## 深入理解JVM虚拟机

### 二、自动内存管理

#### 1、Java内存区域及内存溢出异常

##### 1）运行时的数据区域

注意：HotSpot不分虚拟机栈和本地方法栈



###### A）程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器

###### B）Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stack）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的线程内存模型：每个方法被执行的时候，Java虚拟机都会同步创建一个**栈帧**（StackFrame）用于**存储局部变量表、操作数栈、动态连接、方法出口**等信息。每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

###### C）本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别只是拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用的本地（Native）方法服务。

###### D）Java堆

Java堆（Java Heap）是虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是**存放对象实例**，Java世界里“几乎”所有的对象实例都在这里分配内存。在《Java虚拟机规范》中对Java堆的描述是：“**所有的对象实例以及数组都应当在堆上分配**”，而这里笔者写的“几乎”是指从实现角度来看，随着Java语言的发展，现在已经能看到些许迹象表明日后可能出现值类型的支持，即使只考虑现在，由于即时编译技术的进步，尤其是逃逸分析技术的日渐强大，栈上分配、标量替换优化手段已经导致一些微妙的变化悄然发生，所以说Java对象实例都分配在堆上也渐渐变得不是那么绝对了。

从回收内存的角度看，由于**现代垃圾收集器大部分都是基于分代收集理论设计**的，所以Java堆中经常会出现“新生代”“老年代”“永久代”“Eden空间”“From Survivor空间”“To Survivor空间”等名词

（以G1收集器的出现为分界），作为业界绝对主流的HotSpot虚拟机，它内部的垃圾收集器全部都基于“经典分代”来设计，需要新生代、老年代收集器搭配才能工作，在这种背景下，上述说法还算是不会产生太大歧义

###### E）方法区

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已**被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据**。虽然《Java虚拟机规范》中把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫作“非堆”（Non-Heap），目的是与Java堆区分开来。

在JDK 6的时候HotSpot开发团队就有放弃永久代，逐步改为采用本地内存（Native Memory）来实现方法区的计划了，到了**JDK 7的HotSpot**，已经把原本放在永久代的**字符串常量池、静态变量等移出**，而**到了JDK 8，终于完全废弃了永久代的概念**，改用与JRockit、J9一样在本地内存中实现的**元空间**（Meta-space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

###### F）运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class文件中除了有**类的版本、字段、方法、接口等描述信息**外，还有一项信息是**常量池表**（Constant Pool Table），用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

###### G）直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域。但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现，所以我们放到这里一起讲解。显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括物理内存、SWAP分区或者分页文件）大小以及处理器寻址空间的限制，一般服务器管理员配置虚拟机参数时，会根据实际内存去设置-Xmx等参数信息，但经常忽略掉直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常。

