geek-01

1.（选做）自己写一个简单的 Hello.java，里面需要涉及基本类型，四则运行，if 和 for，然后自己分析一下对应的字节码，有问题群里讨论。

源文件:

```
public class ByteCodeAnalyDemo {
    int d = 300;
    final int f = 200;
    public static void main(String[] args) {
        int a = 100;
        int b = 0;
        for (int i = 1; i < a; i+=2) {
            if (b >= 50) {
                break;
            }
            b += i;
            b /= i;
            b *= i;
        }
        System.out.println(b);
    }
}
```

字节码文件解析:

```
admin@admindeMacBook-Pro-3 homework % javap -c -verbose
ByteCodeAnalyDemo.class
Classfile /...../ByteCodeAnalyDemo.class
  Last modified 2021-6-23; size 841 bytes
  MD5 checksum ba525c48acbcb2642e992354fd2e55f0
  Compiled from "ByteCodeAnalyDemo.java"
public class com.risesun.grass.geek.jvm.homework.ByteCodeAnalyDemo
  minor version: 0
    major version: 52
// java8编译
  flags: ACC_PUBLIC, ACC_SUPER
// public以及super修饰
```

```
Constant pool:
// 常量池
   #1 = Methodref          #7.#30         // java/lang/Object."<init>":()V
   #2 = Fieldref           #6.#31         //
com/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo.d:I
   #3 = Fieldref           #6.#32         //
com/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo.f:I
   #4 = Fieldref           #33.#34        //
java/lang/System.out:Ljava/io/PrintStream;
   #5 = Methodref          #35.#36        // java/io/PrintStream.println:
(I)V
   #6 = Class              #37            //
com/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo
   #7 = Class              #38            // java/lang/Object
   #8 = Utf8               d
   #9 = Utf8               I
  #10 = Utf8               f
  #11 = Utf8               ConstantValue
  #12 = Integer            200
// final int f = 200;作为常量存储在常量池
  #13 = Utf8               <init>
  #14 = Utf8               ()V
  #15 = Utf8               Code
  #16 = Utf8               LineNumberTable
  #17 = Utf8               LocalVariableTable
  #18 = Utf8               this
  #19 = Utf8
Lcom/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo;
  #20 = Utf8               main
  #21 = Utf8               ([Ljava/lang/String;)V
  #22 = Utf8               i
  #23 = Utf8               args
  #24 = Utf8               [Ljava/lang/String;
  #25 = Utf8               a
  #26 = Utf8               b
  #27 = Utf8               StackMapTable
  #28 = Utf8               SourceFile
  #29 = Utf8               ByteCodeAnalyDemo.java
  #30 = NameAndType        #13:#14        // "<init>":()V
  #31 = NameAndType        #8:#9          // d:I
  #32 = NameAndType        #10:#9         // f:I
  #33 = Class              #39            // java/lang/System
  #34 = NameAndType        #40:#41        // out:Ljava/io/PrintStream;
```

```
