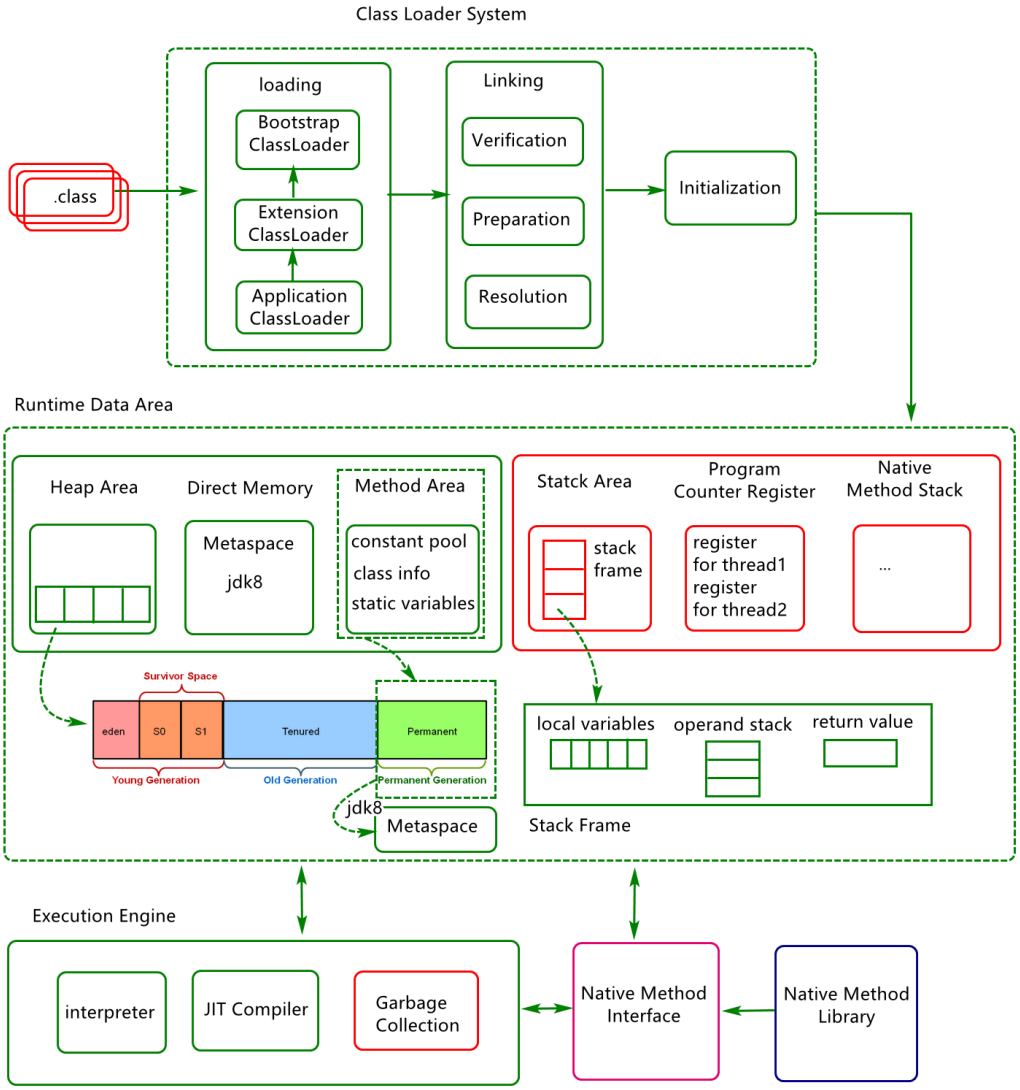
JVM
GC

1. GC 简介

1.1. 引言

在理解 GC 之前，先回顾一下 JVM 体系结构，如下图所示:



1.2. 何为 GC

基于正在使用的对象进行遍历，对存活的对象进行标记，其未标记对象可认为是垃圾对象，然后基于特定算法进行回收，这个过程称之为 GC (Garbage Collection)。所有的 GC 系统可从如下几个方面进行实现：

1.GC 判断策略（例如引用计数，对象可达性分析）
2.GC 收集算法（标记-清除，标记-清除-整理，标记-复制-清除）
3.GC 收集器(例如 Serial,Parallel,CMS,G1)

为何要学习 GC 呢?