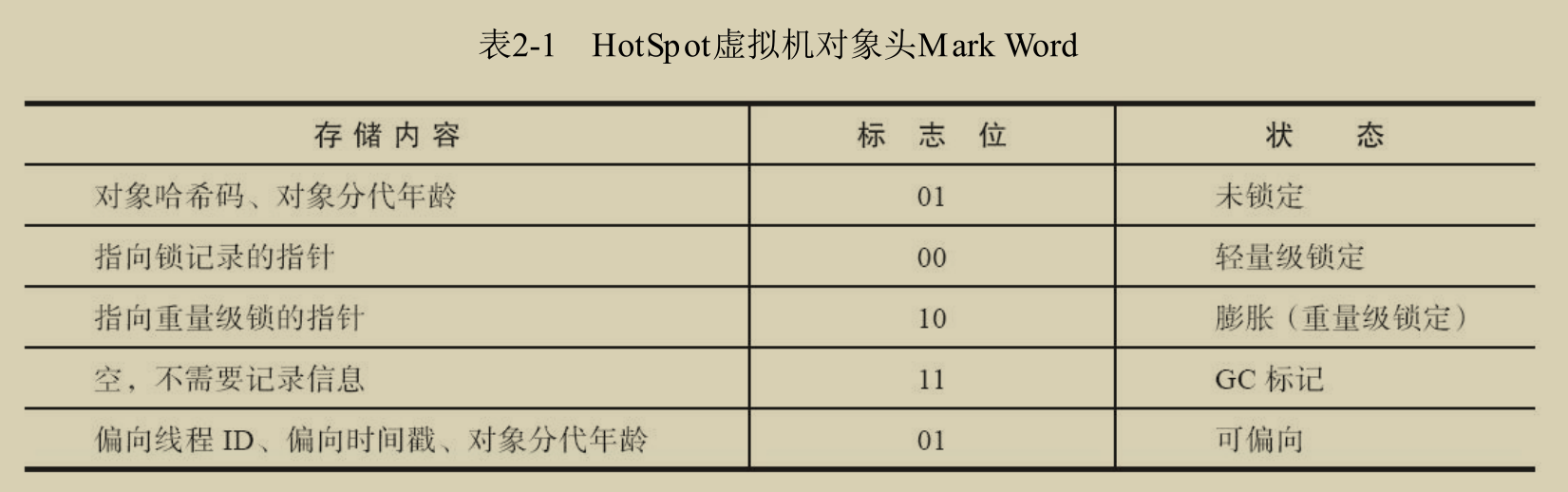
##### 2）对象的内存布局

在HotSpot虚拟机里，对象在堆内存中的存储布局可以划分为三个部分：

对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

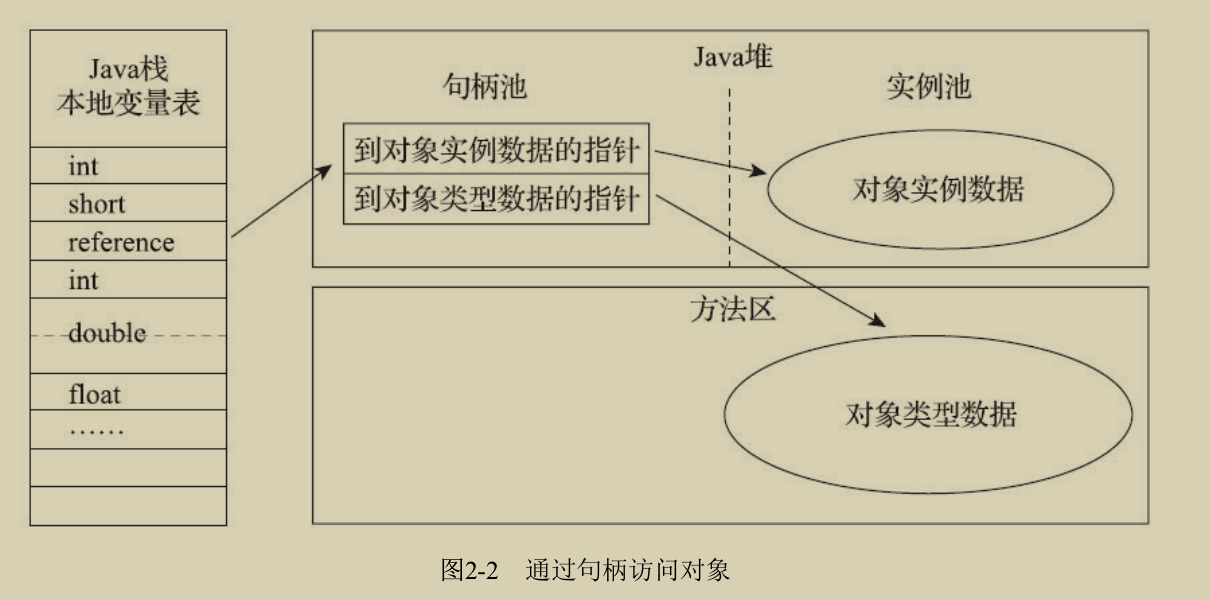
HotSpot虚拟机对象的对象头部分包括两类信息。

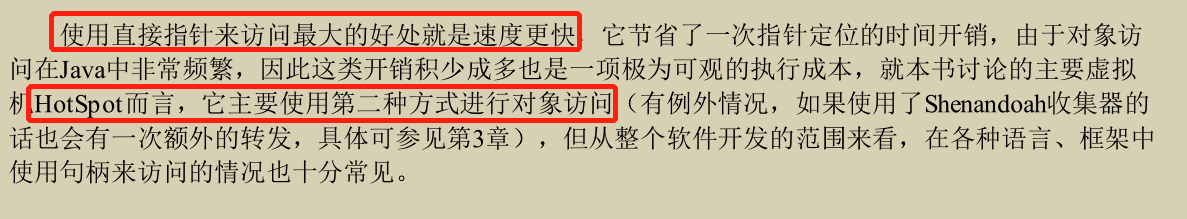
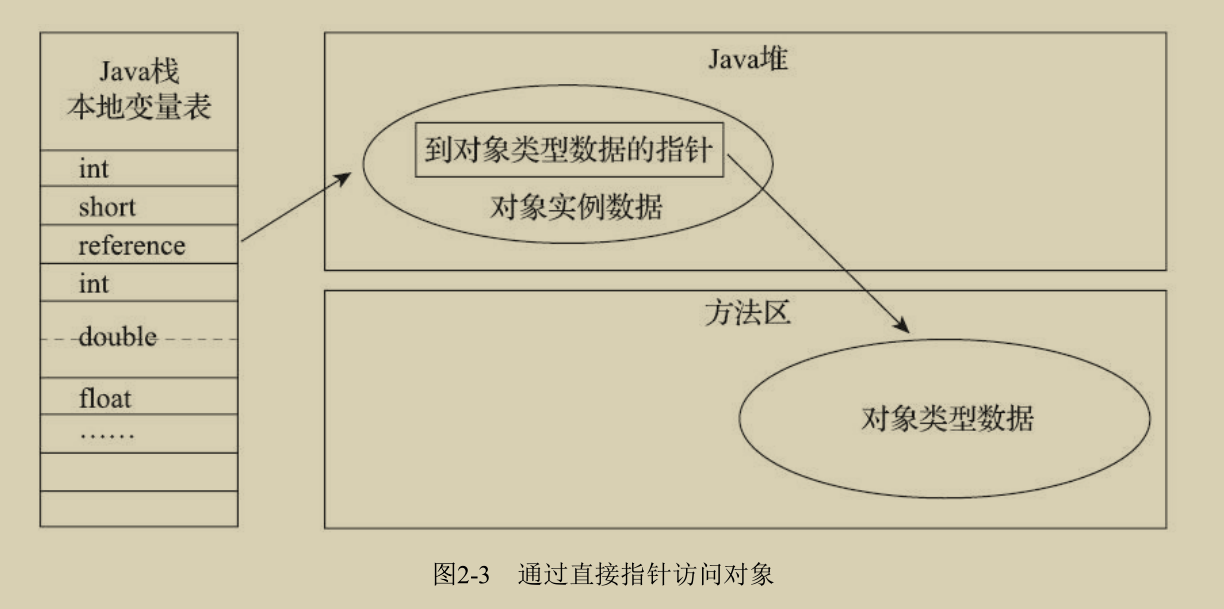
第一类是用于**存储对象自身的运行时数据**，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等



对象头的另外一部分是**类型指针**，即对象指向它的类型元数据的指针，Java虚拟机通过这个指针来确定该对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针

##### 3）对象的访问定位





##### ★★★★★4）OOM实战

###### HotSpot：参数

内存溢出：

-Xms 初始堆大小 物理内存的1/64(<1GB) 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制.

