我们看一个弱引用示例，其中创建了大量的对象，并在 Minor GC 中完成回收。和前面一样，修改提升阈值。可以使用下列 JVM 参数：

```
-Xmx24m -XX:NewSize=16m -XX:MaxTenuringThreshold=1
```

此时 GC 日志如下所示：

```
2.330: [GC (Allocation Failure)  20933K->8229K(22528K), 0.0033848 secs]
2.335: [GC (Allocation Failure)  20517K->7813K(22528K), 0.0022426 secs]
2.339: [GC (Allocation Failure)  20101K->7429K(22528K), 0.0010920 secs]
2.341: [GC (Allocation Failure)  19717K->9157K(22528K), 0.0056285 secs]
2.348: [GC (Allocation Failure)  21445K->8997K(22528K), 0.0041313 secs]
2.354: [GC (Allocation Failure)  21285K->8581K(22528K), 0.0033737 secs]
2.359: [GC (Allocation Failure)  20869K->8197K(22528K), 0.0023407 secs]
2.362: [GC (Allocation Failure)  20485K->7845K(22528K), 0.0011553 secs]
2.365: [GC (Allocation Failure)  20133K->9501K(22528K), 0.0060705 secs]
2.371: [Full GC (Ergonomics)     9501K->2987K(22528K), 0.0171452 secs]
```

可以看到，Full GC 的次数很少。但如果使用弱引用来指向创建的对象，使用 JVM 参数 `-Dweak.refs=true`，则情况会发生明显变化。使用弱引用的原因很多，比如在 `weak hash map` 中将对象作为 Key 的情况。在任何情况下，使用弱引用都可能会导致以下情形：

```
2.059: [Full GC (Ergonomics)  20365K->19611K(22528K), 0.0654090 secs]
```

```
2.125: [Full GC (Ergonomics)  20365K->19711K(22528K), 0.0707499 secs]
2.196: [Full GC (Ergonomics)  20365K->19798K(22528K), 0.0717052 secs]
2.268: [Full GC (Ergonomics)  20365K->19873K(22528K), 0.0686290 secs]
2.337: [Full GC (Ergonomics)  20365K->19939K(22528K), 0.0702009 secs]
2.407: [Full GC (Ergonomics)  20365K->19995K(22528K), 0.0694095 secs]
```

可以看到，发生了多次 Full GC，比起前一节的示例，GC 时间增加了一个数量级！

这是过早提升的另一个例子，但这次情况更加棘手：问题的根源在于弱引用。这些临死的对象，在添加弱引用之后，被提升到了老年代。但是，他们现在陷入另一次 GC 循环之中，所以需要对其做一些适当的清理。

像之前一样，最简单的办法是增加年轻代的大小，例如指定 JVM 参数 `-Xmx64m -XX:NewSize=32m`：

```
2.328: [GC (Allocation Failure)  38940K->13596K(61440K), 0.0012818 secs]
2.332: [GC (Allocation Failure)  38172K->14812K(61440K), 0.0060333 secs]
2.341: [GC (Allocation Failure)  39388K->13948K(61440K), 0.0029427 secs]
2.347: [GC (Allocation Failure)  38524K->15228K(61440K), 0.0101199 secs]
2.361: [GC (Allocation Failure)  39804K->14428K(61440K), 0.0040940 secs]
2.368: [GC (Allocation Failure)  39004K->13532K(61440K), 0.0012451 secs]
```

这时候，对象在 Minor GC 中就被回收了。

更坏的情况是使用软引用，例如这个软引用示例程序。如果程序不是即将发生 OutOfMemoryError，软引用对象就不会被回收。在示例程序中，用软引用替代弱引用，立即出现了更多的 Full GC 事件：

```
2.162: [Full GC (Ergonomics)  31561K->12865K(61440K), 0.0181392 secs]
2.184: [GC (Allocation Failure)  37441K->17585K(61440K), 0.0024479 secs]
2.189: [GC (Allocation Failure)  42161K->27033K(61440K), 0.0061485 secs]
2.195: [Full GC (Ergonomics)  27033K->14385K(61440K), 0.0228773 secs]
2.221: [GC (Allocation Failure)  38961K->20633K(61440K), 0.0030729 secs]
2.227: [GC (Allocation Failure)  45209K->31609K(61440K), 0.0069772 secs]
2.234: [Full GC (Ergonomics)  31609K->15905K(61440K), 0.0257689 secs]
```

