GC日志解读与分析：千淘万漉虽辛苦，吹尽狂沙始到金

本章通过具体示例来演示如何输出GC日志，并对输出的日志信息进行解读分析，从中提取有用的信息。

一、本次演示的示例代码

为了演示需要，我们先来编写一段简单的Java代码：

```
package demo.jvm0204;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.LongAdder;
/*
演示GC日志生成与解读
*/
public class GCLogAnalysis {
    private static Random random = new Random();
    public static void main(String[] args) {
        // 当前毫秒时间戳
        long startMillis = System.currentTimeMillis();
        // 持续运行毫秒数; 可根据需要进行修改
        long timeoutMillis = TimeUnit.SECONDS.toMillis(1);
        // 结束时间戳
        long endMillis = startMillis + timeoutMillis;
        LongAdder counter = new LongAdder();
        System.out.println("正在执行...");
        // 缓存一部分对象; 进入老年代
```

```
        int cacheSize = 2000;
        Object[] cachedGarbage = new Object[cacheSize];
        // 在此时间范围内,持续循环
        while (System.currentTimeMillis() < endMillis) {
            // 生成垃圾对象
            Object garbage = generateGarbage(100*1024);
            counter.increment();
            int randomIndex = random.nextInt(2 * cacheSize);
            if (randomIndex < cacheSize) {
                cachedGarbage[randomIndex] = garbage;
            }
        }
        System.out.println("执行结束!共生成对象次数:" + counter.longValue());
    }

    // 生成对象
    private static Object generateGarbage(int max) {
        int randomSize = random.nextInt(max);
        int type = randomSize % 4;
        Object result = null;
        switch (type) {
            case 0:
                result = new int[randomSize];
                break;
            case 1:
                result = new byte[randomSize];
                break;
            case 2:
                result = new double[randomSize];
                break;
            default:
                StringBuilder builder = new StringBuilder();
                String randomString = "randomString-Anything";
                while (builder.length() < randomSize) {
                    builder.append(randomString);
                    builder.append(max);
                    builder.append(randomSize);
                }
                result = builder.toString();
                break;
        }
        return result;
    }
}
```

程序并不复杂，我们指定一个运行时间作为退出条件，时间一到自动退出循环。在 `generateGarbage` 方法中，我们用了随机数来生成各种类型的数组对象并返回。
在 `main` 方法中，我们用一个数组来随机存放一部分生成的对象，这样可以模拟让部分对象晋升到老年代。具体的持续运行时间和缓存对象个数，各位同学可以自己进行调整。

> 一般来说，Java中的大对象主要就是各种各样的数组，比如开发中最常见的字符串，实际上 `String` 内部就是使用字符数组 `char[]` 来存储的。

额外说一句：这个示例除了可以用来进行GC日志分析之外，稍微修改一下，还可以用作其他用途：

- 比如让缓存的对象变多，在限制堆内存的情况下，就可以模拟 `内存溢出` 。
- 增加运行时长，比如加到30分钟或者更长，我们就可以用前面介绍过的 `VisualVM` 等工具来实时监控和观察。
- 当然，我们也可以使用全局静态变量来缓存，用来模拟 `内存泄漏` ，以及进行堆内存Dump的试验和分析。
- 加大每次生成的数组的大小，可以用来模拟 `大对象/巨无霸对象` （大对象/巨无霸对象主要是G1中的概念，比如超过1MB的数组，具体情况在后面的课程中再进行探讨）。

二、常用的GC参数

我们从简单到复杂，一步一步来验证前面学习的知识，学会使用，加深巩固。

启动示例程序

如果是在IDEA、Eclipse等集成开发环境中，直接在文件中点击鼠标右键，选择"Run..."即可执行。
如果使用JDK命令行，则可以使用 `javac` 工具来编译，使用 `java` 命令来执行（还记得吗？JDK8以上版本，这两个命令可以合并成一个）：

```
$ javac demo/jvm0204/*.java
$ java demo.jvm0204.GCLogAnalysis
正在执行...
执行结束!共生成对象次数:1423
```