  #35 = Class              #42            // java/io/PrintStream
  #36 = NameAndType        #43:#44        // println:(I)V
  #37 = Utf8
com/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo
  #38 = Utf8               java/lang/Object
  #39 = Utf8               java/lang/System
  #40 = Utf8               out
  #41 = Utf8               Ljava/io/PrintStream;
  #42 = Utf8               java/io/PrintStream
  #43 = Utf8               println
  #44 = Utf8               (I)V
{
  int d;
    descriptor: I
// 定义了一个d, int值
    flags:

  final int f;
    descriptor: I
    flags: ACC_FINAL
    ConstantValue: int 200
// 定义了f = 200常量, final修饰

  public com.risesun.grass.geek.jvm.homework.ByteCodeAnalyDemo();
// 构造函数
    descriptor: ()V
// 返回值void
    flags: ACC_PUBLIC
// public修饰
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V                 // 执行object的init方法
         4: aload_0
         5: sipush        300
         8: putfield      #2                  // Field d:I
// 常量池中2号位置为d属性, d的值设置为300
        11: aload_0
        12: sipush        200
        15: putfield      #3                  // Field f:I
// 常量池中3号位置为f属性, f的值设置为200
        18: return
```

```
      LineNumberTable:
        line 8: 0
        line 9: 4
        line 10: 11
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      19     0  this
Lcom/risesun/grass/geek/jvm/homework/ByteCodeAnalyDemo;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=4, args_size=1
         0: bipush        100                  // 将100加载到栈中（代表a=100）
,  由于100数字过大，占用了1个字节的指令空间，所以后面的指令从第3个字节码开始。
         2: istore_1                           // 将栈中值为100的数放入到变量表1
号位置
         3: iconst_0                           // 将常量0加载到栈中（代表b=0）
         4: istore_2                           // 将栈中值为0的数放入到变量表2号
位置
         5: iconst_1                           // 将常量1加载到栈中（代表i=1）
         6: istore_3                           // 将栈中值为1的数放入到变量表3位
置
         7: iload_3                            // 将变量表中3号位置的数据加载到
栈中(代表int i = 1的值取出来放入栈中)
         8: iload_1                            // 将变量表中1号位置的数据加载到
栈中(代表取出a的值100)
         9: if_icmpge     39                   // 将变量表中的3号位置和1号位置中
数据进行对比(此时都在栈中)，即，对比1和100，如果i>=a(ge代表>=)，跳转到字节码39号位置，
即循环结束
        12: iload_2                            // 将变量表中的2号位置数据取出来
然后放入到栈中(2号位置值为0，即b的值)
        13: bipush        50                   // 将50加载到栈中
        15: if_icmplt     21                   // 对栈中的2个数据进行对比，如果
0<50，则跳转到编号21的位置，否则执行下一句跳出循环(lt代表<)
        18: goto          39                   // 跳出循环-break，跳转到编号39
的位置
        21: iload_2                            // 取出本地变量表中2号位置的数
据，值为0放入到栈中
        22: iload_3                            // 取出本地变量表中3号位置的数
据，值为1放入到栈中
        23: iadd                               // 进行求和操作
```