最有趣的是虚引用示例中的虚引用，使用同样的 JVM 参数启动，其结果和弱引用示例非常相似。实际上，Full GC 暂停的次数会小得多，原因前面说过，他们有不同的终结方式。

如果禁用虚引用清理，增加 JVM 启动参数（`-Dno.ref.clearing=true`），则可以看到：

```
4.180: [Full GC (Ergonomics)  57343K->57087K(61440K), 0.0879851 secs]
```

```
4.269: [Full GC (Ergonomics)  57089K->57088K(61440K), 0.0973912 secs]
4.366: [Full GC (Ergonomics)  57091K->57089K(61440K), 0.0948099 secs]
```

主线程中很快抛出异常：

```
java.lang.OutOfMemoryError: Java heap space
```

使用虚引用时要小心谨慎，并及时清理虚可达对象。如果不清理，很可能会发生 OutOfMemoryError。

请相信我们的经验教训：处理 reference queue 的线程中如果没 catch 住异常，系统很快就会被整挂了。

使用非强引用的影响

建议使用 JVM 参数 `-XX:+PrintReferenceGC` 来看看各种引用对 GC 的影响。如果将此参数用于启动弱引用示例，将会看到：

```
2.173: [Full GC (Ergonomics)
        2.234: [SoftReference, 0 refs, 0.0000151 secs]
        2.234: [WeakReference, 2648 refs, 0.0001714 secs]
        2.234: [FinalReference, 1 refs, 0.0000037 secs]
        2.234: [PhantomReference, 0 refs, 0 refs, 0.0000039 secs]
        2.234: [JNI Weak Reference, 0.0000027 secs]
            [PSYoungGen: 9216K->8676K(10752K)]
            [ParOldGen: 12115K->12115K(12288K)]
            21331K->20792K(23040K),
        [Metaspace: 3725K->3725K(1056768K)],
        0.0766685 secs]
    [Times: user=0.49 sys=0.01, real=0.08 secs]
2.250: [Full GC (Ergonomics)
        2.307: [SoftReference, 0 refs, 0.0000173 secs]
        2.307: [WeakReference, 2298 refs, 0.0001535 secs]
        2.307: [FinalReference, 3 refs, 0.0000043 secs]
        2.307: [PhantomReference, 0 refs, 0 refs, 0.0000042 secs]
        2.307: [JNI Weak Reference, 0.0000029 secs]
            [PSYoungGen: 9215K->8747K(10752K)]
            [ParOldGen: 12115K->12115K(12288K)]
            21331K->20863K(23040K),
        [Metaspace: 3725K->3725K(1056768K)],
        0.0734832 secs]
    [Times: user=0.52 sys=0.01, real=0.07 secs]
2.323: [Full GC (Ergonomics)
        2.383: [SoftReference, 0 refs, 0.0000161 secs]
        2.383: [WeakReference, 1981 refs, 0.0001292 secs]
        2.383: [FinalReference, 16 refs, 0.0000049 secs]
        2.383: [PhantomReference, 0 refs, 0 refs, 0.0000040 secs]
```

```
        2.383: [JNI Weak Reference, 0.0000027 secs]
            [PSYoungGen: 9216K->8809K(10752K)]
            [ParOldGen: 12115K->12115K(12288K)]
            21331K->20925K(23040K),
        [Metaspace: 3725K->3725K(1056768K)],
        0.0738414 secs]
    [Times: user=0.52 sys=0.01, real=0.08 secs]
```

只有确定 GC 对应用的吞吐量和延迟造成影响之后，才应该花心思来分析这些信息，审查这部分日志。通常情况下，每次 GC 清理的引用数量都是很少的，大部分情况下为 0。