程序执行1秒钟就自动结束了，因为没有指定任何启动参数，所以输出的日志内容也很简单。
还记得我们在前面的 JVM 启动参数详解 章节中介绍的GC参数吗?
我们依次加上这些参数来看看效果。

输出GC日志详情

然后加上启动参数 `-XX:+PrintGCDetails` ，打印GC日志详情，再次执行示例。

> IDEA等集成开发环境可以在 “VM options” 中指定启动参数，参考前面的课程。 注意不要有多余的空格。

```

```

```
java -XX:+PrintGCDetails demo.jvm0204.GCLogAnalysis
```

执行结果摘录如下:

```
正在执行...
[GC (Allocation Failure)
  [PSYoungGen: 65081K->10728K(76288K)]
  65081K->27102K(251392K), 0.0112478 secs]
  [Times: user=0.03 sys=0.02, real=0.01 secs]
......此处省略了多行
[Full GC (Ergonomics)
  [PSYoungGen: 80376K->0K(872960K)]
  [ParOldGen: 360220K->278814K(481280K)]
  440597K->278814K(1354240K),
  [Metaspace: 3443K->3443K(1056768K)],
  0.0406179 secs]
  [Times: user=0.23 sys=0.01, real=0.04 secs]
执行结束!共生成对象次数:746
Heap
 PSYoungGen total 872960K, used 32300K [0x000000076ab00000, 0x00000007b0180000, 0x0000
  eden space 792576K, 4% used [0x000000076ab00000,0x000000076ca8b370,0x000000079b10000
  from space 80384K, 0% used [0x00000007a3800000,0x00000007a3800000,0x00000007a8680000
  to space 138240K, 0% used [0x000000079b100000,0x000000079b100000,0x00000007a3800000
 ParOldGen total 481280K, used 278814K [0x00000006c0000000, 0x00000006dd600000, 0x0000
  object space 481280K, 57% used [0x00000006c0000000,0x00000006d1047b10,0x00000006dd60
 Metaspace used 3449K, capacity 4494K, committed 4864K, reserved 1056768K
  class space used 366K, capacity 386K, committed 512K, reserved 1048576K
```

可以看到，使用启动参数 `-XX:+PrintGCDetails` ，发生GC时会输出相关的GC日志。

> 这个参数的格式为: `-XX:+` ，这是一个布尔值开关。

在程序执行完成后、JVM关闭前，还会输出各个内存池的使用情况， 从最后面的输出中可以看到。
下面我们来简单解读上面输出的堆内存信息。

** Heap 堆内存使用情况 **

```
PSYoungGen total 872960K, used 32300K [0x......)
 eden space 792576K, 4% used [0x......)
 from space 80384K, 0% used [0x......)
 to space 138240K, 0% used [0x......)
```

- PSYoungGen，年轻代总计 872960K，使用量 32300K，后面的方括号中是内存地址信息

    - 其中 eden space 占用了 792576K， 其中 4% used
    - 其中 from space 占用了 80384K， 其中 0% used
    - 其中 to space 占用了 138240K， 其中 0% used

```
ParOldGen total 481280K, used 278814K [0x......)
 object space 481280K, 57% used [0x......)
```

- ParOldGen， 老年代总计 total 481280K， 使用量 278814K
    - 其中 object space 占用了 481280K， 其中 57% used

```
Metaspace used 3449K, capacity 4494K, committed 4864K, reserved 1056768K
 class space used 366K, capacity 386K, committed 512K, reserved 1048576K
```

- Metaspace， 元数据区总计使用了 3449K， 容量是 4494K， JVM保证可用的大小是 4864K， 保留空间1GB左右
    - 其中 class space 使用了 366K， capacity 386K

指定GC日志文件

我们在前面的基础上， 加上启动参数 `-Xloggc:gc.demo.log`， 再次执行。

```
# 请注意命令行启动时没有换行， 此处是手工排版
java -Xloggc:gc.demo.log -XX:+PrintGCDetails
 demo.jvm0204.GCLogAnalysis
```

