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

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

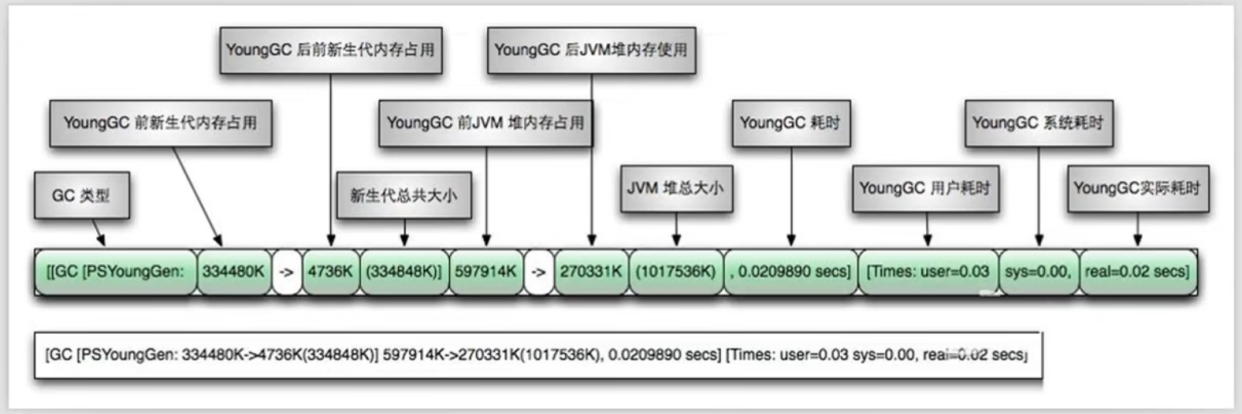
```
GC, Full FC: 同样是GC的类型
Allocation Failure: GC原因
PSYoungGen: 使用了Parallel Scavenge并行垃圾收集器的新生代GC前后大小的变化
ParOldGen: 使用了Parallel Old并行垃圾收集器的老年代GC前后大小的变化
Metaspace: 元数据区GC前后大小的变化， JDK1.8中引入了元数据区以替代永久代
xxx secs: 指GC花费的时间
Times: user: 指的是垃圾收集器花费的所有CPU时间， sys: 花费在等待系统调用或系统事件的时间， real:
GC从开始到结束的时间，包括其他进程占用时间片的实际时间。
```

日志补充说明：

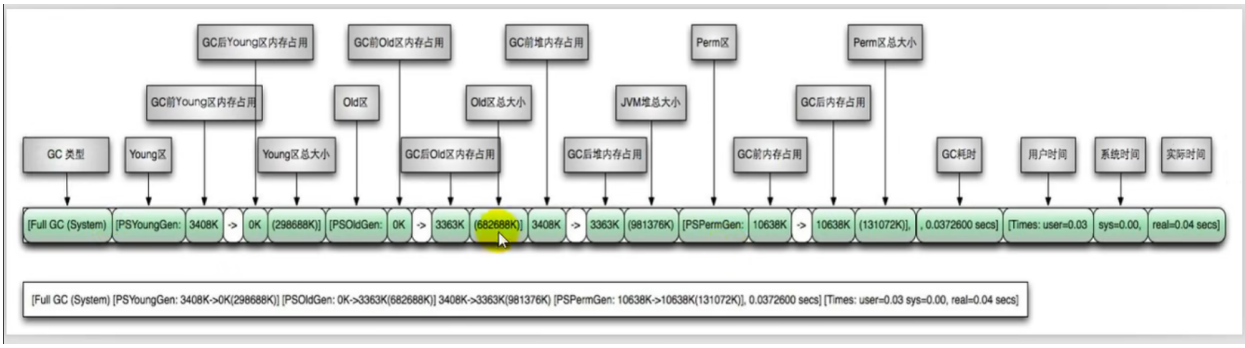
- **Allocation Failure**
  表明本次引起GC的原因是因为在年轻代中没有足够的空间能够存储新的数据了。
- **[PSYoungGen: 5986K->696K(8704K)] 5986K->704K(9216K)**
  中括号内：GC回收前年轻代大小，回收后大小，（年轻代总大小）
  括号外：GC回收前年轻代和老年代大小，回收后大小，（年轻代和老年代总大小）
- **user代表用户态回收耗时，sys内核态回收耗时，rea实际耗时。由于多核的原因，时间总和可能会超过real时间**

```
Heap(堆)
PSYoungGen(Parallel Scavenge收集器新生代)total 9216K, used 6234K
[0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
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
[0x00000000fff00000,0x00000000fff00000,0x0000000100000000)
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)
```

