JDK17的GC调优策略

-- 楼兰

GC垃圾回收器是JVM中最标志性的一个功能特性。而GC的性能极大程度决定了整个JAVA程序执行的性能。因此，对整个JVM调优或许难度太大，但是对GC进行调优，是每个JAVA程序员都应该掌握的技能。

一、JVM有哪些参数可以调?

我们先来回顾一下JDK17中有哪些参数可以调。

关于 JVM 的参数，JVM 提供了三类参数。

一类是标准参数。以-开头，所有 HotSpot 都支持。例如java -version。这类参数可以使用java -help 或者java -? 全部打印出来

以下是一些常用的标准参数：

- --list_modules : 查看当前JAVA进程中的模块
- --show-module-resolution: 查看当前JAVA进程中各个模块的依赖关系
- -verbose:class : 显示类加载的信息
- -verbose:gc : 显示GC事件

二类是非标准参数。以-X 开头，是特定 HotSpot版本支持的指令。例如java -Xms200M -Xmx200M。这类指令可以用java -X 全部打印出来。这一类参数一般都还是比较稳定，除非有重大的版本升级，一般不会有太大的变化。

例如： -Xint 表示当前JAVA进程采用解释执行。-Xcomp表示当前JAVA进程采用编译执行。-Xmixed则表示采用两种执行引擎混合的方式执行。

-Xbatch 禁用后台编译。默认情况下，JVM将该方法作为后台任务进行编译，在解释器模式下运行该方法，直到后台编译完成。-Xbatch标志禁用后台编译，以便所有方法的编译都作为前台任务进行，直到完成。此选项等效于-XX:-BackgroundCompilation。

最后一类，不稳定参数。这也是 JVM调优的噩梦。以-XX 开头，这些参数是跟特定HotSpot版本对应的，很有可能换个版本就没有了。JDK中的以下几个指令可以帮助开发者了解 这一类不稳定参数。

```
java -XX:+PrintFlagsFinal:所有最终生效的不稳定指令。
java -XX:+PrintFlagsInitial:默认的不稳定指令
java -XX:+PrintCommandLineFlags:当前命令的不稳定指令
```

> 小题目：JDK17默认用的是哪种垃圾回收器?

> 留个小的课外题：观察下你在IDEA里启动一个项目，具体的启动指令是什么样的？自行尝试下不用IDEA，直接用java指令启动下自己的一个小项目。

接下来，我们可以通过在java指令后加入相关的参数进行定制。例如对于数字型的参数，可以直接在java指令后指定。而对于boolean型的参数，可以通过在参数前面添加一个加号表示设置为true，添加一个减号则表示设置为false。例如：

```
java -XX:ActiveProcessorCount=1 -XX:+AggressiveHeap -XX:+PrintFlagsFinal -version
```

二、从RocketMQ学习常用GC调优三部曲

在真实项目当中，要如何合理的定制GC相关的运行参数呢?

关于这个问题，其实没有正确答案。要是有绝对正确的答案，那么JDK早就将这些参数的默认值调整为这个正确答案了，我们也就不需要学习了。 而我们想要学习合理定制GC的参数，唯一的方法也就是多练，多尝试。

但是，绝大部分朋友都会遇到的一个共同的困惑。真实项目的线上服务器管控是非常严格的，大概率你是接触不到的。接触不到服务器意味着你没有服务优化的经验。而没有经验，就更不会让你去接触服务器了。那怎么打破这个死局呢？我的建议是跟优秀的开源软件学习。因为这是所有人都能够接触到的，质量最靠谱的java程序了。

例如，在RocketMQ中，有一个核心服务NameServer，这也是一个Java编写的应用程序。他的运行脚本是这样的：

```
choose_gc_options()
{
    # Example of JAVA_MAJOR_VERSION value : '1', '9', '10', '11', ...
    # '1' means releases befor Java 9
    JAVA_MAJOR_VERSION=$("$JAVA" -version 2>&1 | awk -F '"' '/version/ {print $2}' | awk -F '.' '{print $1}')
    if [ -z "$JAVA_MAJOR_VERSION" ] || [ "$JAVA_MAJOR_VERSION" -lt "9" ] ; then
      JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
      JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSInitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled
 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8 -XX:-UseParNewGC"
      JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps"
      JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=30m"
    else
      JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
      JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerM
B=0"
      JAVA_OPT="${JAVA_OPT} -Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,tags:filecount=5,filesize=30M"
    fi
}

choose_gc_log_directory
choose_gc_options
JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
JAVA_OPT="${JAVA_OPT} -XX:-UseLargePages"
#JAVA_OPT="${JAVA_OPT} -Xdebug -Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
JAVA_OPT="${JAVA_OPT} ${JAVA_OPT_EXT}"
JAVA_OPT="${JAVA_OPT} -cp ${CLASSPATH}"

"$JAVA" ${JAVA_OPT} $@
```

> 现在，你可能对RocketMQ这个中间件还完全不了解，没关系，这不是现在的关键，你只要知道他就是一个和我们自己写的Java程序一样的应用程序就可以了。

从这个脚本当中你可以看到，RocketMQ在运行这个java程序时，定制了非常多的参数。而这其中有很多参数，你应该是已经有过接触的了。

这段脚本当中，你唯一感到陌生的，应该就是其中那个JAVA_MAJOR_VERSION指令了。其实那个指令就是打印出当前环境的JDK版本。

```
JAVA -version 2>&1 | awk -F '"' '/version/ {print $2}' | awk -F '.' '{print $1}'
17
```

这样你就能看到，在choose_gc_options函数中，其实就是根据JDK版本不同，定制不同的GC参数。而这，其实就是我们需要重点学习的，如何在真实环境当中选择合适的参数组合了。

接下来，我们也就以RocketMQ的这一个脚本作为示例，来整体回顾一下如何定制一个JAVA程序合理的运行参数。

我们先来重点关注choose_gc_options函数中的调优参数。

对于9以前的版本，RocketMQ选择的GC参数是这样的

```
      JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -
XX:MaxMetaspaceSize=320m"
      JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -
XX:CMSInitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled -
XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8 -XX:-
UseParNewGC"
      JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -
XX:+PrintGCDetails -XX:+PrintGCDateStamps"
      JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -
XX:GCLogFileSize=30m"
```

而对于9以后的版本，RocketMQ选择的GC参数是这样的：

```
      JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -XX:MetaspaceSize=128m -
XX:MaxMetaspaceSize=320m"
      JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25
-XX:InitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerMB=0"
      JAVA_OPT="${JAVA_OPT} -
Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,tags:filecount=5,filesize=30M"
```

JVM的课程学到现在，这些参数你应该是会有一点点感觉的吧。发现他们的共同点了没有？其实不管在哪个JDK版本下，RocketMQ的这个脚本中，定制GC参数大概都分为这三个步骤。