-Xmx 最大堆大小 物理内存的1/4(<1GB) 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制

-Xmn 年轻代大小(1.4or lator)

注意：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。

整个堆大小=年轻代大小 + 年老代大小 + 持久代大小.增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8

栈溢出：

·-XX：MaxMetaspaceSize：设置元空间最大值，默认是-1，即不限制，或者说只受限于本地内存大小。

·-XX：MetaspaceSize：指定元空间的初始空间大小，以字节为单位，**达到该值就会触发垃圾收集进行类型卸载**，同时收集器会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过-XX：MaxMetaspaceSize（如果设置了的话）的情况下，适当提高该值。

·-XX：MinMetaspaceFreeRatio：作用是在垃圾收集之后控制最小的元空间剩余容量的百分比，可减少因为元空间不足导致的垃圾收集的频率。类似的还有-XX：Max-MetaspaceFreeRatio，用于控制最大的元空间剩余容量的百分比

#### 2、垃圾收集器与内存分配策略

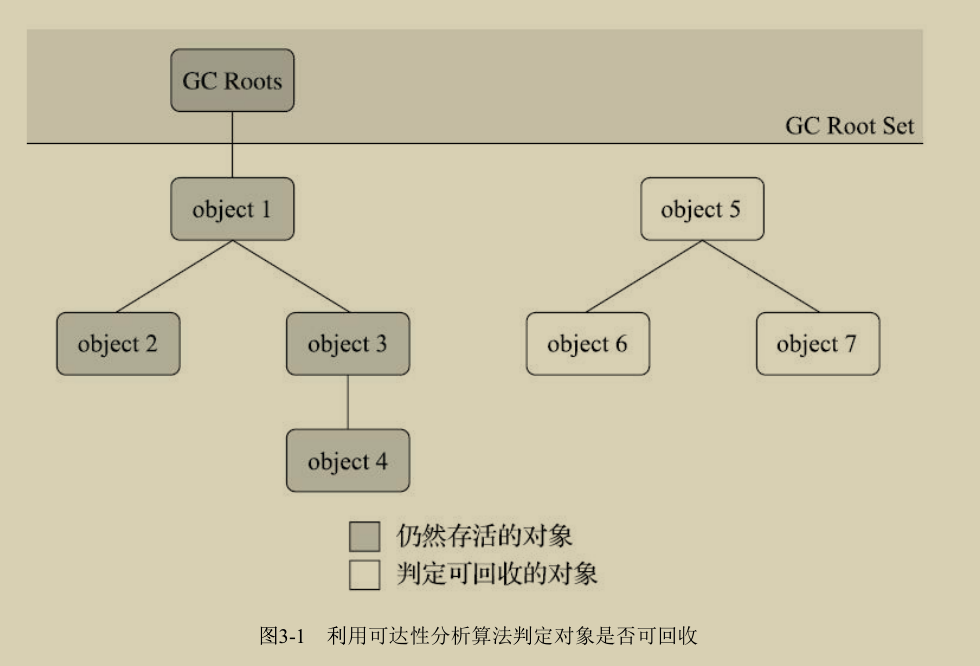
##### 1）判断对象是否已死

###### A）引用计数算法

在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可能再被使用的

###### B）可达性分析算法

基本思路就是通过一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”（Reference Chain），如果某个对象到GC Roots间没有任何引用链相连，或者用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的。



