

1、GC日志分析

查看GC日志参数设置

通过阅读GC日志，我们可以了解Java虚拟机内存分配与回收策略。

内存分配与垃圾回收的参数列表

<code>-XX:+PrintGC</code>	输出GC日志。类似: <code>-verbose:gc</code>
<code>-XX:+PrintGCDetails</code>	输出GC的详细日志
<code>-XX:+PrintGCTimeStamps</code>	输出GC的时间戳（以基准时间的形式）
<code>-XX:+PrintGCDateStamps</code>	输出GC的时间戳（以日期的形式，如2013-05-04T21:53:59.234+0800）
<code>-XX:+PrintHeapAtGC</code>	在进行GC的前后打印出堆的信息
<code>-Xloggc:../logs/gc.log</code>	日志文件的输出路径

- 打开GC日志:

```
-verbose:gc
```

- 这个只会显示总的GC堆的变化，如下:

```
[GC (Allocation Failure) 80832K->19298K(227840K), 0.0084018 secs]
[GC (Metadata GC Threshold) 109499K->21465K(228352K), 0.0184066 secs]
[Full GC (Metadata GC Threshold) 21465K->16716K(201728K), 0.0619261 secs]
```

- 参数解析:

GC、Full GC: GC的类型，GC只在新生代上进行，Full GC包括永生代，新生代，老年代。

Allocation Failure: GC发生的原因。

80832K->19298K: 堆在GC前的大小和GC后的大小。

228840k: 现在的堆大小。

0.0084018 secs: GC持续的时间。

- 打开GC日志:

```
-verbose:gc -XX:+PrintGCDetails
```

- 输入信息如下:

```
[GC (Allocation Failure) [PSYoungGen: 70640K->10116K(141312K)] 80541K->20017K(227328K), 0.0172573
secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
[GC (Metadata GC Threshold) [PSYoungGen: 98859K->8154K(142336K)] 108760K->21261K(228352K),
0.0151573 secs] [Times: user=0.00 sys=0.01, real=0.02 secs]
[Full GC (Metadata GC Threshold) [PSYoungGen: 8154K->0K(142336K)] [ParOldGen: 13107K-
>16809K(62464K)] 21261K->16809K(204800K), [Metaspace: 20599K->20599K(1067008K)], 0.0639732 secs]
[Times: user=0.14 sys=0.00, real=0.06 secs]
```

- 参数解析:

GC, Full GC: 同样是GC的类型

Allocation Failure: GC原因

PSYoungGen: 使用了Parallel Scavenge并行垃圾收集器的新生代GC前后大小的变化

ParOldGen: 使用了Parallel old并行垃圾收集器的老年代GC前后大小的变化

Metaspace: 元数据区GC前后大小的变化, JDK1.8中引入了元数据区以替代永久代

xxx secs: 指GC花费的时间

Times: user: 指的是垃圾收集器花费的所有CPU时间, sys: 花费在等待系统调用或系统事件的时间, real: GC从开始到结束的时间, 包括其他进程占用时间片的实际时间。

日志补充说明:

- **Allocation Failure**

表明本次引起GC的原因是因为在年轻代中没有足够的空间能够存储新的数据了。

- **[PSYoungGen: 5986K->696K(8704K)] [5986K->704K(9216K)]**

中括号内: GC回收前年轻代大小, 回收后大小, (年轻代总大小)

括号外: GC回收前年轻代和老年代大小, 回收后大小, (年轻代和老年代总大小)

- **user**代表用户态回收耗时, **sys**内核态回收耗时, **rea**实际耗时。由于多核的原因, 时间总和可能会超过**real**时间

Heap (堆)

PSYoungGen(Parallel Scavenge收集器新生代)total 9216K, used 6234K

[0x00000000ff600000, 0x0000000010000000, 0x0000000010000000)

eden space(堆中的Eden区默认占比是8) 8192K, 76% used

[0x00000000ff600000, 0x00000000ffc16b08, 0x00000000ffe00000)

from space(堆中的Survivor, 这里是From Survivor区默认占比是1) 1024K, 0% used

[0x00000000fff00000, 0x00000000fff00000, 0x0000000010000000)

to space(堆中的Survivor, 这里是to Survivor区默认占比是1, 需要先了解一下堆的分配策略)

1024K, 0% used [0x00000000ffe00000, 0x00000000ffe00000, 0x00000000fff00000)

ParOldGen(老年代总大小和使用大小)total 10240K, used 7001K [0x00000000fec00000,

0x00000000ff600000, 0x00000000ff600000)

object space(显示个使用百分比) 10240K, 68% used

[0x00000000fec00000, 0x00000000ff2d6630, 0x00000000ff600000)