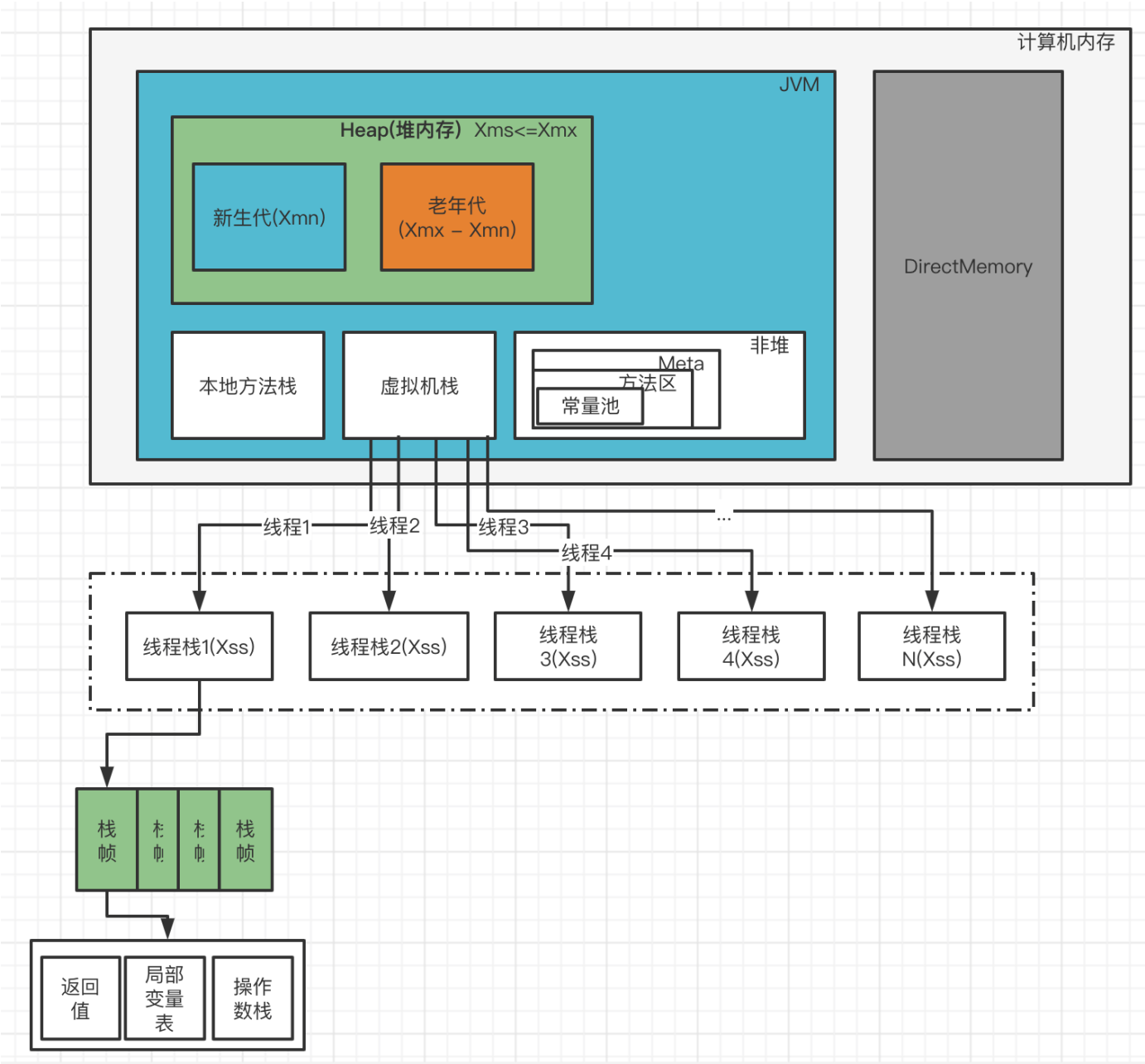
```
        24: istore_2                            // 将上面一步求和的数据值为1，放
入到变量表2号位置，即b = 1;
        25: iload_2                             // 将变量表2号位置的1拿出来放入到
栈中
        26: iload_3                             // 将变量表3号位置的1拿出来放入到
栈中
        27: idiv                                // 进行求商操作
        28: istore_2                            // 将上面一步求商的数据值为1，放
入到变量表2号位置，即b = 1;
        29: iload_2                             // 将变量表2号位置的1拿出来放入到
栈中
        30: iload_3                             // 将变量表3号位置的1拿出来放入到
栈中
        31: imul                                // 进行乘法操作
        32: istore_2                            // 将上面一步求积的数据值为1，放
入到变量表2号位置，即b = 1;
        33: iinc          3, 2                  // 将变量表中3号位置的值，自增数
值2，则3号位置的值 = 3;
        36: goto          7                     // 回到第7个字节码所在的位置，循
环执行
        39: getstatic     #2                    // Field
java/lang/System.out:Ljava/io/PrintStream;l
        42: iload_2                             // 取出本地变量表中2号位置的数
据，到栈中
        43: invokevirtual #3                    // Method
java/io/PrintStream.println:(I)V
        46: return
      LineNumberTable:                          // 行号表
        line 12: 0
        line 13: 3
        line 14: 5
        line 15: 12
        line 16: 18
        line 18: 21
        line 19: 25
        line 20: 29
        line 14: 33
        line 22: 39
        line 23: 46
      LocalVariableTable:                       // 本地变量表，存放i,a,b
        Start  Length  Slot  Name   Signature
            7      32     3     i   I
            0      47     0  args   [Ljava/lang/String;
```

```
            3      44     1     a   I
            5      42     2     b   I
      StackMapTable: number_of_entries = 3
        frame_type = 254 /* append */
          offset_delta = 7
          locals = [ int, int, int ]
        frame_type = 13 /* same */
        frame_type = 250 /* chop */
          offset_delta = 17
}
SourceFile: "ByteCodeAnalyDemo.java"
```