- 调整内存布局
- 选择具体的GC算法，并定制GC算法的部分参数
- 打印GC日志

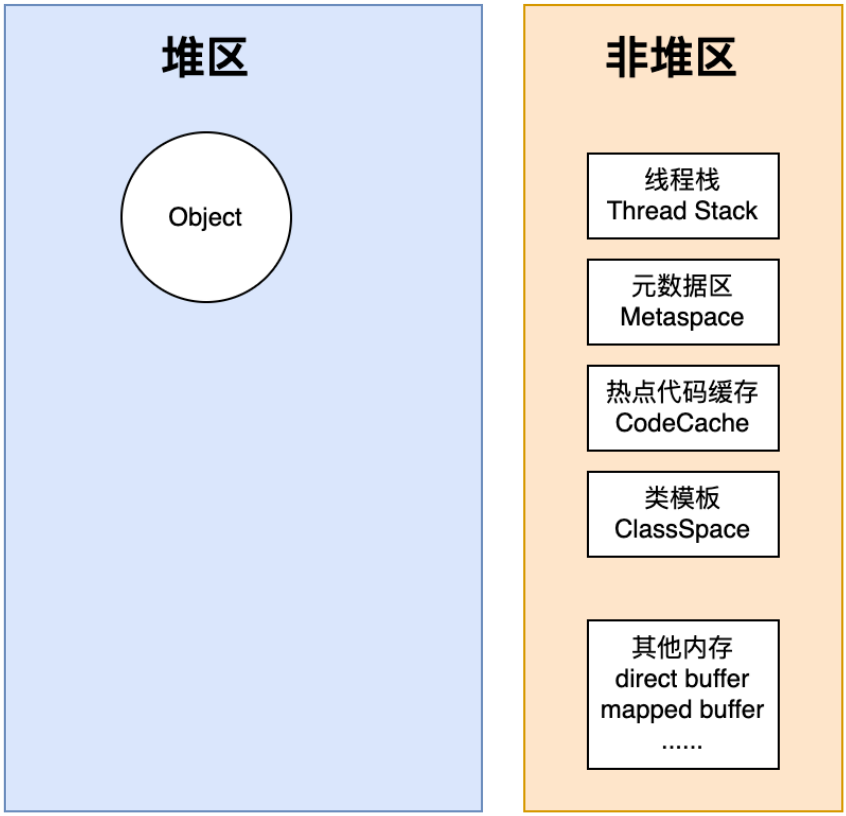
其实，如果你对RocketMQ有一定的了解，那么你会发现，在RocketMQ中，对另外一个关键服务Broker，其实也是按照这样的思路来定制参数的。对于NameServer和Broker这两种服务，他们的业务场景是不一样的，但是RocketMQ都是按照相同的思路进行参数调优的。而这种思路，其实也是我们学来用到我们自己项目当中的。当然，我们要学习的是这种参数调优的思路，而不是复制粘贴。

> 再次强调，对于RocketMQ的Broker和NameServer不了解没有关系，这不是今天的重点。我们今天要讨论的并不是RocketMQ这个特定的业务场景，而是要开始逐渐能够跟着开源软件，学习并积累JVM参数调优的经验。毕竟，JVM那么多参数，你不可能全部学会。所以，你要学的，是如何在项目中挑选重要的相关参数。

接下来，我们就按照这个思路进行详细的梳理。

三、基于JDK17优化JVM内存布局

所有的方法以及GC活动，都发证在JVM内存中，所以，第一步首先是需要定制JVM整体的内存布局。之前带大家用arthas看过，JVM运行时，内存整体会分为堆区和非堆区。



其中堆区Heap是JVM所管理的内存中最大的一块，主要用于存放各种对象实例。 由于JAVA实现了GC垃圾自动回收，所以对于堆内存的管理方式，会根据GC不同，而有很大的差距。

另外一块非堆区则是JVM中相对不是很繁忙的一块内存。这一块内存进行GC垃圾回收的频率相对较低，因此，对这块内存的管理方式相比堆区更加固定。

而我们首先要做的，就是定制JVM的堆区和非堆区的整体布局。接下来再根据选择的GC算法，定制其中的管理细节。

1、定制堆内存大小

堆内存的大小极大的决定了JAVA程序执行的性能，但也并不是越大越好。更大的堆内存，固然让JVM能够存下更多的对象，但同时，也加大了GC线程的压力。

主要涉及到以下几个参数：

- -Xms : 设置堆内存的初始大小。

默认单位是bytes。这个值必须是1024的整数，并且必须大于1M。如果不想设置这么精确，也可以在数字后面加k或者K表示KB，m或者M表示MB，g或者G表示GB。例如 -Xms62991456 , -Xms6144k, -Xms6m。

如果不设置这个值，JVM将会默认将堆内存的初始大小设置为老年代与年轻代的内存之和。

另外，有一个不稳定参数 -XX:InitalHeapSize也可以用来设置堆内存的初始大小。如果他出现在-Xms之后，那么初识对内存大小将最终由这个参数指定。

- -Xmx : 设置堆内存的最大大小。

  配置方式和-Xms一样。只不过，他的默认值可以在运行时基于操作系统自行决定。

  -Xmx配置等同于-XX: MaxHeapSize。

在服务端进行部署时，通常将-Xms和-Xmx设置成相同的值，减少JAVA应用在运行过程中的临时内存申请行为。但是，如果内存资源比较紧张，那就需要JVM能够按需索取内存。先申请一小部分内存，内存不够了，再申请一部分新的内存空间。这时，有下面两个参数需要稍微关注下。

- -XX:MinHeapFreeRatio -XX:MinHeapSize: 设置一次GC后所允许的堆空间的最小值。如果剩余的堆空间降落到这个阈值之下，这时堆空间就会启动一次扩充。这两个参数一个是比例，一个是大小。

> 结合指令 java -XX:+PrintCommandLineFlags -version分析

2、定制非堆内存大小

非堆空间存储的内容相比堆空间，就更安分一些。非堆空间的大部分内容通常变动都不会很大，往往也可以更多的交由JVM统一进行管理。例如ClassSpace主要存储类模板，而一个JAVA程序的类信息，绝大部分都是在JAVA程序启动过程中就加载到内存中，并且几乎不会有什么变化。(虽然也可以通过自定义类加载器，在运行时加载更多的类，但无论是使用频率还是类数量，通常都非常少。)

设置元空间

MetaSpace元空间主要存储的是JAVA类的一些元信息。这包括类的结构信息，如字段、方法、注解等，以及运行时常量池、字段和方法字节码等。简单来说，MeatSpace存储了JAVA程序运行所必须得类型信息。这些信息，很显然，你不可能具体限制他的大小。只要需要，多大的空间都必须提供。

在JDK8以前的版本中，类的元数据存储在永久代(PermGen)中。然而，从JDK8以后，永久代被移除，取而代之的就是MetaSpace元空间。与永久代不同的是，MetaSpace并不使用JVM的内存，而是直接使用本地内存，这意味着Metaspace的大小不再受限于JVM内存大小的限制，而是受操作系统可用内存的限制。需要注意的是，这样也不意味着Metaspace的大小完全不受限，如果本地内存耗尽，还是会导致OutOfMemmoryError异常的。

通常需要通过下面参数控制元空间大小。

- -XX:MetaspaceSize : 元空间大小

  这里设置的并不是元空间具体的大小，而是当元空间大小超过这个阈值时，就会触发一次GC。而在后续运行过程中，触发GC的阈值会根据元空间的使用情况进行自动调整。元空间的默认值取决于平台。

- -XX:MaxMetaspaceSize : 元空间最大值

  设置元空间大小的最大值。默认元空间是无限制的。元空间的大小取决于应用本身以及操作系统可提供的内存大小。一个应用程序中，元空间内的数据大小是不应该经常发生变动的，所以，设定一个合理的最大值，可以尽早避免一些非正常的元空间数据暴涨对操作系统的影响。

设置线程栈空间

JAVA进程在运行时，会为每个进程开辟一块内存，用来执行线程中的对应指令。整个内存是一个栈结构，先进后出。线程中执行的每个方法对应栈空间中的一个方法帧。方法帧中主要包含了程序计数器、操作数栈、局部变量表、返回地址等几个标准部分。另外，某些具体虚拟机实现还会添加一些附加信息。例如HotSpot中还添加了动态链接库以及一些自定义的附加信息。