> **提示**: 从JDK8开始，支持使用 `%p`， `%t` 等占位符来指定GC输出文件。分别表示进程pid和启动时间戳。
> 例如: `-Xloggc:gc.%p.log`; `-Xloggc:gc-%t.log`;
> 在某些情况下，将每次JVM执行的GC日志输出到不同的文件可以方便排查问题。
> 如果业务访问量大，导致GC日志文件太大，可以开启GC日志轮换，分割成多个文件，可以参考: https://blog.gceasy.io/2016/11/15/rotating-gc-log-files 。

执行后在命令行输出的结果如下:

```
正在执行...
执行结束!共生成对象次数:1327
```

GC 日志哪去了？查看当前工作目录， 可以发现多了一个文件 `gc.demo.log` 。 如果是IDE开发环境，`gc.demo.log` 文件可能在项目的根目录下。
当然， 我们也可以指定GC日志文件存放的绝对路径，比如 `-Xloggc:/var/log/gc.demo.log` 等形式。

`gc.demo.log` 文件的内容如下：

```
Java HotSpot(TM) 64-Bit Server VM (25.162-b12) ......
Memory: 4k page,  physical 16777216k(1519448k free)

/proc/meminfo:

CommandLine flags:
  -XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296
  -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
  -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
  -XX:+UseParallelGC
0.310: [GC (Allocation Failure)
  [PSYoungGen: 61807K->10732K(76288K)]
  61807K->22061K(251392K), 0.0094195 secs]
  [Times: user=0.02 sys=0.02, real=0.01 secs]
0.979: [Full GC (Ergonomics)
  [PSYoungGen: 89055K->0K(572928K)]
  [ParOldGen: 280799K->254491K(434176K)]
  369855K->254491K(1007104K),
  [Metaspace: 3445K->3445K(1056768K)],
  0.0362652 secs]
  [Times: user=0.20 sys=0.01, real=0.03 secs]
...... 此处省略部分内容
Heap
...... 堆内存信息格式请参考前面的日志
```

我们可以发现， 加上 `-Xloggc:` 参数之后， GC日志信息输出到日志文件中。
文件里最前面是JVM相关信息， 比如内存页面大小， 物理内存大小， 剩余内存等信息。
然后是 `CommandLine flags` 这部分内容。 在分析GC日志文件时，命令行参数也是一项重要的参考。因为可能你拿到了日志文件，却不知道线上的配置，日志文件中打印了这个信息，能有效减少分析排查时间。
指定 `-Xloggc:` 参数，自动加上了 `-XX:+PrintGCTimeStamps` 配置。观察GC日志文件可以看到，每一行前面多了一个时间戳（如 `0.310:` ），表示JVM启动后经过的时间（单位秒）。

> 细心的同学还可以发现，JDK8默认使用的垃圾收集器参数: `-XX:+UseParallelGC` 。

打印GC事件发生的日期和时间

我们在前面的基础上，加上启动参数 `-XX:+PrintGCDateStamps` ，再次执行。

```
java -Xloggc:gc.demo.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps demo.jvm0204.GCLog
```

执行完成后，GC日志文件中的内容摘录如下：

```
...... 省略多行
CommandLine flags:
  -XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296
  -XX:+PrintGC -XX:+PrintGCDateStamps
  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
  -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
  -XX:+UseParallelGC
2019-12-15T15:09:59.235-0800: 0.296:
  [GC (Allocation Failure)
    [PSYoungGen: 63844K->10323K(76288K)]
    63844K->20481K(251392K),
    0.0087896 secs]
  [Times: user=0.02 sys=0.02, real=0.01 secs]
2019-12-15T15:09:59.889-0800: 0.951:
  [Full GC (Ergonomics)
    [PSYoungGen: 81402K->0K(577536K)]
    [ParOldGen: 270176K->261230K(445952K)]
    351579K->261230K(1023488K),
    [Metaspace: 3445K->3445K(1056768K)],
    0.0369622 secs]
  [Times: user=0.19 sys=0.00, real=0.04 secs]
Heap
  .......省略内容参考前面的格式
```

可以看到，加上 `-XX:+PrintGCDateStamps` 参数之后，GC日志每一行前面，都打印了GC发生时的具体时间。如 `2019-12-15T15:09:59.235-0800`，表示的是：`东8区时间2019年12月15日15:09:59秒.235毫秒`。