2.（必做）自定义一个 Classloader，加载一个 Hello.xlass 文件，执行 hello 方法，此文件内容是一个 Hello.class 文件所有字节（x=255-x）处理后的文件。文件群里提供。

```
public class HelloDecodeClass extends ClassLoader {
    public static void main(String[] args) throws Exception {
        InputStream inputStream =
HelloDecodeClass.class.getClassLoader().getResourceAsStream("Hello.xlass")
;
        byte[] bytes = new byte[1000];
        int index = -1;
        while (true) {
            int bytecode = inputStream.read();
            if (bytecode == -1) {
                break;
            }
            index++;
            bytecode = 255 - bytecode;
            bytes[index] = (byte) bytecode;
        }
        byte[] result = new byte[++index];
        System.arraycopy(bytes, 0, result, 0, index);
        Class<?> clazz = new HelloDecodeClass().defineClass("Hello",
result, 0, result.length);
        Object obj = clazz.newInstance();
        Method helloMethod = obj.getClass().getDeclaredMethod("hello");
        helloMethod.invoke(obj);
    }
}
```

3.（必做）画一张图，展示 Xmx、Xms、Xmn、Meta、DirectMemory、Xss 这些内存参数的关系。



4. (选做)检查一下自己维护的业务系统的 JVM 参数配置，用 jstat 和 jstack、jmap 查看一下详情，并且自己独立分析一下大概情况，思考有没有不合理的地方，如何改进。

以一个后台管理系统为例做分析:

1. 通过jps查询其进程ID，由于做了资源隔离，进程id都是1.

2. 查询jvm内存情况

```
ro[redacted]:/# jcmd 1 GC.heap_info
1:
 def new generation   total 1120704K, used 958668K [0x000000070c800000, 0x0000000758800000, 0x0000000758800000)
  eden space 996224K,  92% used [0x000000070c800000, 0x000000074501bf10, 0x00000007494e0000)
  from space 124480K,  26% used [0x0000000750e70000, 0x0000000752e873d8, 0x0000000758800000)
  to   space 124480K,   0% used [0x00000007494e0000, 0x00000007494e0000, 0x0000000750e70000)
 tenured generation   total 2228224K, used 789374K [0x0000000758800000, 0x00000007e0800000, 0x00000007e0800000)
   the space 2228224K,  35% used [0x0000000758800000, 0x0000000788adf838, 0x0000000788adfa00, 0x00000007e0800000)
 Metaspace       used 194764K, capacity 200688K, committed 200832K, reserved 1228800K
  class space    used 19250K, capacity 20517K, committed 20608K, reserved 1048576K
```

由于线上系统无法使用jmap，所以先使用jcmd代替。
由以上数据可知，新生代一共1094MB，大概是1g, 而内部eden区972MB,存活区各自121MB。而老年代一共是2176MB,大概是2g左右，目前使用35%。推测堆大小是3GB。Metaspace区域195MB。