线程栈空间大部分的内存都会随着方法结束而释放，所以通常不需要单独设置。但是如果你的应用中的方法嵌套非常多，或者有很多长期执行的复杂方法，那么就需要调整栈空间大小。如果栈空间内存不够，就会抛出StackOverFlowException。

> 展示示例

主要涉及到下面参数：

- -Xss : 设置线程栈空间的大小

  栈空间默认值大小，在Linux和MacOS系统中，都是1024KB。在Windows中，则需要依靠配置的虚拟内存大小决定。例如 -Xss1m, -Xss1024k。这样。

  配置栈空间大小，还可以用另外一个参数： -XX:ThreadStackSize。这个参数的作用和-Xss是差不多的，只是配置方式稍有不同。例如 -XX:ThreadStackSize=1K, -XX:ThreadStackSize=1024k。

设置热点代码缓存空间

在JAVA中，-server选项表示JVM以服务器模式运行。在服务器模式下，HotSpot虚拟机会将执行频率高的热点代码识别出来，提前进行编译，并将编译结果缓存起来。后续执行时，就可以以编译执行的方式直接读取缓存，而不用再一行代码一行代码的进行解释执行。而这些热点代码，就保存在非堆区的CodeSpace中。

热点代码缓存空间也涉及到几个核心参数：

- -XX:InitialCodeCacheSize=size

  设定代码缓存空间的初始大小

- -XX:ReservedCodeCacheSize=size

  设定代码缓存空间的最大大小。代码缓冲区最大大小默认值是240MB。如果禁止提前编译(-XX:-TieredCompilation) ，那么最大的大小默认是48MB。如果自己指定，这个值不能比初始值小。

- -XX:+SegmentedCodeCache 启用代码缓存分割

  这是JDK8中没有的一个参数，在JDK17中默认启用。这个参数的作用主要是优化代码缓存空间的内存使用。如果没有打开这个选项，那么所有的代码缓存是一个大的内存片段，这不利于内存空间的灵活使用。

  这个机制如果需要生效，还需要启用提前编译-XX:+TieredCompilation 并且-XX:ReservedCodeCacheSize >= 240 MB 。

> JDK8中没有这个特性

如果你对这个代码缓存分割感兴趣，后面几个参数或许能帮你大概了解一下，JVM底层是如何对代码缓存进行分割的。

- -XX:ProfiledCodeHeapSize=size ;
- -XX:NonNMethodCodeHeapSize=size ;
- -XX:NonProfiledCodeHeapSize=size ;

启用代码缓存分割后，JVM底层就是将代码缓存划分成这三个部分的。

应用程序类数据共享

关于元数据区，补充一个比较小众的JVM优化机制。

Application Class Data Sharing(应用程序列数据共享，简称AppCDS)是一种旨在提高运行相同代码的多个JVM的启动时间，并减少他们的内存占用的一种优化机制。

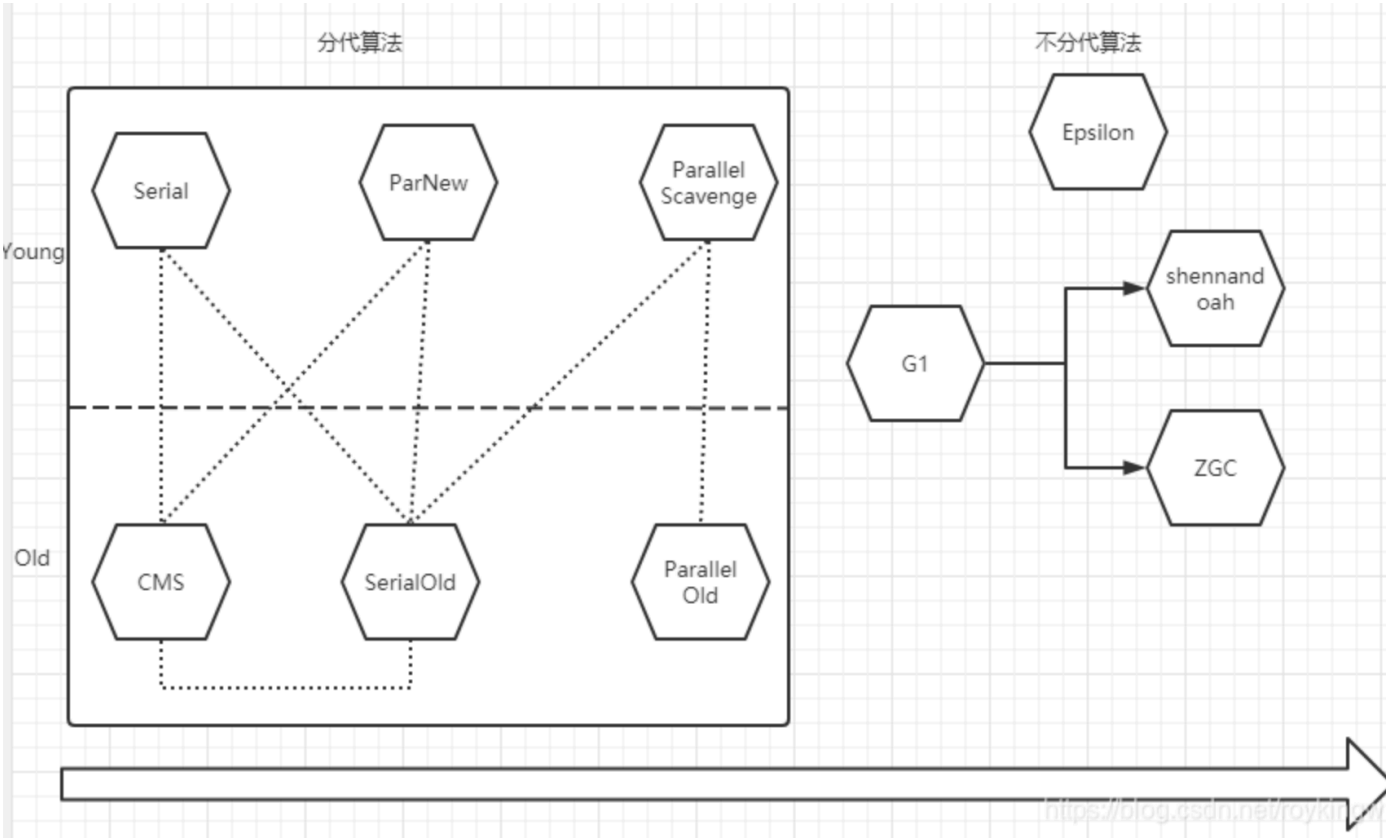
使用AppCDS机制，可以在JVM第一次运行时，对他加载过的类的数据进行收集并归档，记录到数据文件中。之后，这些数据文件还可以被后续的JVM进程使用。相比于每次从class文件中加载类信息，AppCDS可以节省JVM初始化过程中的时间和资源。

```
# 将类信息归档到hello.jsa文件中。
java -Xshare:dump -XX:SharedArchiveFile=hello.jsa -version
# 使用归档文件启动，并打印类加载日志
java -XX:SharedArchiveFile=hello.jsa -Xlog:class+load -version
# 有hello.jsa文件，加载的最后一个类
[0.021s][info][class,load] java.nio.charset.CoderResult source: shared objects file
# 删掉hello.jsa文件后，依然可以加载类，但是比有归档文件时会慢一些
[0.038s][info][class,load] java.nio.charset.CoderResult source: jrt:/java.base
```