深入理解 GC 的工作机制，可以帮你写出更好的 Java 应用，同时也是进军大规模应用开发的一个前提。

1.2.1. 手动 GC

手动 GC 即显式地进行内存分配(allocate)和内存释放(free)。如果忘记释放，则对应的那块内存不能再次使用。内存一直被占着，却不能使用，这种情况就称为内存泄漏(memory leak)。

这是一个使用手动内存管理用 C 编写的简单示例：

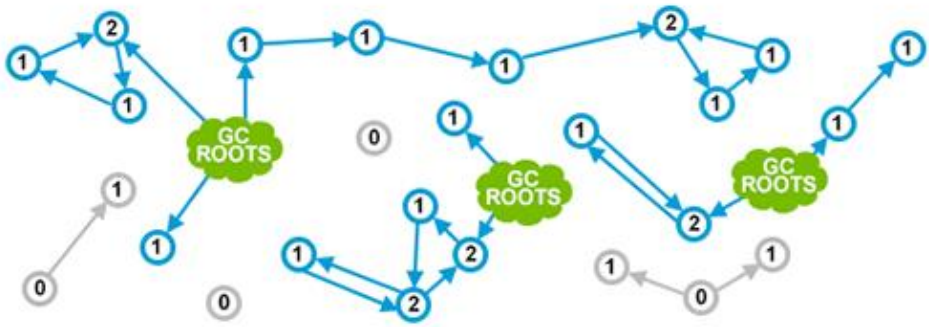
```
int send_request() {
    size_t n = read_size();
    int *elements = malloc(n * sizeof(int));

    if(read_elements(n, elements) < n) {
        // elements not freed!
        return -1;
    }
    // …
    free(elements)
    return 0;
}
```

手动 GC 时忘记释放内存是相当容易的。这样会直接导致内存泄漏。

1.2.2. 自动 GC

自动 GC 一般是在 JVM 系统内存不足时，由 JVM 系统启动 GC 对象，自动对内存进行垃圾回收。

引用计数法



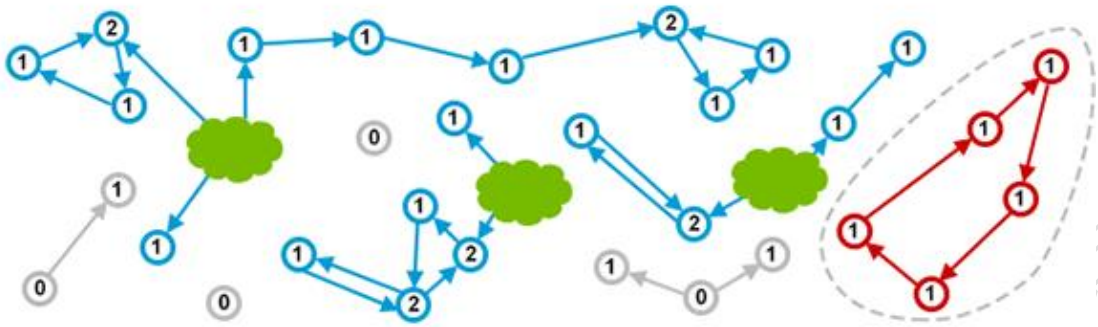
其中：

1） 绿色云朵是内存中的根对象，表示程序中正在使用的对象。

2） 蓝色圆圈是内存中的活动对象，其中的数字表示其引用计数。

3） 灰色圆圈是内存中没有活动对象引用的对象，表示非活动对象。

对于引用计数法，有一个很大的缺陷就是循环引用，例如：



其中红色对象实际上是应用程序不使用的垃圾。但由于引用计数的限制，仍然存在内存泄漏。当然也有一些办法来应来对这种情况，例如 “弱引用”（‘weak’ references)或者使用其它的算法来排查循环引用等。