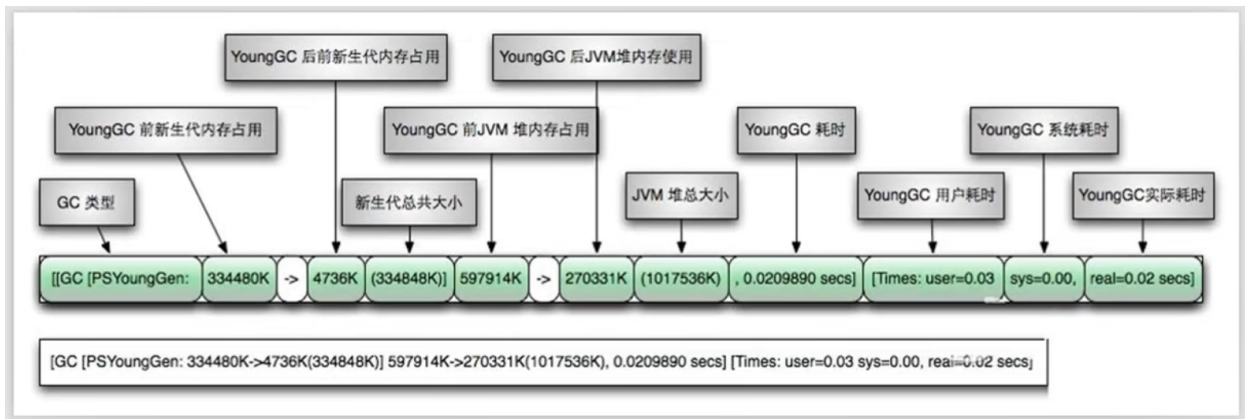
PSPermGen(永久代总大小和使用大小)total 21504K, used 4949K [0x00000000f9a00000,

0x00000000faf00000, 0x00000000fec00000)

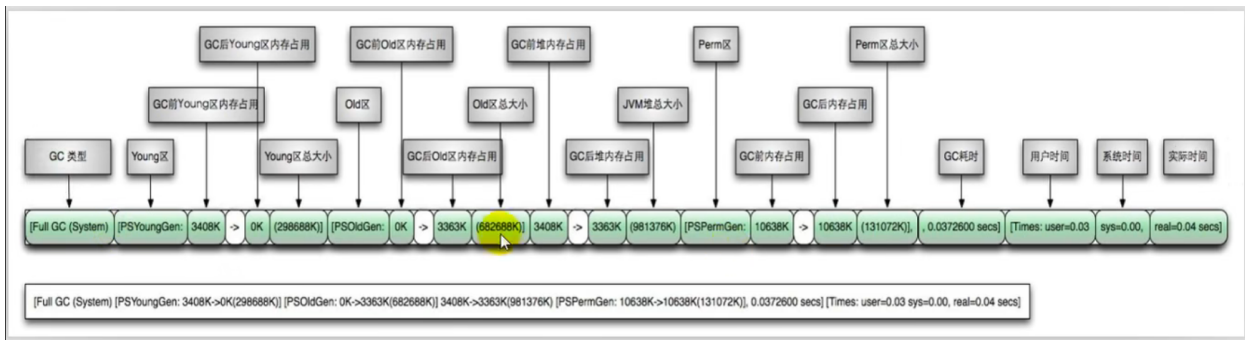
object space(显示个使用百分比, 自己能算出来) 21504K, 23% used

[0x00000000f9a00000, 0x00000000f9ed55e0, 0x00000000faf00000)

年轻代GC日志



Full GC日志



查看堆空间回收日志

6 /**

7 * VM Options: -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags

8 *

9 * @author 卡哇米了

10 * @date 2022-08-19 10:37

11 */

① 设置的新生代的大小为10m也就是10240K，这里只显示9216K，是因为堆区有Eden、Survivor0、Survivor1，但是Eden同时只和一个Survivor区工作，因为使用的是复制算法，总有一个Survivor区是空的

```

C:\Program Files\Java\jdk1.8.0_241\bin\java.exe ...
-XX:InitialHeapSize=20971520 -XX:MaxHeapSize=20971520 -XX:MaxNewSize=10485760 -XX:NewSize=10485760 -XX:+PrintCommandLineFlags
[GC (Allocation Failure) [PSYoungGen: 7248K->1016K (9216K)] 7248K->5345K (19456K), 0.0042306 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]
Heap
 PSYoungGen    total 9216K  used 5436K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
  eden space 8192K, 53% used [0x00000000ff600000, 0x00000000ffa51118, 0x00000000ffe00000)
 from space 1024K, 99% used [0x00000000ffe00000, 0x00000000ffe010, 0x00000000fff00000)
 to space 1024K, 0% used [0x00000000ff000000, 0x00000000ff000000, 0x0000000100000000)
 ParOldGen     total 10240K, used 4329K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
 object space 10240K, 42% used [0x00000000fec00000, 0x00000000ff03a450, 0x00000000ff600000)
 Metaspace     used 3290K, capacity 4496K, committed 4864K, reserved 1056768K
 class space   used 354K, capacity 388K, committed 512K, reserved 1048576K
  