在部署微服务应用时，这会是一个可选的方式。

四、基于JDK17定制JVM的GC参数

设定完整体的内存布局后，接下来就到了最重要的环节，优化GC参数。不同的GC算法，对内存的管理方法也不同。



关于各种GC算法的机制，之前课程中已经进行过讲解，这里就不再详细分析各种GC算法，只是从参数调优的角度进行学习。注意，这同样是一个没有标准答案的过程，不要希望一套配置打天下。多上阵，多试错才是唯一正确的方法。比如以后学习RocketMQ时，自己多尝试尝试修改不同的参数组合，会有什么不同的效果。

从上面RocketMQ的案例也能看到，在优化整体布局时， 在JDK8以前版本时，还设定了-Xmn参数，但是在JDK8以后的版本中，就没有设置这个参数，这其实也跟GC算法有关。

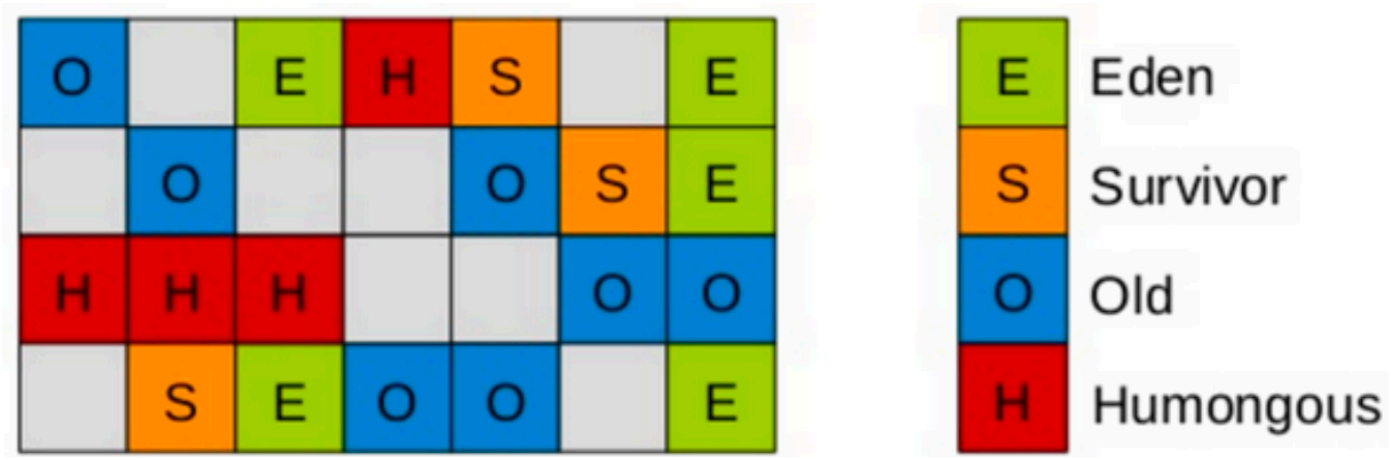
- -Xmn :

  设置分代收集的GC中年轻代的最大大小。堆中的年轻代用于存放新new出来的对象。年轻代的GC会比其他区域更频繁。年轻代太小，就会导致过于频繁的youngGC，而如果年轻代过大，又会导致youngGC的回收效果变差，加大fullGC的压力，从而进一步影响整个程序的运行效率。官方明确建议，对于基础的分代收集器，建议保持年轻代的大小在整个堆内存的25%到50%之间。而对于G1垃圾回收器，由于G1不再有严格固定的年轻代，所有官方明确建议，对G1垃圾回收器，不要设置-Xmn参数。

在JDK17中，已经取消了CMS算法，所以，接下来，主要针对JDK17中的G1和ZGC，分析相关常见的重要参数。

G1重要参数

优化G1的配置之前，还是需要先回顾一下G1的基本思想。



G1 GC 是一种分代的、并发的、基于区域的垃圾回收器，它将堆内存划分为多个独立的区域（Regions），每个区域可以是 Eden 区、Survivor 区或者 Old 区。为了保持系统的响应性，G1 GC 会尽量在达到用户设定的停顿时间目标（通过 -XX:MaxGCPauseMillis 参数指定）前进行垃圾回收。

> 之前的课程中，我们已经基于JDK8，给大家介绍了G1的回收机制，这里就不再过多介绍G1的工作机制，只是从实战角度理解对于G1应该做哪些考虑。

G1虽然还是一个分代的垃圾回收器，但是他的各个Region的划分并不固定，所以，在使用G1垃圾回收器时，请忘记以往非常熟悉的堆内存分代模型。比如-Xmn(年轻代空间大小)，-XX:NewRatio(年轻代与老年代比例)，-XX:SurvivorRatio(年轻代中eden区与suvivor区的比例)等等这些参数。对于G1垃圾回收器，最为核心的参数就是三个： **-XX:+UseG1GC(启动G1)，-Xmx(堆内存大小)，-XX:MaxGCPauseMillis(期望G1达到的最大停顿时间，默认200毫秒)**

接下来，我们就以JDK17为标准，结合RocketMQ的运行脚本，整理一下在实际项目中，还有哪些需要了解的G1相关参数。在之前的课程中，我们已经基于JDK8，给大家详细介绍了G1的回收机制，其中也详细介绍了G1的核心参数。这里就以JDK17为标准，结合RocketMQ的运行脚本，整理一下RocketMQ这样一个开源软件是如何在项目中优化G1的。

> 相比JDK8，JDK17版本中的G1整体并没有太大的变动，参数部分整体变化也不太大。所以，可以参照之前的课程内容进行补充。

- -XX:+UseG1GC :

  使用G1垃圾回收器。在JDK17中，是默认选项。G1垃圾回收器适合那些需要大量堆内存(建议是6GB以上)同时还需要有稳定的GC延迟(STW延迟时间稳定在0.5秒以下)的应用。

- -XX:G1HeapRegionSize=size

  设定每个Region的大小。这个值必须是2的N次幂，且范围在1MB到32MB之间。这是G1最为重要的一个参数。

  这个参数的默认值是不固定的，通常JVM会将堆内存划分为2048个Region。每个Region的大小就是 堆内存/2048。

  从这里看到，RocketMQ将这个参数设定成了16M，这是一个偏大的设置了。较大的Region区域可以减少GC的频率，从而降低停顿时间的影响。但是，这也意味着增加了每次GC回收的停顿时间。所以，设定这个参数时，需要在停顿时间和数据吞吐量之间进行权衡。

- -XX:G1ReservePercent=percent

  这个参数是告诉G1，为了满足停顿时间的整体目标时，应该保留多少比例的堆空间作为空闲。这样，在突发的内存分配需求活垃圾回收效率下降时，G1任然有足够的缓冲空间来避免长时间的停顿。

  这个参数的默认值是10%，这意味着堆内存的10%将被保留为空闲状态。这显然是一种空间换时间的策略，而RocketMQ将这个值设定为25%，这也是为了追求性能做的一种取舍。显然你也能猜到，RocketMQ这样的设定，如果在内存不太够的情况下，反而会造成内存更加紧张。

- -XX:InitiatingHeapOccupancyPercent=percent

  这个参数的默认值是45，表示当整个堆中，老年代Region达到45%时，G1就会开始并发标记周期。RocketMQ将这个参数设定为30，显然是为了更积极的进行GC。