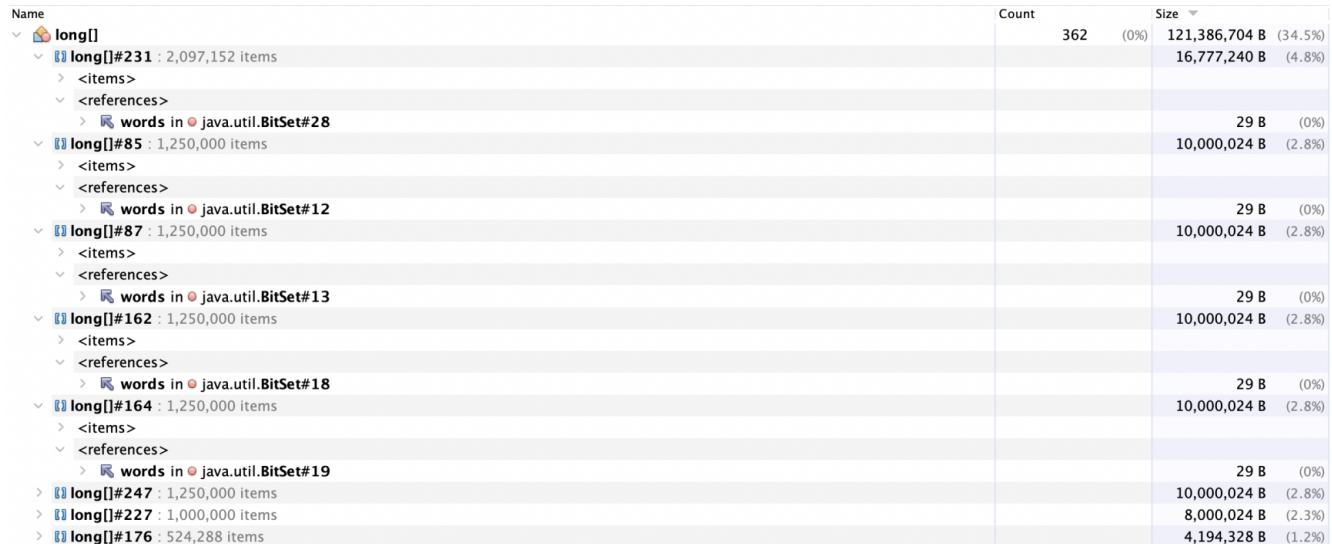
3. 查询启动参数
-Xmx3392M -Xms3392M -Xmn1216M -XX:MaxMetaspaceSize=512M
-XX:MetaspaceSize=512M -XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/home/www/tomcat/apache-tomcat-7.0.93/logs/dump.hprof

4. 分析其gc情况

```
root@tf8-web-rose-fc8cffb4-rp2rn:/# jstat -gcutil 1 1000 1000
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00  26.11  12.92  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  12.92  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  12.92  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  13.01  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  13.01  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  13.01  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  13.01  35.47  96.93  93.44    197   16.383     0    0.000   16.383
  0.00  26.11  13.02  35.47  96.93  93.44    197   16.383     0    0.000   16.383
```