##### 2）垃圾收集算法

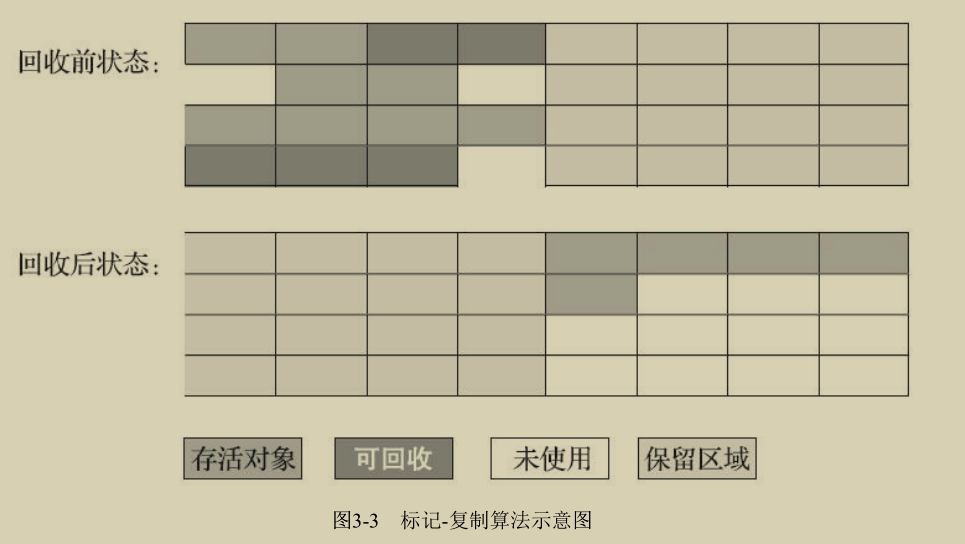
###### A）标记-清除算法

算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象



###### B）标记-复制算法

标记-复制算法常被简称为复制算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

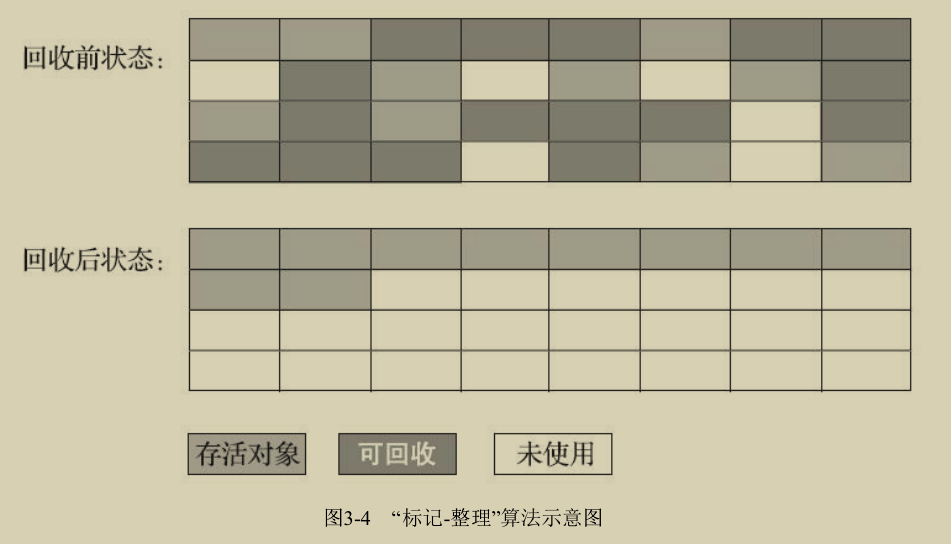


###### C）标记-整理算法

针对老年代对象的存亡特征，“标记-整理”（Mark-Compact）算法，其中的标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存

STW：

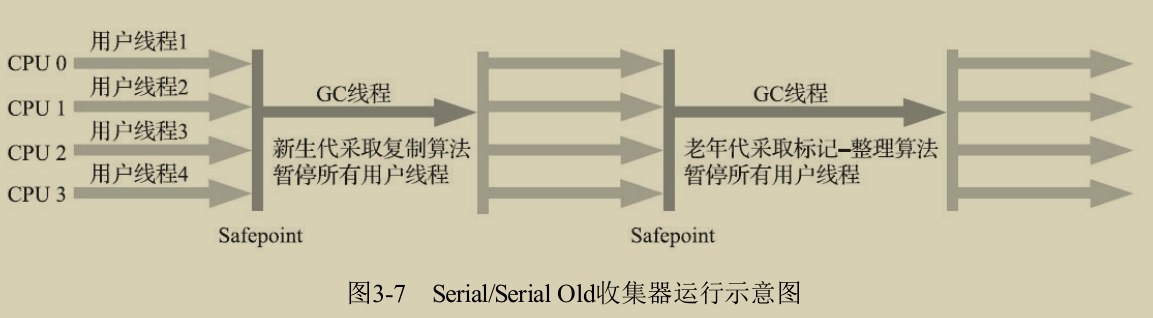
如果移动存活对象，尤其是在老年代这种每次回收都有大量对象存活区域，移动存活对象并更新所有引用这些对象的地方将会是一种极为负重的操作，而且这种对象移动操作必须全程暂停用户应用程序才能进行，像这样的停顿被最初的虚拟机设计者形象地描述为“Stop The World”