  但是，并不意味着每次堆内存用到了30%就开始GC，与此相关的还有另外两个参数：-XX:G1UseAdaptiveIHOP 和 -XX:G1AdaptiveIHOPNumInitialSamples。

  其中，-XX:G1UseAdaptiveIHOP 是一个bool型的参数，默认是启动的。这个参数启动后，G1只会在 -XX:G1AdaptiveIHOPNumInitialSamples 参数指定的前面几次GC活动中按照-XX:InitiatingHeapOccupancyPercent 参数进行计算。之后，G1会自动根据目标调整参数。-XX:G1AdaptiveIHOPNumInitialSamples 的默认值是3。

  所以，RocketMQ采用这种比较激进的设计，其实只是为了让RocketMQ的服务在启动时更稳健。

- -XX:SoftRefLRUPolicyMSPerMB=time

  这个参数就比较隐蔽了。他表示在每MB的堆内存中，软引用经过多长时间才被认为过期。默认值是1000，表示1秒。

  > Java中4种引用的级别和强度由高到低依次为：强引用 -> 软引用 -> 弱引用 -> 虚引用
  >
  > 1 强引用 StrongReference
  >
  > 强引用是最普遍的引用，如果一个对象有强引用，那垃圾回收期绝不会回收该对象。
  >
  > 2 软引用 SoftReference
  >
  > 如果一个对象只有软引用，则内存空间充足时，垃圾回收期不变会回收。但是如果内存空间不足，垃圾回收器就会回收该对象，节省内存空间。
  >
  > 3 弱引用 WeakReference

> 弱引用与软引用的区别在于，只有弱引用的对象拥有更短暂的生命周期。垃圾回收器线程在扫描他所管辖的内存区域时，一旦发现只具有弱引用的对象，不管当前内存是否足够，都会立即回收。不过，由于垃圾回收器线程是一个优先级非常低的线程，因此不一定会很快发现这些只具有弱引用的对象。弱引用可以重新声明成强引用。
>
> 4 虚引用 PhantomReference
>
> 虚引用顾名思义，就是形同虚设。与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
>
> 设置虚引用的唯一目的就是在这个对象被GC回收时，收到一个系统的通知或者后续添加进一步的处理。例如触发对象的finalize方法。还有JAVA当中使用直接内存directBuffer时，也会使用虚引用来保证回收对象时能释放对应的直接内存。

这个参数对于控制内存使用和避免内存泄漏非常有用。通过调整这个参数，可以平衡内存使用和对象生命周期的需求。如果设置得太低，可能会导致频繁地清理软引用对象，增加 GC 的负担；如果设置得太高，则可能会导致内存占用过高，甚至发生 OutOfMemoryError。

软引用在项目当中通常用得比较少。所以，如果你的项目中也几乎没有用到过软引用，那么不妨可以考虑下参照RocketMQ的经验，进行一下尝试。

另外，相比JDK8，JDK17中新增了几个G1相关的参数

- -XX:ParallelGCThreads=threads

  设置GC的工作线程数。默认值取决于GC算法以及CPU核心数。比如对于G1，可以通过以下方式设置线程数为2： -XX:ParallelGCThreads=2

- -XX:G1HeapWastePercent=percent

  设置堆空间的浪费比例。HotSpot虚拟机在对空间的使用比例低于这个值时不会启动GC周期。默认值是5%。

- -XX:G1OldCSetRegionThresholdPercent=percent
  设置一次混合GC中需要清理的Old区的内存比例。默认值是堆空间的10%。这也是G1非常重要的一个参数。将他调大，可以降低G1的频率，但是会让每一次GC的时间变长。

- -XX:G1MixedGCCountTarget=number
  设置G1垃圾回收器的线程上限。HotSpot会为了达到清理G1OldCSetRegionThresholdPercent比例的Old区的目标，会自动计算需要启动几个G1垃圾回收器。但是垃圾回收器的个数不会超过这个上限。默认值是8.

> 还有很多参数，就不一一列举了。你需要的是要用到的时候知道怎么去查。

ZGC重要参数

ZGC从JDK11版本开始引入，JDK13开始发布正式版本。后续每个JDK版本都对ZGC有一定的优化，到JDK17版本已经基本稳定，官方建议已经可以用于生产环境。

> ZGC之前课程中同样介绍过，这里就不做过多介绍。

但是目前业界使用ZGC的场景还是比较少，一方面是很多项目的JDK版本还没有跟上，另一方面也是因为大部分项目中用到的内存还没有达到需要ZGC出马的量级。因此实际使用ZGC的案例现在还比较少。

这里就只是帮大家整理一下官方对于ZGC核心参数的配置说明。目前阶段，大家了解一下即可。

- **-XX:+UseZGC**
  Enables the use of the Z garbage collector (ZGC). This is a low latency garbage collector, providing max pause times of a few milliseconds, at some throughput cost. Pause times are independent of what heap size is used. Supports heap sizes from 8MB to 16TB.
- **-XX:ZAllocationSpikeTolerance=factor**
  Sets the allocation spike tolerance for ZGC. By default, this option is set to 2.0. This factor describes the level of allocation spikes to expect. For example, using a factor of 3.0 means the current allocation rate can be expected to triple at any time.
- **-XX:ZCollectionInterval=seconds**
  Sets the maximum interval (in seconds) between two GC cycles when using ZGC. By default, this option is set to 0 (disabled).
- **-XX:ZFragmentationLimit=percent**
  Sets the maximum acceptable heap fragmentation (in percent) for ZGC. By default, this option is set to 25. Using a lower value will cause the heap to be compacted more aggressively, to reclaim more memory at the cost of using more CPU time.
- **-XX:+ZProactive**
  Enables proactive GC cycles when using ZGC. By default, this option is enabled. ZGC will start a proactive GC cycle if doing so is expected to have minimal impact on the running application. This is useful if the application is mostly idle or allocates very few objects, but you still want to keep the heap size down and allow reference processing to happen even when there are a lot of free space on the heap.
- **-XX:+ZUncommit**
  Enables uncommitting of unused heap memory when using ZGC. By default, this option is enabled. Uncommitting unused heap memory will lower the memory footprint of the JVM, and make that memory available for other processes to use.
- **-XX:ZUncommitDelay=seconds**
  Sets the amount of time (in seconds) that heap memory must have been unused before being uncommitted. By default, this option is set to 300 (5 minutes). Committing and uncommitting memory are relatively expensive operations. Using a lower value will cause heap memory to be uncommitted earlier, at the risk of soon having to commit it again.

