

## 如何查看jvm当前使用的是什么垃圾回收器

windows: `java -XX:+PrintFlagsFinal -version | FINDSTR /i ":"`

Linux :

`java -XX:+PrintFlagsFinal -version | grep :`

或

`java -XX:+PrintCommandLineFlags -version` (使用这种比较直观 个人感觉)

```
[root@VM_0_49_centos ~]# java -XX:+PrintCommandLineFlags -version
-XX:InitialHeapSize=128166912 -XX:MaxHeapSize=2050670592 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
[root@VM_0_49_centos ~]#
```

## 解读JVM使用的垃圾回收器属性

```
-XX:InitialHeapSize=128166912 -XX:MaxHeapSize=2050670592 -
XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -
XX:+UseCompressedOops -XX:+UseParallelGC
```

JDK1.8

默认初始化堆大小 : 128MB;

默认最大堆大小 : 2048MB;

### -XX:InitialHeapSize=128166912

初始化堆大小, 字节单位:  $128166912 / 1024 / 1024 = 122.2294921875$  MB  $\approx$  **128MB**

### -XX:MaxHeapSize=2050670592

最大堆大小,  $2050670592 / 1024 / 1024 / 1024 = 1.909$ GB  $\approx$  **2GB**

### -XX:+PrintCommandLineFlags

### -XX:+UseCompressedClassPointers -XX:+UseCompressedOops

这两个下面来讲解

### -XX:+UseParallelGC

Jvm运行在Server模式下的默认值, 打开此开关后, 使用Parallel Scavenge + Serial Old的收集器组合进行回收

执行: `java -XX:+PrintFlagsFinal -version | grep :`

```
Last login: Sun Oct 18 12:50:06 2020 from 172.16.1.4
[yypt@VM_0_49-centos ~]$ java -XX:+PrintFlagsFinal -version | grep :
    intx C1CompilerCount                := 2                {product}
    uintx InitialHeapSize                := 130023424          {product}
    uintx MaxHeapSize                    := 2051014656         {product}
    uintx MaxNewSize                     := 683671552          {product}
    uintx MinHeapDeltaBytes               := 524288            {product}
    uintx NewSize                        := 42991616           {product}
    uintx OldSize                        := 87031808            {product}
    bool PrintFlagsFinal                  := true              {product}
    bool UseCompressedClassPointers       := true              {1p64_product}
    bool UseCompressedOops                := true              {1p64_product}
    bool UseParallelGC                    := true              {product}
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
[yypt@VM_0_49-centos ~]$
```

InitialHeapSize = 130023424=124MB

=NewSize + OldSize

NewSize = 42991616/1024/1024=41MB

OldSize = 87031808/1024/1024=83MB

NewSize /OldSize = 1:2;

MaxHeapSize = 2051014656/1024/1024=1,956MB

MaxNewSize = 683671552/1024/1024=652MB

因为NewSize /OldSize = 1:2;

所以 MaxNewSize /MaxHeapSize =0.3333,即新生代 占 堆总大小的33.33%;

MaxHeapSize = MaxNewSize +

新生代将内存按照8:1:1分为一个Eden和so, s1三个区域; , 因为对象生命周期不一样, 所以产生了分代。

新生代: 老年代=1: 2

当新生代沾满时, 老年代会作为一个 分配担保, 保存到老年代。

## 垃圾收集器参数总结

参数	描述
-XX:+UseParallelGC	Jvm运行在Server模式下的默认值, 打开此开关后, 使用Parallel Scavenge + Serial Old的收集器组合进行回收

UseParallelGC ----> 是JDK8默认垃圾回收器

### JDK8默认垃圾回收器详解

今天面试问道JDK8默认使用的垃圾收集器是什么, 然后回来第一时间CMD命令查看了一下:

```
java -XX:+PrintCommandLineFlags -version

-XX:InitialHeapSize=266390080 -XX:MaxHeapSize=4262241280 -
XX:+PrintCommandLineFlags
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
12345678
```

UseParallelGC 即 Parallel Scavenge + Parallel Old,再查看详细信息

```
java -XX:+PrintGCDetails -version

java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
Heap
  PSYoungGen      total 76288K, used 2621K [0x000000076b500000,
0x0000000770a00000, 0x00000007c0000000)
    eden space 65536K, 4% used
    [0x000000076b500000,0x000000076b78f748,0x000000076f500000)
    from space 10752K, 0% used
    [0x000000076ff80000,0x000000076ff80000,0x0000000770a00000)
    to   space 10752K, 0% used
    [0x000000076f500000,0x000000076f500000,0x000000076ff80000)
  ParOldGen       total 175104K, used 0K [0x00000006c1e00000,
0x00000006cc900000, 0x000000076b500000)
    object space 175104K, 0% used
    [0x00000006c1e00000,0x00000006c1e00000,0x00000006cc900000)
  Metaspace       used 2291K, capacity 4480K, committed 4480K, reserved
1056768K
                    class space    used 254K, capacity 384K, committed
384K, reserved 1048576K
```

## UseCompressedOops / UseCompressedClassPointers

UseCompressedOops: 普通对象指针压缩, oops: ordinary object pointer

UseCompressedClassPointers: 类指针压缩

这两者有什么作用呢?