标记清除

标记清除就是对可达对象进行标记，不可达对象即认为是垃圾，然后进行清除。标记清除通常有两个步骤：

1)标记所有可到达的对象

2)清除不可到达对象占用的内存地址。

此方法解决了循环依赖问题，但存在短时间的线程暂停,我们一般称这种现象为STW 停顿(Stop The World pause，全线暂停)。

说明:

JVM 中包含了多种 GC 算法收集器,他们虽在实现上略有不同， 但理论上都采用了以上两个步骤。

2.GC 入门分析

2.1. 碎片整理

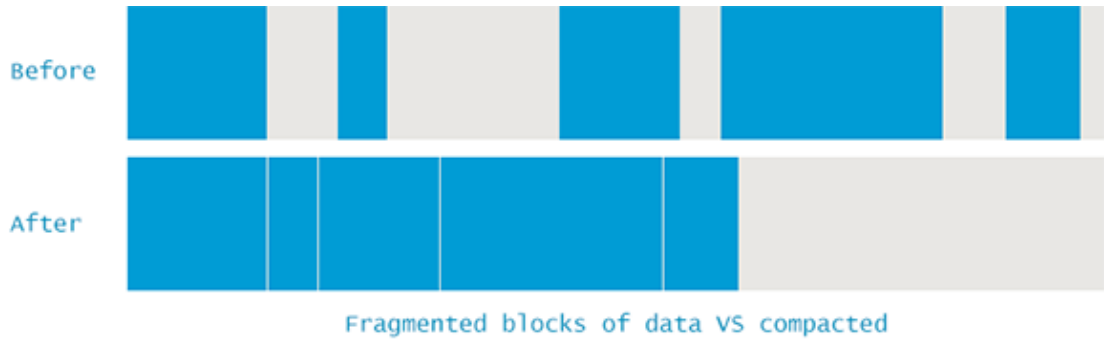
系统 GC 时每次执行清除(sweeping)操作， JVM 都必须保证“不可达对象“占用的内存能被回收然后重用。内存是被回收了，但这有可能会产生大量的内存碎片(类似于磁盘碎片)，进而引发两个问题:

1)对象创建时，执行写入操作越来越耗时， 因为寻找一块足够大的空闲内存会变得更加麻烦。

2)对象创建时， JVM 需要在连续的内存块中为对象分配内存。如果碎片问题很严重， 直至没有空闲片段能存放新创建的对象,就会发生内存分配错误(allocation error)。

为了解决碎片问题，JVM 在启动 GC 执行垃圾收集的过程中，不仅仅是标记和清除， 还需要执行 “内存碎片整理”。这个过程会让所有可达对象(reachable

objects)进行依次移动,进而可以消除(或减少)内存碎片,并为新对象提供更大并且连续的内存空间。示意图如下



说明：内存整理时会将对象移动到靠近内存地址的起始位置。

2.2. 分代设想

我们知道垃圾收集要停止整个应用程序的运行,那么假如这个收集过程需要的时间很长，就会对应用程序产生很大性能问题，如何解决这个问题呢？通过实验发现内存中的对象通常可以将其分为两大类：

1）存活时间较短(这样的对象比较多)

2）存活时间较长(这样的对象比较少)

基于对如上问题的分析，科学家提出了分代回收思路，将 VM 中内存分为年轻代(Young Generation)和老年代(Old Generation-老年代有时候也称为年老区(Tenured)。