如果 GC 花了较多时间来清理这类引用，或者清除了很多的此类引用，就需要进一步观察和分析了。

解决方案

如果程序确实碰到了 `mis-` 、 `ab-` 等问题或者滥用 weak/soft/phantom 引用，一般都要修改程序的实现逻辑。每个系统不一样，因此很难提供通用的指导建议，但有一些常用的经验办法：

- 弱引用（Weak references）：如果某个内存池的使用量增大，造成了性能问题，那么增加这个内存池的大小（可能也要增加堆内存的最大容量）。如同示例中所看到的，增加堆内存的大小，以及年轻代的大小，可以减轻症状。
- 软引用（Soft references）：如果确定问题的根源是软引用，唯一的解决办法是修改程序源码，改变内部实现逻辑。
- 虚引用（Phantom references）：请确保在程序中调用了虚引用的 clear 方法。编程中很容易忽略某些虚引用，或者清理的速度跟不上生产的速度，又或者清除引用队列的线程挂了，就会对 GC 造成很大压力，最终可能引起 OutOfMemoryError。

其他性能问题的案例

前面介绍了最常见的 GC 性能问题，本节介绍一些不常见、但也可能会导致系统故障的问题。

RMI 与 GC

如果系统提供或者消费 RMI 服务，则 JVM 会定期执行 Full GC 来确保本地未使用的对象在另一端也不占用空间。即使你的代码中没有发布 RMI 服务，但第三方或者工具库也可能会打开 RMI 终端。最常见的元凶是 JMX，如果通过 JMX 连接到远端，底层则会使用 RMI 发布数据。

问题是有很多不必要的周期性 Full GC。查看老年代的使用情况，一般是没有内存压力，其中还存在大量的空闲区域，但 Full GC 就是被触发了，也就会暂停所有的应用线程。

这种周期性调用 System.gc() 删除远程引用的行为，是在 sun.rmi.transport.ObjectTable 类中，通过 sun.misc.GC.requestLatency(long gcInterval) 调用的。

对许多应用来说，根本没必要，甚至对性能有害。禁止这种周期性的 GC 行为，可以使用以下 JVM 参数：

```
java -Dsun.rmi.dgc.server.gcInterval=9223372036854775807L
    -Dsun.rmi.dgc.client.gcInterval=9223372036854775807L
    com.yourcompany.YourApplication
```

这让 Long.MAX_VALUE 毫秒之后，才调用 System.gc()，实际运行的系统可能永远都不会触发。

```
// ObjectTable.class
private static final long gcInterval =
((Long)AccessController.doPrivileged(
    new GetLongAction("sun.rmi.dgc.server.gcInterval", 3600000L)
    )).longValue();
```

可以看到，默认值为 3600000L，也就是 1 小时触发一次 Full GC。

另一种方式是指定 JVM 参数 `-XX:+DisableExplicitGC`，禁止显式地调用 System.gc()。但我们**强烈反对**这种方式，因为我们不清楚这么做是否埋有地雷，例如第三方库里需要显式调研。

JVMTI tagging 与 GC

如果在程序启动时指定了 Java Agent（`-javaagent`），Agent 就可以使用 JVMTI tagging 标记堆中的对象。如果 tagging 标记了大量的对象，很可能会引起 GC 性能问题，导致延迟增加，以及吞吐量降低。

问题发生在 native 代码中，`JvmtiTagMap::do_weak_oops` 在每次 GC 时，都会遍历所有标标记（tag），并执行一些比较耗时的操作。更坑的是，这种操作是串行执行的。

如果存在大量的标记，就意味着 GC 时有很大一部分工作是单线程执行的，GC 暂停时间可能会增加一个数量级。

检查是否因为 Java Agent 增加了 GC 暂停时间，可以使用诊断参数 `-XX:+TraceJVMTIObjectTagging` 。

启用跟踪之后，可以估算出内存中 的标记映射了多少 native 内存，以及遍历所消耗的时间。

如果你不是 需要使用的这个 agent 的作者，那一般是搞不定这类问题的。除了提 Bug 之外你什么都做不了。如果发生了这种情况，请建议厂商清理不必要的标记。（以前我们就在生产环境里发现 APM 厂商的 Agent 偶尔会导致 JVM OOM 崩溃。）