从参数数量就能看到， ZGC暴露出来的参数非常少。而算法方面，ZGC更多的被设计为根据运行环境进行自适应调整。比如GC的并发线程数这类参数，ZGC都已经完成了自行调整。在绝大部分情况下，对ZGC，不调整就是最好的调整。根据现在公布的情况来看 ，对ZGC，基本只需要指定堆大小–Xmx 即可。而优化配置的方式只需要在内存使用量与GC运行频率之间找到一个平衡点。

五、GC日志处理

GC日志的重要性，在之前课程中已经给大家做过介绍。这里就不再做过多讲解，只是根据RocketMQ的经验，整理一下要如何打印GC日志。

在JDK8以前，JVM中的日志打印是比较混乱的，很多日志被分散在多个不同的地方，也需要很多不同的参数进行打印。从JDK8往后，JVM的日志得到了极大的精简，所有日志打印相关的指令，都集中到了-Xlog 选项当中。这里就直接整理一个列表，比较JDK8与JDK17在日志方面的参数差异：

Legacy Garbage Collection (GC) Flag	Xlog Configuration	Comment
`GlPrintHeapRegions`	`-Xlog:gc+region=trace`	Not Applicable
`GCLogFileSize`	No configuration available	Log rotation is handled by the framework.
`NumberOfGCLogFiles`	Not Applicable	Log rotation is handled by the framework.
`PrintAdaptiveSizePolicy`	`-Xlog:gc+ergo*=`*level*	Use a *level* of `debug` for most of the information, or a *level* of `trace` for all of what was logged for `PrintAdaptiveSizePolicy`.
`PrintGC`	`-Xlog:gc`	Not Applicable
`PrintGCApplicationConcurrentTime`	`-Xlog:safepoint`	Note that `PrintGCApplicationConcurrentTime` and `PrintGCApplicationStoppedTime` are logged on the same tag and aren't separated in the new logging.
`PrintGCApplicationStoppedTime`	`-Xlog:safepoint`	Note that `PrintGCApplicationConcurrentTime` and `PrintGCApplicationStoppedTime` are logged on the same tag and not separated in the new logging.
`PrintGCCause`	Not Applicable	GC cause is now always logged.
`PrintGCDateStamps`	Not Applicable	Date stamps are logged by the framework.
`PrintGCDetails`	`-Xlog:gc*`	Not Applicable
`PrintGCID`	Not Applicable	GC ID is now always logged.
`PrintGCTaskTimeStamps`	`-Xlog:gc+task*=debug`	Not Applicable
`PrintGCTimeStamps`	Not Applicable	Time stamps are logged by the framework.
`PrintHeapAtGC`	`-Xlog:gc+heap=trace`	Not Applicable
`PrintReferenceGC`	`-Xlog:gc+ref*=debug`	Note that in the old logging, `PrintReferenceGC` had an effect only if `PrintGCDetails` was also enabled.
`PrintStringDeduplicationStatistics`	`-Xlog:gc+stringdedup*=debug`	Not Applicable
`PrintTenuringDistribution`	`-Xlog:gc+age*=`*level*	Use a *level* of `debug` for the most relevant information, or a *level* of `trace` for all of what was logged for `PrintTenuringDistribution`.
`UseGCLogFileRotation`	Not Applicable	What was logged for `PrintTenuringDistribution`.

例如RocketMQ中对JDK8以后的版本，统一采用以下参数打印GC日志

```
-Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,tags:filecount=5,filesize=30M
```

其中 -Xlog:gc* 表示打印每次GC的详细信息，等同于JDK8中的 -XX:PrintGCDetails。后面的file分为三个部分：

第一部分是文件名

第二部分表示历史文件的后缀，有以下几个选项：

`time`(`t`), `utctime`(`utc`), `uptime`(`u`), `timemillis`(`tm`), `uptimemillis`(`um`), `timenanos`(`tn`), `uptimenanos`(`un`), `hostname`(`hn`), `pid`(`p`), `tid`(`ti`), `level`(`l`), `tags`(`tg`)

第三部分表示历史文件的个数和大小。RocketMQ中的配置就表示保留5个文件，每个文件写满30M就切换下一个文件。

> 还记得之前给大家分享过的通过gceasy网站辅助分析GC日志吗？以下是我部署一次RocketMQ后，拿到NameServer日志，gceasy给出的分析日志。这就可以作为JVM调优的基础。

GC 情报

rmq_srv_gc_pid12078_2023-08-08_17-45-47.log.0.current

时长: 93 days 2 hrs 4 min 5 sec

祝贺！您程序的 GC 活动状态健康。

建议

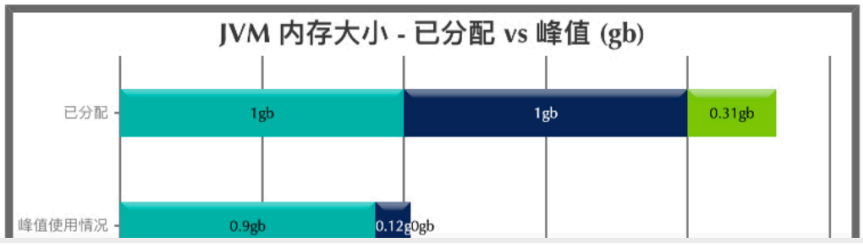
(CAUTION: Please do thorough testing before implementing below recommendations.)

- You are using CMS (Concurrent Mark & Sweep) GC algorithm. CMS GC has been deprecated in Java 9. You may consider switching to different GC algorithm. For more details refer to **refer here.**
- -XX:+UseGCLogFileRotation argument is not recommended to be passed. It can have undesirable effects. To learn more about it's effects and alternative solution refer to **refer here.**
- -XX:+UseCompressedOops is not required to be passed, if you are running in Java SE update 23 and later. Compressed oops is supported and enabled by default in Java SE 6u23 and later versions. For more details, **refer here.**
- It looks like you have over allocated Old Generation size. During entire run, Old Generation's peak utilization was only **12.48%** of the allocated size. You can consider lowering the Old Generation Size.
- It looks like you have over allocated Heap size. During entire run, Heap's peak utilization was only **46.28%** of the allocated size. You can consider lowering the Heap Size.

JVM 内存大小

(To learn about JVM Memory, click here)

代	已分配	峰值
Young 代	1 gb	921.62 mb
Old 代	1 gb	127.83 mb
Metaspace	320 mb	n/a



六、其他JVM调优小经验

接下来，仔细查看下RocketMQ的启动脚本，会发现一行注释掉了的配置信息：

```
#JAVA_OPT="${JAVA_OPT} -Xdebug -
Xrunjdwp:transport=dt_socket,address=9555,server=y,suspend=n"
```

一个精益求精的开源软件，为什么要保留这么一行注释掉的配置呢？其实这里就隐藏了一个非常有用的开发技巧。就是远程进行断点调试。

远程断点调试，简单来理解， 就是允许我们用本地Debug断点调试的方式，去调试在远端服务器上运行的应用程序。这个技巧对于开发一些复杂软件，是非常有用的。因为软件运行情况怎么样，只有部署到真实服务器上才能得到真正的验证。

我们可以自己做个小案例来理解一下这种远程调试的方案。

```java
package com.roy;

import java.util.Scanner;
//-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
public class RemoteDebugTest {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String command="";
        int count = 0;
        do{
            command = scanner.next();
            System.out.println("第"+(++count)+"个指令: "+command);
        }while (!"quit".equals(command));
        System.out.println("接收到退出指令");
        System.exit(-1);
    }
}
```

这个小案例主要就是通过监听控制台输入，来模拟一个长时间运行的计算程序。如果在IDEA上运行，那么平平无奇。但是，我们可以将这个文件在控制台启动，然后启动时，加上那一串莫名其妙的配置信息：