##### 3）经典垃圾收集器

###### A）Serial收集器

年轻代：单线程工作的收集器

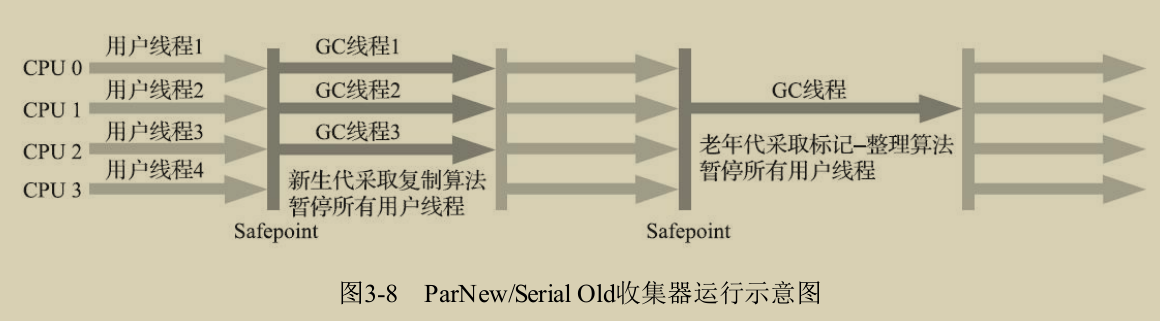


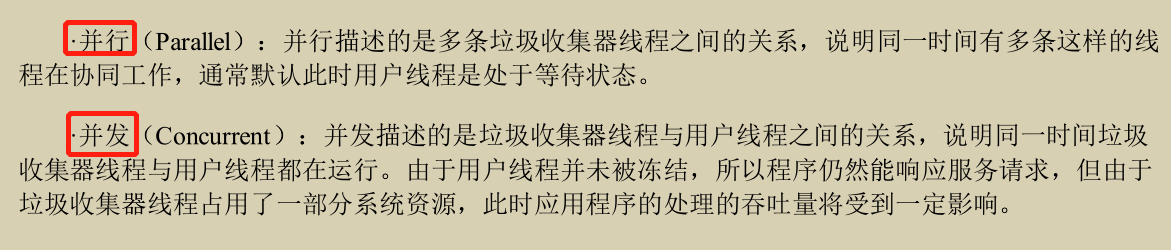
###### B）ParNew收集器

ParNew收集器实质上是Serial收集器的多线程并行版本

Serial收集器可用的所有控制参数

（例如：-XX：SurvivorRatio、-XX：PretenureSizeThreshold、-XX：HandlePromotionFailure等）

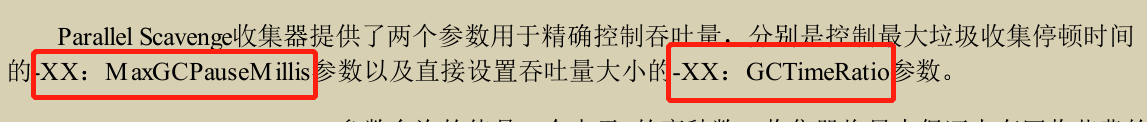
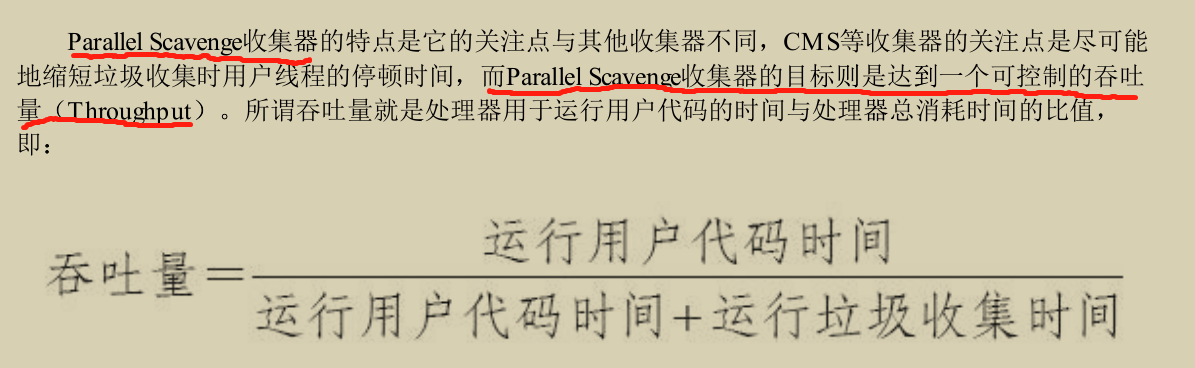




###### ★C）Parallel Scavenge收集器

Parallel Scavenge收集器也是一款新生代收集器，它同样是基于**标记-复制**算法实现的收集器，也是能够并行收集的多线程收集器

Parallel Scavenge收集器也经常被称作“吞吐量优先收集器”【关注吞吐量】



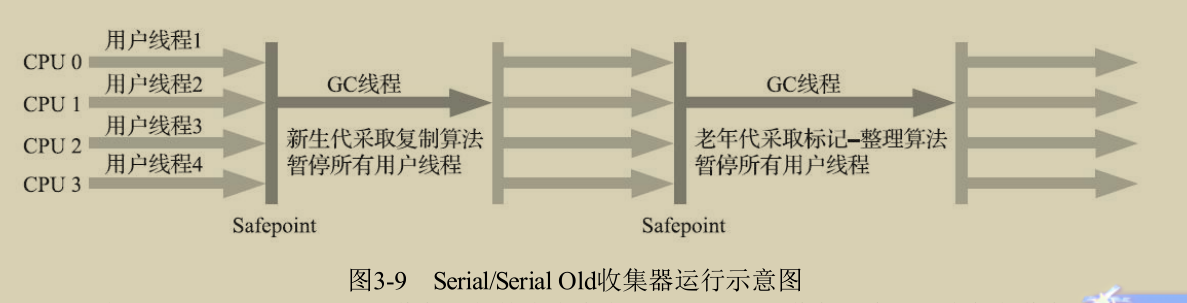
-XX：MaxGCPauseMillis

-XX：GCTimeRatio

-XX：+UseAdaptiveSizePolicy【动态调整参数以提供最合适的停顿时间或者最大的吞吐量】

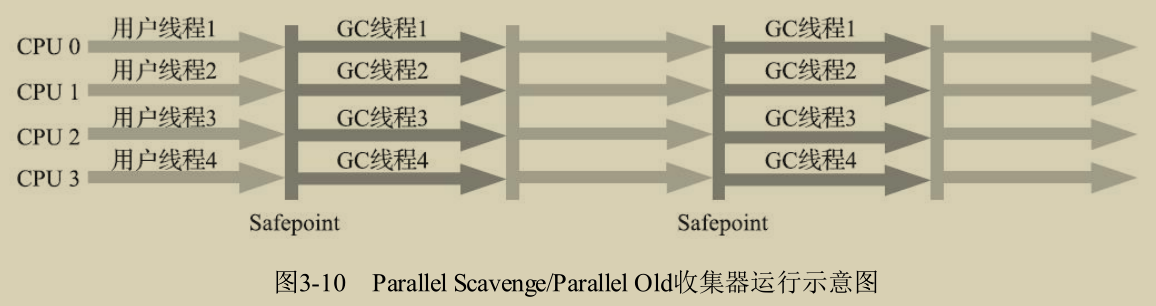
###### D）Serial Old收集器

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法



###### ★E）Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，支持**多线程并发收集**，基于**标记-整理**算法实现。这个收集器是直到JDK 6时才开始提供的



###### ★F）CMS收集器

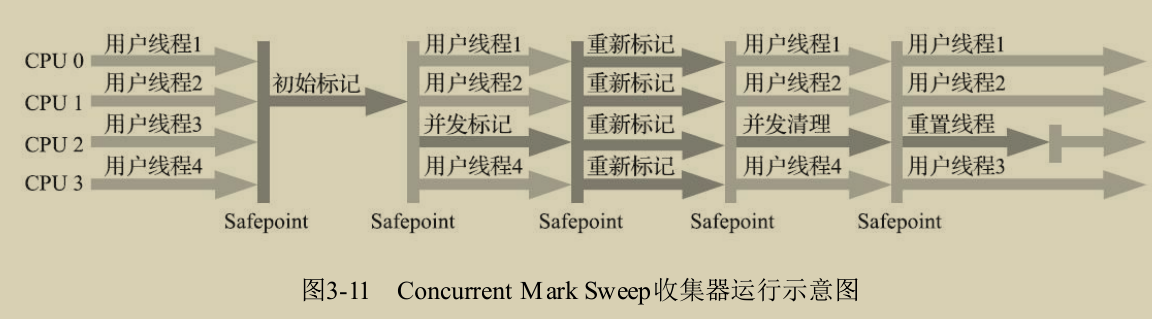
**CMS（Concurrent Mark Sweep）**收集器是一种以**获取最短回收停顿时间为目标的收集器**。目前很大一部分的Java应用集中在互联网网站或者基于浏览器的B/S系统的服务端上，这类应用通常都会较为关注服务的响应速度，希望系统停顿时间尽可能短，以给用户带来良好的交互体验