巨无霸对象的分配（Humongous Allocations）

如果使用 G1 垃圾收集算法，会产生一种巨无霸对象引起的 GC 性能问题。

> **说明**：在 G1 中，巨无霸对象是指所占空间超过一个小堆区（region）50% 的对象。

频繁地创建巨无霸对象，无疑会造成 GC 的性能问题，看看 G1 的处理方式：

- 如果某个 region 中含有巨无霸对象，则巨无霸对象后面的空间将不会被分配。如果所有巨无霸对象都超过某个比例，则未使用的空间就会引发内存碎片问题。
- G1 没有对巨无霸对象进行优化。这在 JDK 8 以前是个特别棘手的问题——在 **Java 1.8u40** 之前的版本中，巨无霸对象所在 region 的回收只能在 Full GC 中进行。最新版本的 Hotspot JVM，在 marking 阶段之后的 cleanup 阶段中释放巨无霸区间，所以这个问题在新版本 JVM 中的影响已大大降低。

要监控是否存在巨无霸对象，可以打开 GC 日志，使用的命令如下：

```
java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
    -XX:+PrintReferenceGC -XX:+UseG1GC
    -XX:+PrintAdaptiveSizePolicy -Xmx128m
    MyClass
```

GC 日志中可能会发现这样的部分：

```
 0.106: [G1Ergonomics (Concurrent Cycles)
        request concurrent cycle initiation,
        reason: occupancy higher than threshold,
        occupancy: 60817408 bytes,
        allocation request: 1048592 bytes,
        threshold: 60397965 bytes (45.00 %),
        source: concurrent humongous allocation]
 0.106: [G1Ergonomics (Concurrent Cycles)
```

```
        request concurrent cycle initiation,
        reason: requested by GC cause,
        GC cause: G1 Humongous Allocation]
0.106: [G1Ergonomics (Concurrent Cycles)
        initiate concurrent cycle,
        reason: concurrent cycle initiation requested]
0.106: [GC pause (G1 Humongous Allocation)
        (young) (initial-mark)
        0.106: [G1Ergonomics (CSet Construction)
            start choosing CSet,
            _pending_cards: 0,
            predicted base
            time: 10.00 ms,
            remaining time: 190.00 ms,
            target pause time: 200.00 ms]
```

这样的日志就是证据，表明程序中确实创建了巨无霸对象。可以看到 G1 Humongous Allocation 是 GC 暂停的原因。再看前面一点的 `allocation request: 1048592 bytes`，可以发现程序试图分配一个 1048592 字节的对象，这要比巨无霸区域（2MB）的 50% 多出 16 个字节。

第一种解决方式，是修改 region size，以使得大多数的对象不超过 50%，也就不进行巨无霸对象区域的分配。G1 的 region 大小默认值在启动时根据堆内存的大小算出。但也可以指定参数来覆盖默认设置，`-XX:G1HeapRegionSize=XX`。指定的 region size 必须在 1~32MB 之间，还必须是 2 的幂（2^10=1024=1KB，2^20=1MB，所以 region size 只能是下列值之一：1m、2m、4m、8m、16m、32m）。

这种方式也有副作用，增加 region 的大小也就变相地减少了 region 的数量，所以需要谨慎使用，最好进行一些测试，看看是否改善了吞吐量和延迟。

更好的使用方式是，在程序中限制对象的大小，我们可以在运行时使用内存分析工具，展示出巨无霸对象的信息，以及分配时所在的堆栈跟踪信息。

总结

Java 作为一个通用平台，运行在 JVM 上的应用程序多种多样，其启动参数也有上百个，其中有很多会影响到 GC 和性能，所以调优 GC 性能的方法也有很多种。

但是我们也要时刻提醒自己：没有真正的银弹，能满足所有的性能调优指标。

我们需要做的，就是了解这些可能会出现问题的各个要点，掌握常见的排查分析方法和工具。

在碰到类似问题时知道是知其然知其所以然，深入理解 JVM/GC 的工作原理，熟练应用各

种手段，观察各种现象，收集各种有用的指标数据，进行定性和定量的分析，找到瓶颈，制定解决方案，进行调优和改进，提高应用系统的性能和稳定性。

上一页 下一页