指定堆内存的大小

从前面的示例中可以看到GC日志文件中输出的 `CommandLine flags` 信息。
即使我们没有指定堆内存，JVM在启动时也会自动算出一个默认值出来。 例如：`-XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296` 是笔者机器上的默认值， 等价于 `-Xms256m -Xmx4g` 配置。

我们现在继续增加参数，这次加上启动参数 `-Xms512m -Xmx512m`， 再次执行。

```
```

```
java -Xms512m -Xmx512m -Xloggc:gc.demo.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps
```

此时输出的GC日志文件内容摘录如下：

```
.....
CommandLine flags:
  -XX:InitialHeapSize=536870912 -XX:MaxHeapSize=536870912
  -XX:+PrintGC -XX:+PrintGCDateStamps
  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
  -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
  -XX:+UseParallelGC
2019-12-15T15:15:09.677-0800: 0.358:
  [GC (Allocation Failure)
    [PSYoungGen: 129204K->21481K(153088K)]
    129204K->37020K(502784K), 0.0121865 secs]
  [Times: user=0.03 sys=0.03, real=0.01 secs]
2019-12-15T15:15:10.058-0800: 0.739:
  [Full GC (Ergonomics)
    [PSYoungGen: 20742K->0K(116736K)]
    [ParOldGen: 304175K->247922K(349696K)]
    324918K->247922K(466432K),
    [Metaspace: 3444K->3444K(1056768K)],
    0.0319225 secs]
  [Times: user=0.18 sys=0.01, real=0.04 secs]
```

此时堆内存的初始值和最大值都是512MB。具体的参数可根据实际需要配置，我们为了演示，使用了一个较小的堆内存配置。

指定垃圾收集器

一般来说，使用JDK8时我们可以使用以下几种垃圾收集器：

```
-XX:+UseSerialGC
-XX:+UseParallelGC
-XX:+UseParallelGC -XX:+UseParallelOldGC
-XX:+UseConcMarkSweepGC
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
-XX:+UseG1GC
```

它们都是什么意思呢，我们再简单回顾一下：

- 使用串行垃圾收集器： `-XX:+UseSerialGC`

- 使用并行垃圾收集器： `-XX:+UseParallelGC` 和 `-XX:+UseParallelGC -XX:+UseParallelOldGC` 是等价的, 可以通过GC日志文件中的flags看出来。
- 使用CMS垃圾收集器： `-XX:+UseConcMarkSweepGC` 和 `-XX:+UseParNewGC -XX:+UseConcMarkSweepGC` 是等价的。 但如果只指定 `-XX:+UseParNewGC` 参数则老年代GC会使用SerialGC。使用CMS时，命令行参数中会自动计算出年轻代、老年代的初始值和最大值，以及最大晋升阈值等信息（例如 `-XX:MaxNewSize=178958336 -XX:NewSize=178958336 -XX:OldSize=357912576` ）。
- 使用 G1垃圾收集器： `-XX:+UseG1GC` 。原则上不能指定G1垃圾收集器的年轻代大小，否则不仅是画蛇添足，更是自废武功了。因为G1的回收方式是小批量划定区块（region）进行，可能一次普通GC中既有年轻代又有老年代，可能某个区块一会是老年代，一会又变成年轻代了。

> 如果使用不支持的GC组合，会怎么样呢？答案是会启动失败，报fatal错误，有兴趣的同学可以试一下。

下一节课程会依次演示各种垃圾收集器的使用，并采集和分析他们产生的日志。它们的格式差距并不大，学会分析一种GC日志之后，就可以举一反三，对于其他类型的GC日志，基本上也能看懂各项信息的大概意思。

其他参数

JVM里还有一些GC日志相关的参数， 例如：

- `-XX:+PrintGCApplicationStoppedTime` 可以输出每次GC的持续时间和程序暂停时间；
- `-XX:+PrintReferenceGC` 输出GC清理了多少引用类型。

这里就不在累述，想了解配置详情的，可以回头复习前面的章节。

> **说明**：大部分情况下，配置GC参数并不是越多越好。原则上只配置最重要的几个参数即可，其他的都保持默认值，除非你对系统的业务特征有了深入的分析和了解，才需要进行某些细微参数的调整。 毕竟，古语有云：“过早优化是万恶之源”。