分代设想将内存拆分为两个可单独清理的区域,允许采用不同的算法来大幅提高

GC 的性能。但这种方法也不是没有问题。例如，在不同分代中的对象可能会互相引用，这样的对象就难以回收。

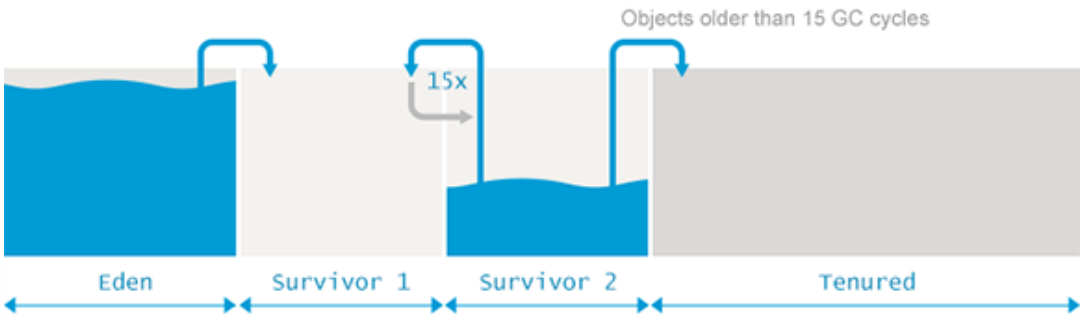
2.3. 对象分配

JAVA 中堆内存的内存结构如下图所示：

基于此内存架构，对象内存分配过程如下：

1）编译器通过逃逸分析（JDK8 已默认开启），确定对象是在栈上分配还是在堆上分配。
2) 如果是在堆上分配，则首先检测是否可在 TLAB (Thread Local Allocation Buffer) 上直接分配。
3）如果 TLAB 上无法直接分配则在 Eden 加锁区进行分配(线程共享区)。
4）如果 Eden 区无法存储对象，则执行 Yong GC（Minor Collection）
5）如果 Yong GC 之后 Eden 区仍然不足以存储对象，则直接分配在老年代。

说明：在对象创建时可能会触发 Yong GC，此 GC 过程的简易原理图分析如下：



其中：

1)新生代由 Eden 区和两个幸存区构成(假定为 s1,s2)，任意时刻至少有一个幸存区是空的(empty)，用于存放下次 GC 时未被收集的对象。
2)GC 触发时 Eden 区所有”可达对象”会被复制到一个幸存区，假设为 s1，当幸存区 s1 无法存储这些对象时会直接复制到老年代。
3)GC 再次触发时 Eden 区和 s1 幸存区中的”可达对象”会被复制到另一个幸存区 s2，同时清空 eden 区和 s1 幸存区.
4)GC 再次触发时 Eden 区和 s2 幸存区中的”可达对象”会被复制到另一个幸存区 s1，同时清空 eden 区和 s2 幸存区.依次类推。
5)当多次 GC 过程完成后，幸存区中的对象存活时间达到了一定阀值（可以用参数 -XX:+MaxTenuringThreshold 来指定上限，默认 15），会被看成是“年老”的对象然后直接移动到老年代。

2.4. GC 模式分析

垃圾收集事件(Garbage Collection events)通常分为:
1) Minor GC （小型 GC）：年轻代 GC 事件，(新对象)分配频率越高，Minor GC 的频率就越高
2) Major GC （大型 GC)：老年代 GC 事件
3) Full GC （完全 GC）:整个堆的 GC 事件

说明：一般情况下可以将 Major GC 与 Full GC 看成是同一种 GC。

3.GC 算法基础

3.1. 标记可达对象

现在的 GC 算法，基本都是要从标记”可达对象”开始(Marking Reachable Objects),这些标记为可达的对象即为存活对象。同时我们可以将查找可达对象

时的起始位置对象，认为是根对象（**Garbage Collection Roots**）。基于根对象标记可访问或可达对象，对于不可达对象，GC 会认为是垃圾对象.例如：在下面图示中的绿色云朵为根对象，蓝色圆圈为可达对象，灰色圆圈为垃圾对象。

首先，GC 遍历(traverses)内存中整体的对象关系图(object graph)确定根对象,那什么样的对象可作为根对象呢？GC 规范中指出根对象可以是：

1）栈中变量直接引用的对象

2）常量池中引用的对象

3）…

其次，确定了根对象以后，进而从根对象开始进行依赖查找，所有可访问到的对象都认为是存活对象，然后进行标记（mark）。

说明：标记可达对象需要暂停所有应用线程，以确定对象的引用关系。其暂停的时间， 与堆内存大小、对象的总数没有直接关系， 而是由存活对象(alive objects)的数量来决定。

3.2. 移除不可达对象

移除不可达对象（Removing Unused Objects）时会因 GC 算法的不同而不同，但是大部分的 GC 操作一般都可大致分为三类：清除(Mark-Sweep)，标记整理清除(Mark-Sweep-Compact)，标记复制(Mark-Copy).