1）初始标记（CMS initial mark）

2）并发标记（CMS concurrent mark）

3）重新标记（CMS remark）

4）并发清除（CMS concurrent sweep）



CMS是一款优秀的收集器，它最主要的优点在名字上已经体现出来：并发收集、低停顿，一些官方公开文档里面也称之为“并发低停顿收集器”

三个明显的缺点：

**首先，CMS收集器对处理器资源非常敏感（过大过小影响吞吐量）**

**然后，CMS收集器无法处理“浮动垃圾”**

有可能出现“Con-current ModeFailure”失败进而导致另一次完全“Stop The World”的Full GC的产生。在CMS的并发标记和并发清理阶段，用户线程是还在继续运行的，程序在运行自然就还会伴随有新的垃圾对象不断产生，但这一部分垃圾对象是出现在标记过程结束以后，CMS无法在当次收集中处理掉它们，只好留待下一次垃圾收集时再清理掉。这一部分垃圾就称为“浮动垃圾”。

**最后，CMS会有大量空间碎片产生（标记清除）**

**-XX：CMSInitiatingOccu-pancyFraction【老年代回收阀值，默认68%】**

###### ★★★G）Garbage First收集器（G1收集器）

面向局部收集的设计思路和基于Region的内存布局形式

JDK 9发布之日，G1宣告取代Parallel Scavenge加Parallel Old组合，成为服务端模式下的默认垃圾收集器

G1跳出了这个樊笼，它可以面向堆内存任何部分来组成回收集（Collection Set，一般简称CSet）进行回收，**衡量标准不再是它属于哪个分代，而是哪块内存中存放的垃圾数量最多，回收收益最大**，这就是G1收集器的Mixed GC模式。

内存的布局与其他收集器有非常明显的差异：

G1不再坚持固定大小以及固定数量的分代区域划分，而是把连续的Java堆划分为多个**大小相等的独立区域（Region）**，每一个Region都可以根据需要，扮演新生代的Eden空间、Survivor空间，或者老年代空间。收集器能够对扮演不同角色的Region采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果

Region中还有一类特殊的Humongous区域，专门用来存储大对象

Region的大小可以通过参数**-XX：G1HeapRegionSize**设定，取值范围为1MB～32MB，且应为2的N次幂

每次根据用户设定允许的收集停顿时间（使用参数-XX：MaxGCPauseMillis指定，默认值是200毫秒），优先处理回收价值收益最大的那些Region，这也就是“Garbage First”名字的由来。

G1收集器至少有（不限于）以下这些关键的细节问题需要妥善解决：

**·譬如，Region里面存在的跨Region引用对象如何解决？**

使用记忆集避免全堆作为GC Roots扫描

**·譬如，在并发标记阶段如何保证收集线程与用户线程互不干扰地运行？**

G1收集器则是通过原始快照（SATB）算法来实现的，G1为每一个Region设计了两个名为TAMS（Top at Mark Start）的指针，把Region中的一部分空间划分出来用于并发回收过程中的新对象分配，并发回收时新分配的对象地址都必须要在这两个指针位置以上。如果内存回收的速度赶不上内存分配的速度，G1收集器也要被迫冻结用户线程执行，导致Full GC而产生长时间“Stop The World”。

**·譬如，怎样建立起可靠的停顿预测模型？**

**G1收集器的运作过程大致可划分为以下四个步骤：**

**·初始标记（Initial Marking）**

仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS指针的值

**·并发标记（Concurrent Marking）**

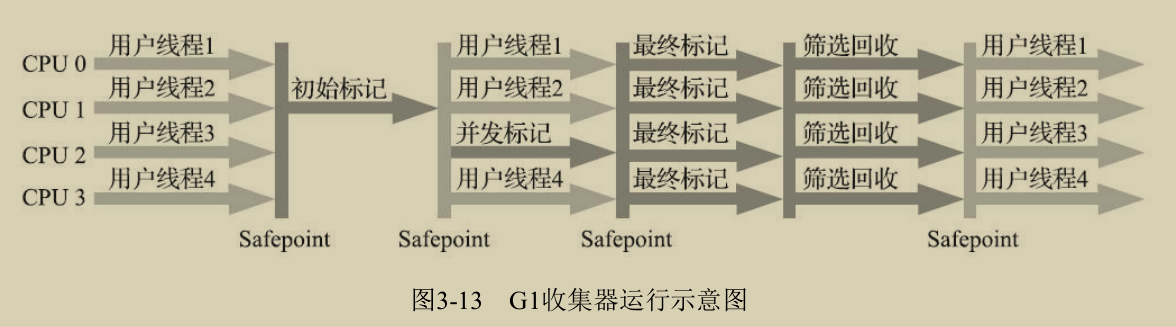
递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，还要重新处理SATB记录下的在并发时有引用变动的对象

**·最终标记（Final Marking）**

对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的SATB记录。

**·筛选回收（Live Data Counting and Evacuation）**

负责更新Region的统计数据，对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个Region构成回收集，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间



**可以由用户指定期望的停顿时间是G1收集器很强大的一个功能**

它默认的停顿目标为两百毫秒，一般来说，回收阶段占到几十到一百甚至接近两百毫秒都很正常（不能过高过低，过低则收集的速度跟不上分配的速度造成FullGC反而降低性能）