```

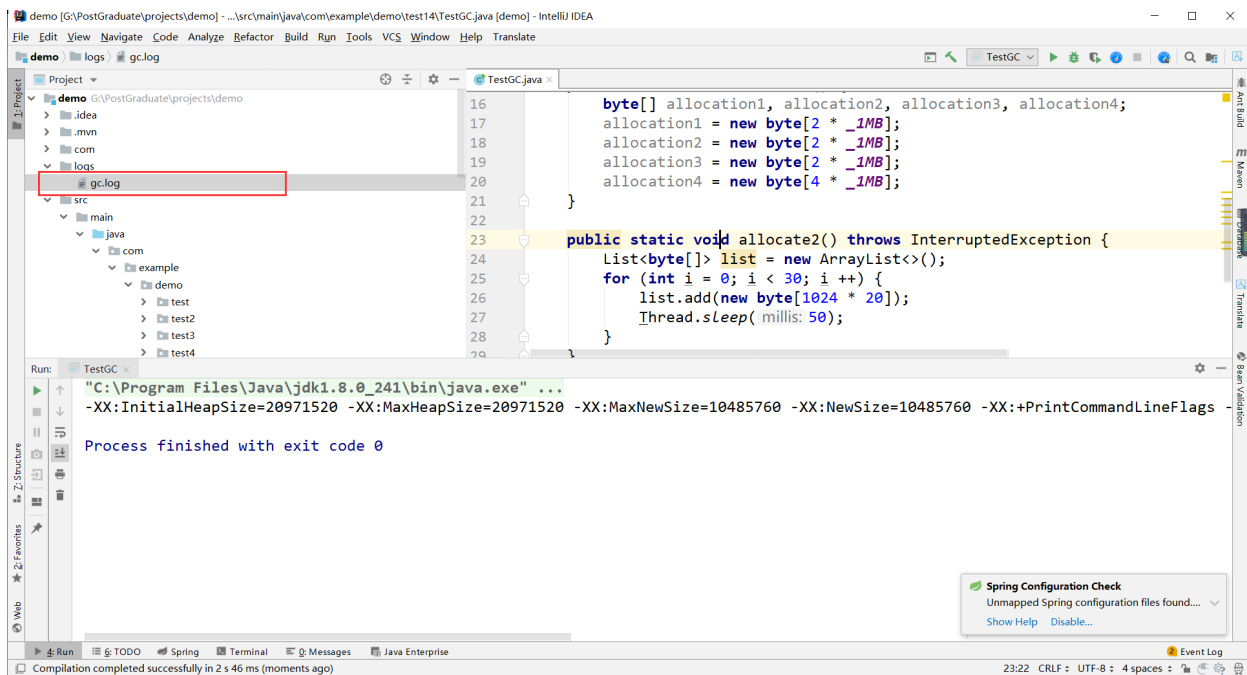
GC日志分析工具

可以用一些工具去分析这些gc日志。

常用的日志分析工具有：**GCviewer**、GCEasy、GCHisto、GCLogViewer、Hpjmeter、garbagecat等。

首先编写代码，生成日志文件

```
1 package com.example.demo.test14;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * VM Options: -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+PrintComm
andLineFlags -Xloggc:./logs/gc.log
8  *
9  * @author 千祎来了
10  * @date 2022-08-19 10:37
11  */
12 public class TestGC {
13     private static final int _1MB = 1024 * 1024;
14
15     public static void allocate() {
16         byte[] allocation1, allocation2, allocation3, allocation4;
17         allocation1 = new byte[2 * _1MB];
18         allocation2 = new byte[2 * _1MB];
19         allocation3 = new byte[2 * _1MB];
20         allocation4 = new byte[4 * _1MB];
21     }
22
23     public static void allocate2() throws InterruptedException {
24         List<byte[]> list = new ArrayList<>();
25         for (int i = 0; i < 30; i++) {
26             list.add(new byte[1024 * 20]);
27             Thread.sleep(50);
28         }
29     }
30
31     public static void main(String[] args) throws InterruptedException {
32         allocate2();
33     }
34 }
```



进入官网

选择日志文件即可分析

<https://gceasy.io/>