标记-清除（Mark-Sweep）

对于标记清除算法(Mark and Sweep algorithms)应用相对简单，但内存会产

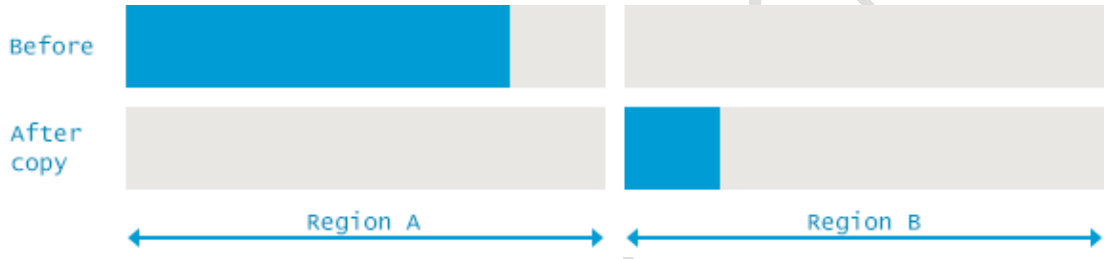
生大量的碎片，这样再创建大对象时，假如内存没有足够连续的内存空间可能会出现 OutOfMemoryError。

标记-清除-整理（Mark-Sweep-Compact）

标记清除整理算法中在清除垃圾对象以后会移动可用对象，对碎片进行压缩，这样会在内存中构建相对比较大的连续空间便于大对象的直接存储，但是会增加 GC 暂停时间。

标记-复制（Mark and Copy）

标记复制算法会基于标记清除整理算法，但是会创建新的内存空间用于存储幸存对象，同时复制与标记可以同时并发执行，这样可以较少 GC 时系统的暂停时间，提高系统性能。



4.GC 算法实现

4.1. GC 算法实现简介

我们知道，JVM 系统在运行时，因新对象的创建，可能会触发 GC 事件。无论哪种 GC 都可能会暂停应用程序的执行，但如何将暂停时间降到最小，这要看我们使用的 GC 算法。现在对于 JVM 中的 GC 算法无非两大类：一类负责收集年轻代，一类负责收集老年代。假如没有显式指定垃圾回收算法，一般会采用系统平台默认算法，当然也可以自己指定，例如 JDK8 中基于特定垃圾回收算法的垃圾收集器应用组合如下：

Young	Tenured	JVM options
Serial	Serial	-XX:+UseSerialGC
Parallel Scavenge	Parallel Old	-XX:+UseParallelGC -XX:+UseParallelOldGC
Parallel New	CMS	-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
G1		-XX:+UseG1GC

其中：

1）年轻代和老年代的串行收集器： Serial GC)

2）年轻代和老年代的并行收集器：(Parallel GC)

3）年轻代的并行收集器 (Parallel New) + 老年代的并发收集器 (CMS-Concurrent Mark and Sweep)

4）年轻代和老年代的 G1 收集器，负责回收年轻代和老年代

说明：除了以上几种组合方式外，其它的组合方式要么现在已经不支持，要么不推荐。如何对这些组合进行选择，要结合系统的特点。例如系统是追求高吞吐量还是响应时间，还是两者都要兼顾。总之，对于 GC 组合的选择没有最好，只有更好。知己知彼，才能百战不殆。结合当前系统的环境配置，性能指标以及 GC 器特点，不断进行 GC 日志分析，定位系统问题，才是一般是选择哪种 GC 的关

键。

4.2. GC 收集器应用分析

4.2.1. Serial 收集器应用分析

Serial GC 是最古老也是最基本的收集器，但是现在依然广泛使用，JAVA SE5 和 JAVA SE6 中客户端虚拟机采用的默认配置。

Serial GC（串行收集器）应用特点:

1）内部只使用一个线程执行垃圾回收(不能充分利用 CPU 的多核特性),无法并行化。
2）GC 时所有正在执行的用户线程暂停并且可能会产生较长时间的停顿(Stop the world)

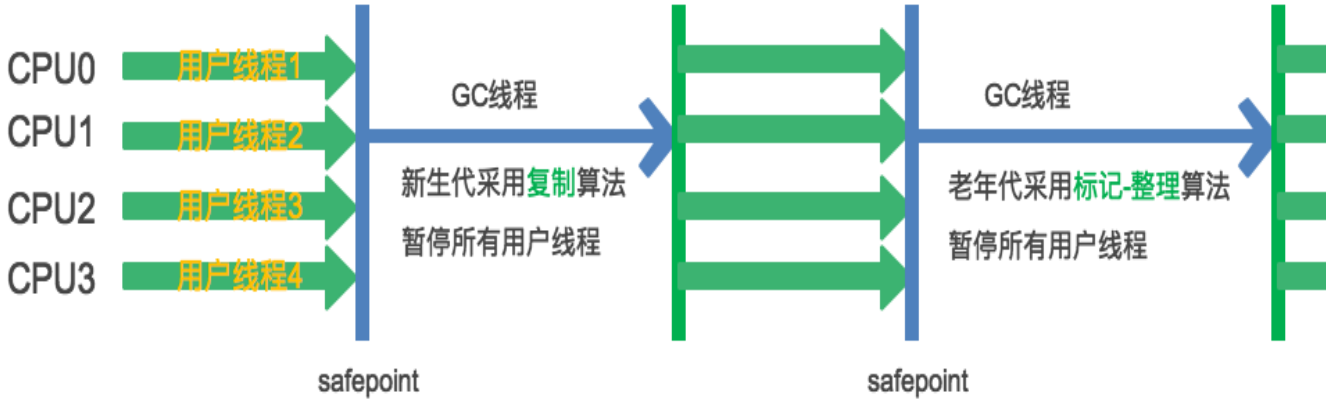
Serial GC（串行收集器）场景应用:

1）一般可工作在 JVM 的客户端模式。
2）适用于 CPU 个数或核数较少且内存空间较小（越大可能停顿时间越长）的场景

Serial GC（串行收集器）算法应用

1）新生代使用 mark-copy(标记-复制）算法(新生代存活对象较少)
2）老年代使用 mark-sweep-compact(标记-清除-整理)算法(老年代对象回收较少，容易产生碎片)

Serial GC（串行收集器）实践应用



其应用参数配置：-XX:+UseSerialGC

总之，Serial GC 一个单线程的收集器，在进行垃圾收集时，必须暂停其他所有的工作线程。适合单 CPU 小应用，实时性要求不是那么高场景。一般在 JVM 的客户端模式下应用比较好。

4.2.2. Parallel 收集器应用分析

Parallel 收集器为并行收集器，它可利用多个或多核 CPU 优势实现多线程并行 GC 操作，其目标是减少停顿时间，实现更高的吞吐量（Throughput）。

Parallel GC（并行收集器）应用特点：

1）可利用 CPU 的多核特性执行多线程下的并行化 GC 操作。
2）GC 期间，所有 CPU 内核都在并行清理垃圾，所以暂停时间较短。
3）最大优势是可实现可控的吞吐量与停顿时间。