**G1对比CMS，并不是全方位压倒优势，还有下列不足：**

1）内存占用，G1卡表处理跨代引用指针，占用内存高

2）G1不仅有写后屏障，还有写前屏障，写前写后屏障放入队列，异步执行，写屏障的复杂性带来了额外的运算资源

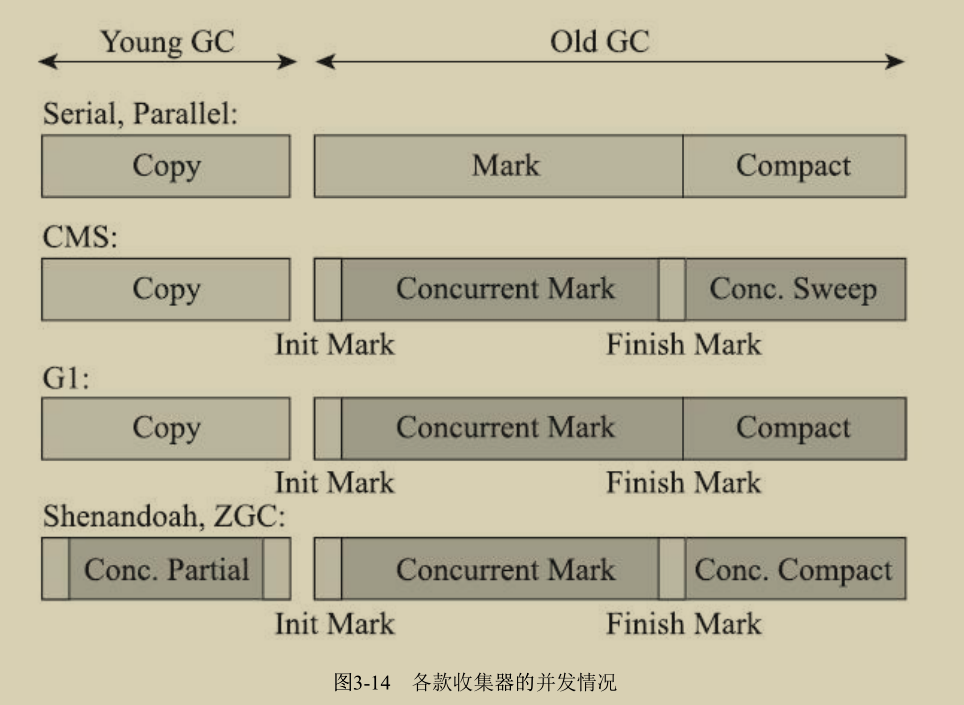
目前在小内存应用上CMS的表现大概率仍然要会优于G1，而在大内存应用上G1则大多能发挥其优势，这个优劣势的Java堆容量平衡点通常在6GB至8GB之间，当然，以上这些也仅是经验之谈

###### H）低延迟GC之Shenandoah收集器

衡量垃圾收集器的三项最重要的指标是：

**内存占用（Footprint）、吞吐量（Throughput）和延迟（Latency）**

在内存占用、吞吐量和延迟这三项指标里，**延迟**的重要性日益凸显，越发备受关注



Shenandoah作为第一款不由Oracle（包括以前的Sun）公司的虚拟机团队所领导开发的HotSpot垃圾收集器，不可避免地会受到一些来自“官方”的排挤。Oracle仍明确拒绝在OracleJDK 12中支持Shenandoah收集器，并执意在打包OracleJDK时通过条件编译完全排除掉了Shenandoah的代码，换句话说，Shenandoah是一款只有OpenJDK才会包含

Shenandoah：

这个项目的目标是实现一种能在**任何堆内存大小**下都可以把**垃圾收集的停顿时间限制在十毫秒以内**的垃圾收集器

Shenandoah新特点：

1）首先，最重要的当然是支持并发的整理算法

2）其次，Shenandoah（目前）是默认不使用分代收集的

3）最后，Shenandoah摒弃了在G1中耗费大量内存和计算资源去维护的记忆集，改用名为“**连接矩阵**”（ConnectionMatrix）的全局数据结构来**记录跨Region的引用**关系，降低了处理跨代指针时的记忆集维护消耗，也降低了伪共享问题（见3.4.4节）的发生概率。

###### ★★★I）低延迟GC之ZGC收集器

**ZGC**（“Z”并非什么专业名词的缩写，这款收集器的名字就叫作Z Garbage Collector）是一款在JDK 11中新加入的具有实验性质的**低延迟垃圾收集器**，是由Oracle公司研发的。

ZGC和Shenandoah的目标是高度相似的，都希望在尽可能对吞吐量影响不太大的前提下，实现在任意堆内存大小下都可以把垃圾收集的停顿时间限制**在十毫秒以内的低延迟**。

主要特征：

ZGC收集器是一款**基于Region内存布局**的，（暂时）**不设分代**的，使用了**读屏障、染色指针和内存多重映射等技术来实现可并发的标记-整理算法**的，以**低延迟为首要目标**的一款垃圾收集器

ZGC的Region可以具有大、中、小三类容量：

·小型Region（Small Region）：容量固定为2MB，用于放置小于256KB的小对象。

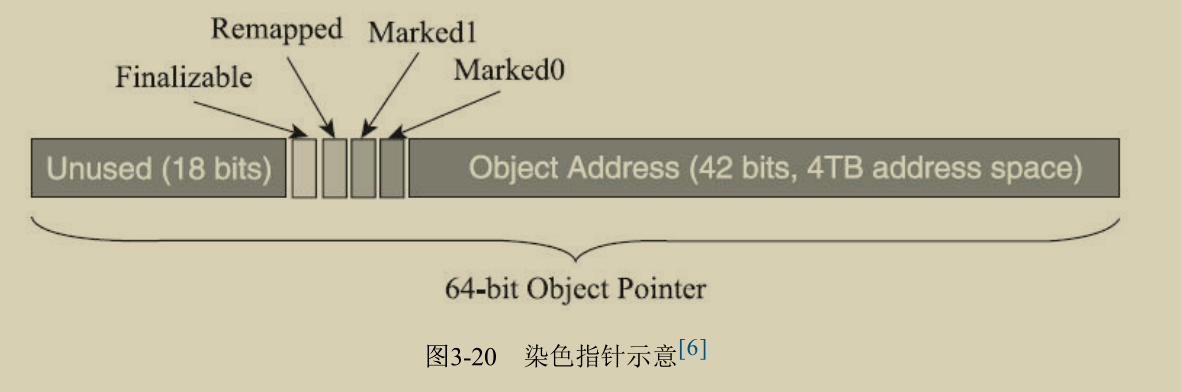
·中型Region（Medium Region）：容量固定为32MB，用于放置大于等于256KB但小于4MB的对象。