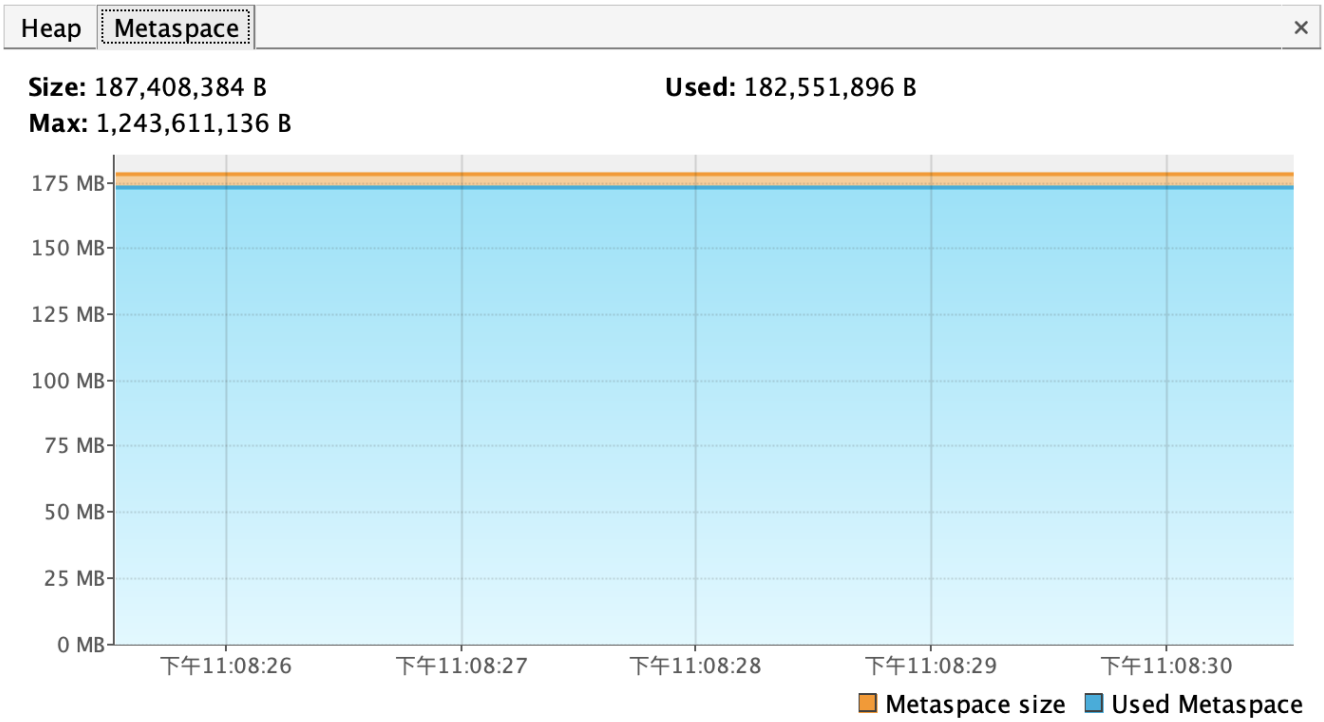
根据指令返回的结果分析，old区域稳定在35%的使用率，eden区增长也较为缓慢，年轻代GC次数为197次，而full gc为0次。

5. 通过jvisualvm的heap dump功能生成内存快照，发现long[]数组占据很多空间。根据GC roots功能查询其引用的来源:



6. 原来是系统内部使用的一个布隆过滤器生成的long[]数组，将近使用了115MB, 通过分析代码发现公司内部的布隆过滤器使用了jdk的BitSet实现，而BitSet内部生成了一个很大的long数组，但是因为布隆过滤器一般作为static静态变量使用，所以一直不能被回收。

7. 通过元数据区的展现，发现Metaspace的区域都快满了。



但暂时没有发现，如何看Metaspace区域内容的命令。

8. 通过类的直方图发现内存的占用如下(只贴了前面一部分);

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:           412      121385520  [J
   2:        491765       53076704  [C
   3:        167116       14706208  java.lang.reflect.Method
   4:        483019       11592456  java.lang.String
   5:        477493        7639888
java.util.concurrent.atomic.AtomicReference
   6:        178288        5705216
java.util.concurrent.ConcurrentHashMap$Node
```

暂时不清楚[J和[C是什么意思。 但是目前来看，[J应该是一个大对象，因为才412个实例，就占据了将近115MB，等下，这个值好像跟第6步发现的long[]数组是接近的。

暂时发现的问题：

1. 公司内部使用的布隆过滤器内部产生了很多long[]数据，占据很多内存，后面可以将布隆过滤器进行改造内部实现，或者是禁止使用static修饰导致其无法被回收。像一些job系统，完全可以将过滤器定义在方法内部，方法执行结束，这部分占据的内存就可以被回收了。
2. Metaspace空间占满了，除了扩容之外，暂时不得知内部有哪些数据。

5.（选做）本机使用 G1 GC 启动一个程序，仿照课上案例分析一下 JVM 情况。

> 本机mac系统无法使用sudo jhsdb jmap --heap --pid [pid]命令(即使更换了很多个jdk版本)，将war包发送到远程linux服务器，发现使用了openJdk，该命令使用也受限了。生产环境下面，由于docker的原因，命令的执行也受到阻碍，这道题先留着，后面再补上。

总结

经过这次作业，发现自己对一些知识完全停留在了理论上，当要开始分析一个系统GC的情况，竟然从脑子里面提取不出来什么信息，完全没有思路。后面还要加强实战这块。