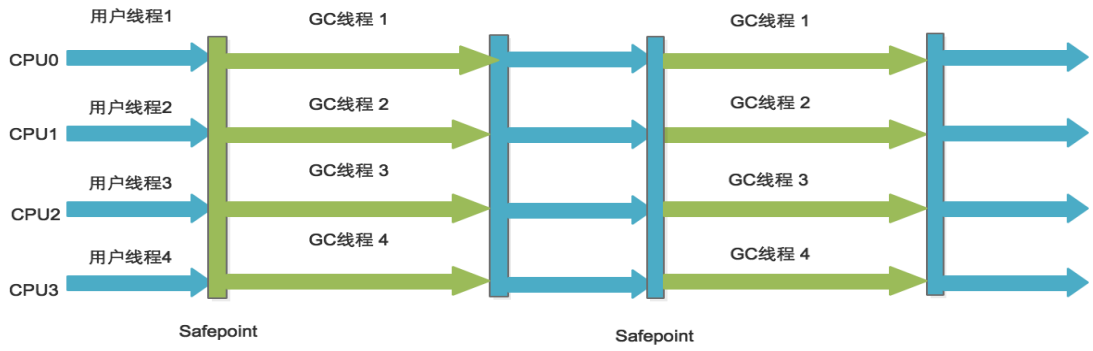
Parallel GC（并行收集器）场景应用：

1） GC 操作仍需暂停应用程序（也有可能暂停时间比较长，因为 GC 阶段不能被打断），所以不适合要求低延迟的场景。
2） 因其高吞吐 GC(throughput GC)量的特性，适用于后台计算、后台处理的弱交互场景而不是 web 交互场景

Parallel GC（并行收集器）算法应用：

1）在年轻代使用标记-复制(mark-copy)算法，对应的是 Parallel Scavenge 收集器
2）在老年代使用标记-清除-整理(mark-sweep-compact)算法，对应的是 Parallel Old 收集器。

Parallel GC（并行收集器）实践应用：



使用 Parallel GC 的参数配置：-XX:+UseParallelGC ，默认开启-XX:+UseParallelOldGC
其它参数配置：

1）-XX:ParallelGCThreads=20：设置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。
2）-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值。
3）-XX:+UseAdaptiveSizePolicy 设置并行收集器自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低响应时间或者收集频率等，此值建议使用并行收集器时，一直打开。
4）-XX:GCTimeRatio=99，设置吞吐量大小，默认值就是 99，也就是将垃圾回收的时间设置成了总时间的 1%。它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集。

总之，Parallel GC 是一种并行收集器，可利用多 CPU 优势，执行并行 GC 操作，吞吐量较高，并可有效降低工作线程暂停时长。但是因为垃圾收集的所有阶

段都不能被打断，所以 Parallel GC 还是有可能导致长时间的应用暂停。所以 Parallel GC 适合于需要高吞吐量而对暂停时间不敏感的场合，比如批处理任务。

说明：
所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量 = 运行用户代码时间 / （运行用户代码时间 + 垃圾收集时间）

4.2.3. CMS 收集器应用分析

CMS 的官方名称为 “Mostly Concurrent Mark and Sweep Garbage Collector”，其设计目标是追求更快的响应时间。

CMS （并发收集器)应用特点：

1） 使用空闲列表(free-lists)管理内存空间的回收，不对老年代进行碎片整理，减少用户线程暂停时间。
2） 在标记-清除阶段的大部分工作和用户线程一起并发执行。
3） 最大优点是可减少停顿时间（可提高服务的响应速度），最大缺陷是老年代的内存碎片