三、GC事件的类型简介

一般来说，垃圾收集事件(Garbage Collection events)可以分为三种类型：

- `Minor GC(小型GC)`
- `Major GC(大型GC)`
- `Full GC(完全GC)`

虽然 `Minor GC` ， `Major GC` 和 `Full GC` 这几个词汇到处都在用，但官方并没有给出标准的定义。这些术语出现在官方的各种分析工具和垃圾收集日志中，并不是很统一。官方的文档和工具之间也常常混淆，这些混淆甚至根植于标准的JVM工具中。

> MinorGC 称为 `小型GC` ，还是 `次要GC` 更合理呢?
> **辨析**：在大部分情况下， 发生在年轻代的 `Minor GC` 次数更多， 有些文章将次数更多的GC称为次要GC 明显是不太合理的。
> 在这里，我们将 `Minor GC` 翻译为 `小型GC` ，而不是 次要GC；

> 将 `Major GC` 翻译为 `大型GC` 而不是 ~~主要GC~~；
> `Full GC` 翻译为： **完全GC**；有时候也直接称为 Full GC。
> 其实这也是因为专有名词在中英文翻译的时候，可能会有多个英语词汇对应一个中文词语，也会有一个英文词汇对应多个中文词语，要看具体情况而定。
> 比如一个类似的情况： `Major Version` 和 `Minor Version`，这两个名词一般翻译为 `主要版本` 和 `次要版本`。这当然没问题，大家都能理解，一看就知道什么意思。甚至直接翻译为 `大版本号` 和 `小版本号` 也是能讲得通的。

本节简单介绍了这几种事件类型及其区别，下面我们来看看这些事件类型的具体细节。

Minor GC（小型GC）

收集年轻代内存的GC事件称为 `Minor GC`。关于 `Minor GC` 事件， 我们需要了解一些相关的内容：

1. 当JVM无法为新对象分配内存空间时就会触发 `Minor GC`（一般就是 Eden 区用满了）。 如果对象的分配速率很快， 那么 `Minor GC` 的次数也就会很多，频率也就会很快。
2. `Minor GC` 事件不处理老年代， 所以会把所有从老年代指向年轻代的引用都当做 `GC Root`。从年轻代指向老年代的引用则在标记阶段被忽略。
3. 与我们一般的认知相反， `Minor GC` 每次都会引起STW停顿（stop-the-world）， 挂起所有的应用线程。 对大部分应用程序来说， `Minor GC` 的暂停时间可以忽略不计，因为 Eden 区里面的对象大部分都是垃圾，也不怎么复制到存活区/老年代。 但如果不符合这种情况，那么很多新创建的对象就不能被GC清理， `Minor GC` 的停顿时间就会增大，就会产生比较明显的GC性能影响。

> 简单定义： `Minor GC` 清理的是年轻代，又或者说 `Minor GC` 就是 `年轻代GC`（`Young GC`，简称YGC）。

Major GC vs. Full GC

值得一提的是，这几个术语都没有正式的定义--无论是在JVM规范中还是在GC论文中。

我们知道，除了 `Minor GC` 外，另外两种GC事件则是：

- `Major GC(大型GC)`：清理老年代空间（Old Space）的GC事件。
- `Full GC(完全GC)`：清理整个堆内存空间的GC事件，包括年轻代空间和老年代空间。

其实 `Major GC` 和 `Full GC` 有时候并不能很好地区分。更复杂的情况是， 很多 `Major GC` 是由 `Minor GC` 触发的，所以很多情况下这两者是不可分离的。
另外，像G1这种垃圾收集算法，是每次找一小部分区域来进行清理，这部分区域中可能有一部分是年轻代，另一部分区域属于老年代。

所以我们不要太纠结具体是叫 `Major GC` 呢还是叫 `Full GC`，它们一般都会造成单次较长时间的STW暂停。所以我们需要关注的是：某次GC事件，是暂停了所有线程、进而对系统造成了性能影响呢，还是与其他业务线程并发执行、暂停时间几乎可以忽略不计。

本节内容到此就结束了，下一节我们通过实例来分析各种GC算法产生的日志。