拿新建一个对象来说:

```
Object o = new Object()
```

如果不开启普通对象指针压缩, -UseCompressedOops, 会在内存中消耗24个字节, o 指针占8个字节, Object对象占16个字节。

如果开启普通对象指针压缩, +UseCompressedOops, 会在内存中消耗20个字节, o指针占4个字节, Object对象占16个字节。

这样一看，好像UseCompressedOops对Object的内存并没有影响，其实不然，Object对象在内存中的布局，包括markword、

class pointer、实例数据和填充对其，开启UseCompressedOops，默认会开启UseCompressedClassPointers，会压缩class pointer 这部分的大小，由8字节压缩至4字节，间接的提高内存的利用率。

由于UseCompressedClassPointers的开启是依赖于UseCompressedOops的开启，因此，要使UseCompressedClassPointers起作用，得先开启UseCompressedOops，并且开启UseCompressedOops也默认强制开启UseCompressedClassPointers，关闭UseCompressedOops默认关闭UseCompressedClassPointers。

如果开启类指针压缩，+UseCompressedClassPointers，根据上面的条件，结果跟只开启UseCompressedOops一样，会在内存中消耗20个字节，o指针占4个字节，Object对象占16个字节。

如果关闭类指针压缩，-UseCompressedClassPointers，根据上面的条件，UseCompressedOops还是会开启，会在内存中消耗20个字节，o指针占4个字节，Object对象占16个字节。

如果开启类指针压缩，+UseCompressedClassPointers，并关闭普通对象指针压缩，-UseCompressedOops，此时会报错，  
UseCompressedClassPointers requires UseCompressedOops

```
// UseCompressedOops must be on for UseCompressedClassPointers to be on.
if (!UseCompressedOops) {
    if (UseCompressedClassPointers) {
        warning("UseCompressedClassPointers requires UseCompressedOops");
    }
    FLAG_SET_DEFAULT(UseCompressedClassPointers, false);
}
```

[垃圾收集器参数总结](#)

参数	描述
-XX:+UseSerialGC	Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收
-XX:+UseParNewGC	打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收
-XX:+UseConcMarkSweepGC	使用ParNew + CMS + Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。
-XX:+UseParallelGC	Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old的收集器组合进行回收
-XX:+UseParallelOldGC	使用Parallel Scavenge + Parallel Old的收集器组合进行回收
-XX:SurvivorRatio	新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1
-XX:PretenureSizeThreshold	直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
-XX:MaxTenuringThreshold	晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代
-XX:UseAdaptiveSizePolicy	动态调整java堆中各个区域的大小以及进入老年代的年龄

参数	描述
-XX:+HandlePromotionFailure	是否允许新生代收集担保，进行一次minor gc后，另一块Survivor空间不足时，将直接会在老年代中保留
-XX:ParallelGCThreads	设置并行GC进行内存回收的线程数
-XX:GCTimeRatio	GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效
-XX:MaxGCPauseMillis	设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效
-XX:CMSInitiatingOccupancyFraction	设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70
-XX:+UseCMSCompactAtFullCollection	由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效
-XX:+CMSFullGCBeforeCompaction	设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用
-XX:+UseFastAccessorMethods	原始类型优化
-XX:+DisableExplicitGC	是否关闭手动System.gc
-XX:+CMSParallelRemarkEnabled	降低标记停顿
-XX:LargePageSizeInBytes	内存页的大小不可设置过大，会影响Perm的大小，-XX:LargePageSizeInBytes=128m

### Client、Server模式默认GC

	新生代GC方式	老年代和持久代GC方式
Client	Serial 串行GC	Serial Old 串行GC
Server	Parallel Scavenge 并行回收GC	Parallel Old 并行GC

### Sun/Oracle JDK GC组合方式

	新生代GC方式	老年代和持久代GC方式
-XX:+UseSerialGC	Serial 串行GC	Serial Old 串行GC
-XX:+UseParallelGC	Parallel Scavenge 并行回收GC	Serial Old 并行GC
-XX:+UseConcMarkSweepGC	ParNew 并行GC	CMS 并发GC 当出现“Concurrent Mode Failure”时 采用Serial Old 串行GC
-XX:+UseParNewGC	ParNew 并行GC	Serial Old 串行GC
	Parallel	

-XX:+UseParallelOldGC	<b>新生代GC方式</b>	<b>老年代和持久代GC方式</b>
	回收GC	
- XX:+UseConcMarkSweepGC -XX:+UseParNewGC	Serial 串行GC	CMS 并发GC 当出现“Concurrent Mode Failure”时 采用Serial Old 串行GC