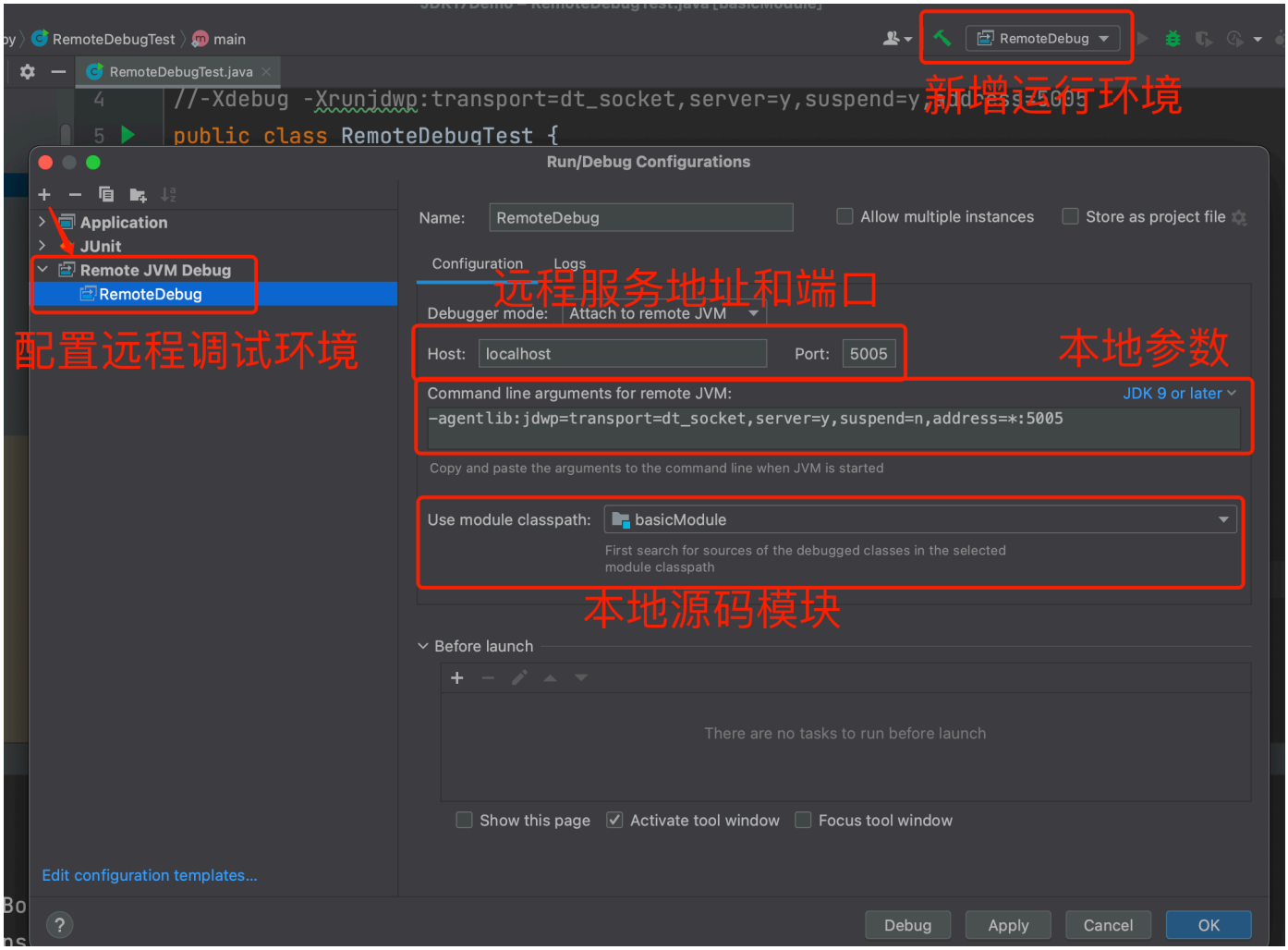
```
(base) roykingw@roykingwdeMacBook-Pro classes % java -Xdebug -
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 com.roy.RemoteDebugTest
Listening for transport dt_socket at address: 5005
```

启动后，就会在当前机器上启动一个应用监听，监听端口就是address参数指定的端口号。这样，就相当于在服务器上运行了一个JAVA程序。并且启动后，这个应用程序会阻塞住，等待开启远程监听。