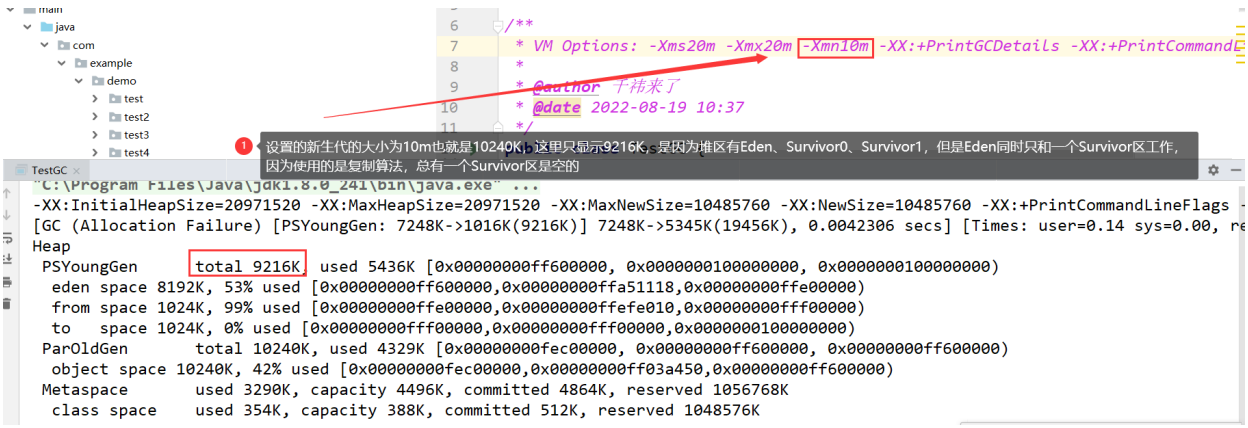
年轻代GC日志



Full GC日志



查看堆空间回收日志



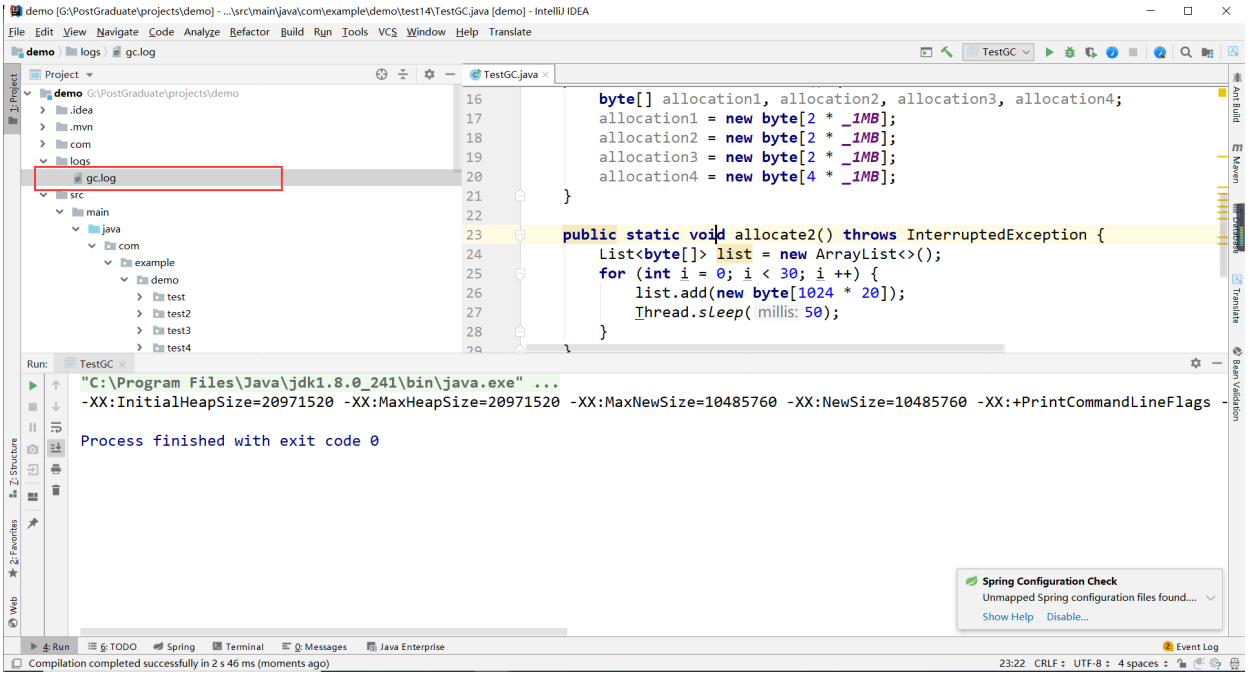
GC日志分析工具

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

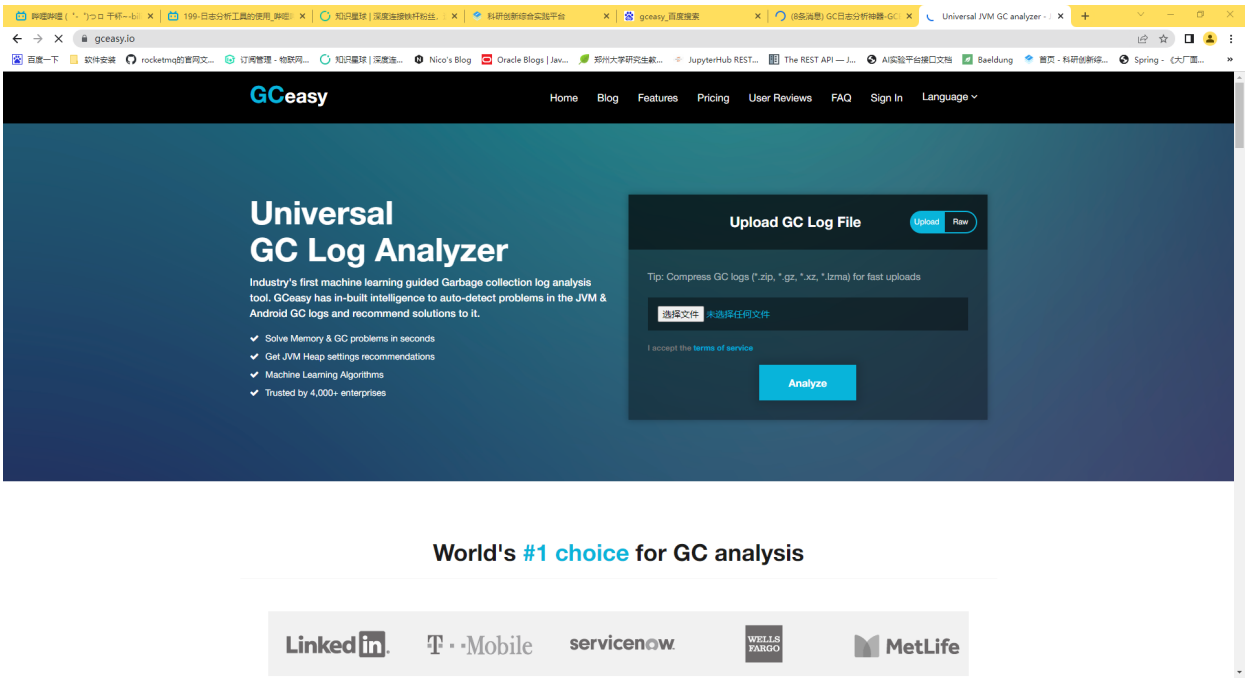
```
package com.example.demo.test14;

import java.util.ArrayList;
import java.util.List;

/**
 * VM Options: -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -Xloggc:./logs/gc.log
 *
 * @author 千祎来了
 * @date 2022-08-19 10:37
 */
public class TestGC {
 private static final int _1MB = 1024 * 1024;

 public static void allocate() {
 byte[] allocation1, allocation2, allocation3, allocation4;
 allocation1 = new byte[2 * _1MB];
 allocation2 = new byte[2 * _1MB];
 allocation3 = new byte[2 * _1MB];
 allocation4 = new byte[4 * _1MB];
 }

 public static void allocate2() throws InterruptedException {
 List<byte[]> list = new ArrayList<>();
 for (int i = 0; i < 30; i ++) {
 list.add(new byte[1024 * 20]);
 Thread.sleep(50);
 }
 }

 public static void main(String[] args) throws InterruptedException {
 allocate2();
 }
}
```



进入官网

选择日志文件即可分析

https://gceasy.io/



200