·大型Region（Large Region）：容量不固定，可以动态变化，但必须为2MB的整数倍，用于放置4MB或以上的大对象。

**ZGC的核心问题——并发整理算法的实现**

1）ZGC收集器有一个标志性的设计是它采用的染色指针技术

染色指针是一种直接将少量额外的信息存储在指针上的技术



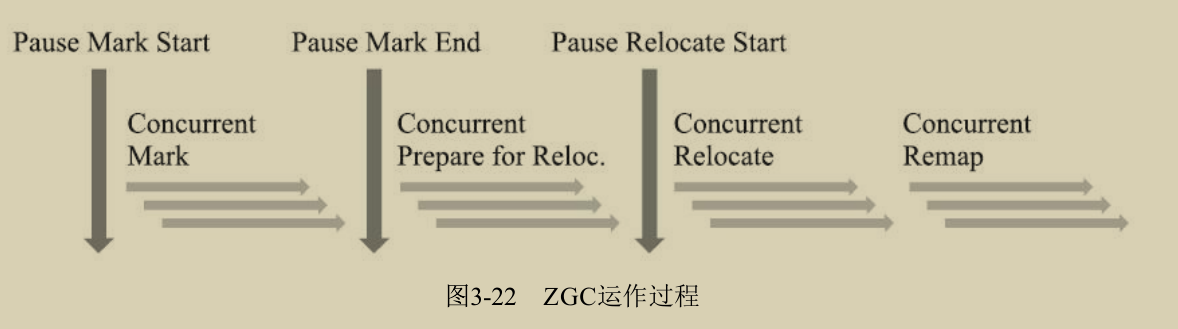
虽然染色指针有4TB的内存限制，不能支持32位平台，不能支持压缩指针（-XX：+UseCompressedOops）等诸多约束，但它带来的收益也是非常可观的

染色指针的三大优势：

**1）·染色指针可以使得一旦某个Region的存活对象被移走之后，这个Region立即就能够被释放和重用掉，**而不必等待整个堆中所有指向该Region的引用都被修正后才能清理。

**2）·染色指针可以大幅减少在垃圾收集过程中内存屏障的使用数量**，设置内存屏障，尤其是写屏障的目的通常是为了记录对象引用的变动情况，如果将这些信息直接维护在指针中，显然就可以省去一些专门的记录操作。

**3）·染色指针可以作为一种可扩展的存储结构用来记录更多与对象标记、重定位过程相关的数据**，以便日后进一步提高性能。



**·并发标记（Concurrent Mark）**

与G1、Shenandoah一样，并发标记是遍历对象图做可达性分析的阶段，前后也要经过类似于G1、Shenandoah的初始标记、最终标记（尽管ZGC中的名字不叫这些）的短暂停顿ZGC的标记是在指针上而不是在对象上进行的，标记阶段会更新染色指针中的Marked 0、Marked 1标志位。

**·并发预备重分配（Concurrent Prepare for Relocate）**

这个阶段需要根据特定的查询条件统计得出本次收集过程要清理哪些Region，将这些Region组成重分配集（Relocation Set）。

**·并发重分配（Concurrent Relocate）**

重分配是ZGC执行过程中的核心阶段，这个过程要把重分配集中的存活对象复制到新的Region上，并为重分配集中的每个Region维护一个转发表（Forward Table），记录从旧对象到新对象的转向关系。

**·并发重映射（Concurrent Remap）**

重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用

测试性能：

1）ZGC的“弱项”吞吐量方面，以低延迟为首要目标的ZGC已经达到了以高吞吐量为目标Parallel Scavenge的99%，直接超越了G1。

2）而在ZGC的强项停顿时间测试上，它就毫不留情地与Parallel Scavenge、G1拉开了两个数量级的差距。不论是平均停顿，还是95%停顿、99%停顿、99.9%停顿，抑或是最大停顿时间，ZGC均能毫不费劲地控制在十毫秒之内

笔者相信它完全成熟后，将会成为**服务端、大内存、低延迟应用**的首选收集器的有力竞争者

##### 4）垃圾收集器的选择

垃圾收集器的选择

1、应用程序关注点是什么，吞吐量，停顿时间延迟？

2、基础设施，系统架构，处理器个数/核数，分配内存，操作系统

3、Jdk版本，发行商

##### 5）常用垃圾收集器参数

-XX:+UseParallelGC

选择垃圾收集器为并行收集器

-XX:+UseParallelOldGC

年老代垃圾收集方式为并行收集

-XX: MaxTenuringThredShould

XX:CMSInitiatingOccupancyFraction=70

CMS老年代70%开始触发收集（默认68%）

-XX:MaxGCPauseMillis

最大GC停顿时间

-XX:GCTimeRatio

设置垃圾回收时间占程序运行时间的百分比

-XX:UseG1GC

使用G1收集器

G1HeapRegionSize

设置Region大小

G1NewSizePrecent

新生代最小值，默认5%

G1MaxNewSizePrecent

新生代最大值，默认60%

补充：

老年代分配担保

eden新生代进行minor gc后，极端情况都存活，这是需要老年代内存分配担保，前提老年代有空间，晋升老年代依据历次晋升的平均值，若老年代剩余空间大于平均值，则晋升，否则full gc