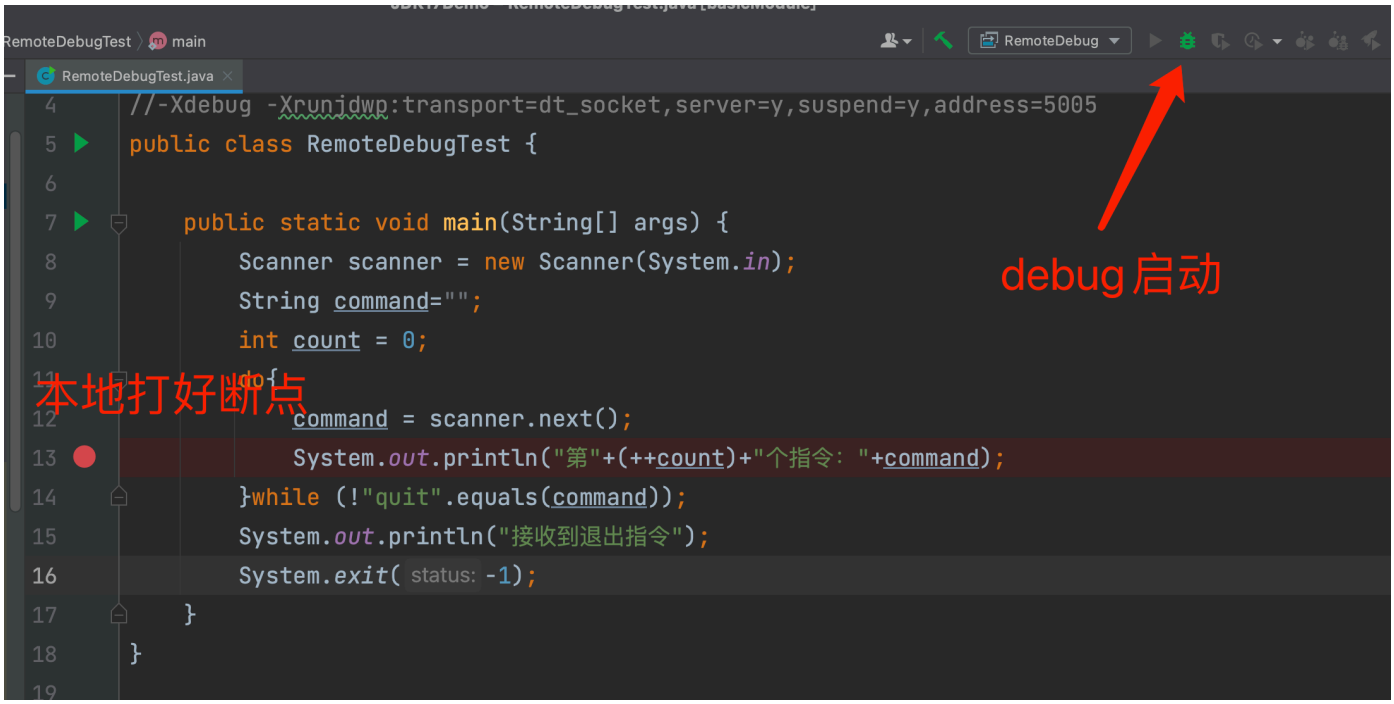
> 此时应用是阻塞住的，在控制台输入指令是没有用的。

接下来，我们就可以在本地IDEA配置远程调试。

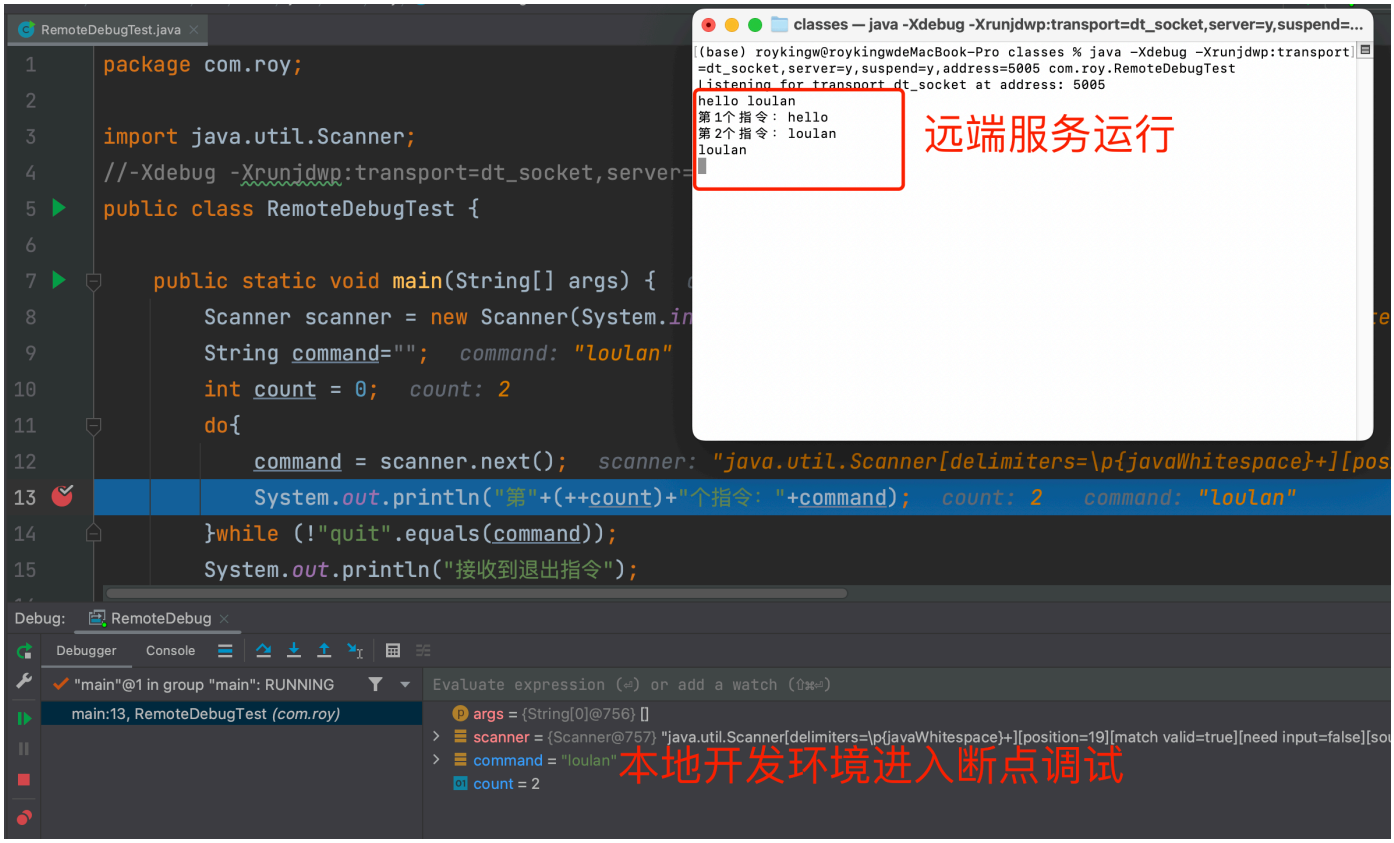
首先需要在IDEA中配置一个远端调试的运行环境：



然后，就可以在IDEA中打好断点，然后启动这个远程运行的环境。



接下来，在本地IDEA中debug启动刚才配置的远端JVM调试指令，就可以在远端服务器上正常执行应用。而本地IDEA也会在远端服务器运行到断点代码处时，自动进入断点调试模式。



我们这个小案例是用本地机器模拟的远端服务器，接下来，自己尝试一下，把你感兴趣的项目放到远端服务器上试试看。

但是要注意一下，这种远程调试的方式显然是不能用在生产环境的。因为打开远程调试后，服务端的应用程序必须监听到调试请求才会正常执行。如果你把IDEA中的调试任务终止了，远端的应用程序就会重新回归到阻塞状态。

当然，这样的小技巧其实并不需要你一定记得非常熟练，因为大部分的JAVA程序其实对服务器配置没有那么敏感。但是如果你后面需要接触一些spark，flink这样大型的计算框架时，记得楼兰老师教过你，从RocketMQ中学习过这样一个小技巧，就非常好了。

七、章节总结

我们第一个JVM性能调优专题，到这里就结束了。但是，JVM底层的这些知识，其实就像是武林高手的内功，见面三招可能用不上，但是，越往后越能体现他的价值。这些经过时间沉淀下来的经验，才是程序员最不可替代的核心竞争力。

关于JVM部分，我们这一期的课程暂告一段落，但是，大家的学习并没有结束。对于大家以后的JVM学习，我给大家三条具体的学习建议。

1、重框架

JVM这种底层语言要处理的问题也非常复杂，非常深，因此，JVM部分的知识，或者说面试点也是非常虚，非常杂的。任何一个细节知识点往下，都能够挖出无数多的问题。并且，由于JVM很难也没有必要去研究底层的源码，所以很多细节问题并不太好具体验证。

所以我建议大家对于JVM部分的问题，要重框架，而不要太注重细节。注重各个层面的逻辑自洽，而不要纠结于各个细节。与其花功夫研究茴香豆的茴字有几种写法，不如多花花功夫怎么写出一个自己的孔乙己。

2、形成习惯

JVM东西很多，所以要形成一个大的整体逻辑是不容易的。加上现在JDK各个新版本层出不穷，各种各样的新框架也在不断出现，所以，你的整体逻辑也要不断更新。但是，JVM这种底层知识又注定了你不太可能像Spring这样的应用框架，边用边熟悉。所以，这也主动了你不太可能每次都专门花上很长一段时间专门来学习或者复习JVM。而对于这种经常容易忘记的技术，最好的学习方式，就是形成思考习惯，收集更多零碎的时间，一点点思考积累。

这次课程中，我们从RocketMQ的角度梳理了一下JVM的一些应用场景。以后，你还会学更多的中间件，接触更多的实际项目，还要了解Jenkins，Docker，大数据等各种不同类型的Java项目部署运行环境。每接触一个新的环境，甚至以后每遇到一个新颖的JVM面试题，都不妨都像这样，顺便梳理一下他们的环境优化思路，补充自己不太熟悉的知识细节。

3、重表达

一个很现实的问题，对于JAVA程序员来说，JVM部分很多知识对于开发工作的直接帮助并不是很大。更大的作用，大概率会体现在以后面试或者处理一些稀奇古怪的问题的时候。而这个时候，大概率你需要的不只是自己明白问题，还要能够跟其他人表达清楚你的认识。但是，JVM这一部分的东西，非常凌乱，加上其中数不尽的细节问题，所以，我见过很多程序员对JVM这部分的东西，明明自己很清楚，但是跟别人说的时候就是表达不清楚。

所以我建议大家在学习JVM时，除了搞懂原理，一定要多练习表达。要争取在更短的时间内，把问题表达清楚。如果有时间，我建议大家每个人都来重构一下JVM篇的第一节课。

作为第一个篇章的结尾，恭喜你找到了图灵这个最为专注，并且开放的学习组织。预祝咱们顶峰相见。

> 参考资料： JDK工具官网文档：https://docs.oracle.com/en/java/javase/17/docs/specs/man/index.html
>
> JDK17的java指令的官方文档： https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html
>
> JDK8的java指令的官方文档： https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BGBCIEFC

> 有道云笔记：https://note.youdao.com/s/SxNl4ZdO