CMS （并发收集器)场景应用：

1） 应用于多个或多核处理器，目标降低延迟，缩短停顿时间，响应时间优先.
2） CPU 受限场景下，因与用户线程竞争 CPU，吞吐量会减少。

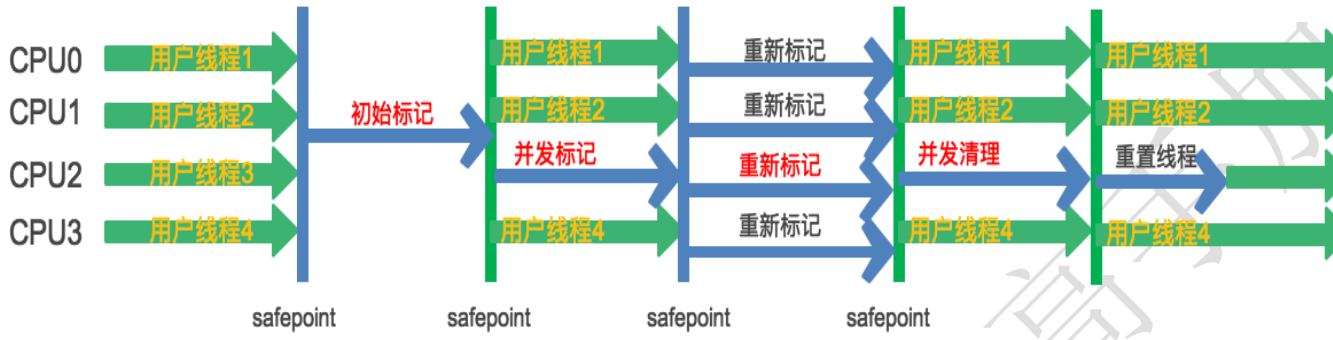
CMS （并发收集器)算法应用：

1） 年轻代采用并行方式的 mark-copy （标记-复制)算法。
2） 老年代主要使用并发 mark-sweep （标记-清除)算法。

CMS （并发收集器)关键步骤分析：

1）初始标记（initial mark）此阶段标记一下 GC Roots 能直接关联到的对象，速度很快。
2）并发标记（concurrent mark）此阶段就是进行 GC Roots Tracing 的过程，从直接关联对象遍历所有可达对象，然后进行标记。
3）重新标记（final remark）此阶段要修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录
4）并发清除（concurrent sweep）此阶段与应用程序并发执行，不需要 STW 停顿。目的是删除未使用的对象，并收回他们占用的空间。
5）并发重置（concurrent Reset）此阶段与应用程序并发执行，重置 CMS 算法相关的内部数据，同时 GC 线程切换到用户线程。

CMS（并发收集器）实践应用：



使用 CMS 的参数配置：-XX:+UseConcMarkSweepGC，默认开启 -XX:+UseParNewGC

其它参数配置：

1）-XX:+ UseCMSCompactAtFullCollection 执行 Full GC 后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长
2）-XX:+CMSFullGCsBeforeCompaction 设置进行几次 Full GC 后，进行一次碎片整理
3）-XX:ParallelCMSThreads 设定 CMS 的线程数量（一般情况约等于可用 CPU 数量）

总之， CMS 垃圾收集器在减少停顿时间上做了很多给力的工作， 大量并发执行的工作并不需要暂停应用线程。如果服务器是多核 CPU，并且主要调优目标是降低延迟， 那么使用 CMS 是个很明智的选择。 CMS 垃圾收集可减少每一次 GC 停

顿的时间,这样会直接影响到终端用户对系统的体验，用户会认为系统非常灵敏。但是因为多数时候都有部分 CPU 资源被 GC 消耗， 所以在 CPU 资源受限的情况下,CMS 会比并行 GC 的吞吐量差一些。还有就是老年代内存碎片问题， 在某些情况下 GC 会造成不可预测的暂停时间，特别是堆内存较大的情况下。

4.2.4. G1 收集器应用分析

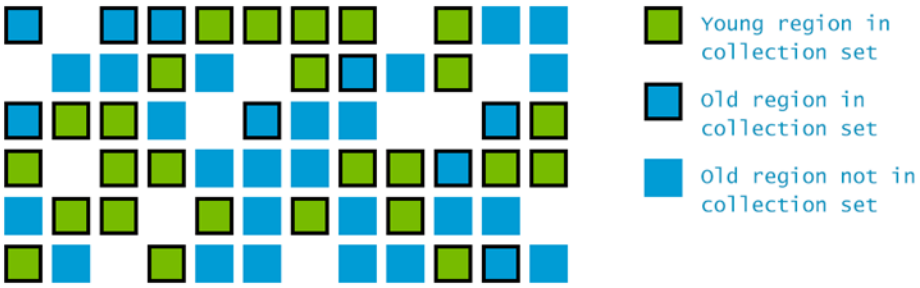
G1(Garbage-First)收集器是一种工作于服务端模式的垃圾回收器，主要面向多核，大内存的服务器。G1 在实现高吞吐的同时，也最大限度满足了 GC 停顿时间可控的目标。在 Oracle JDK7 update 4 后续的版本中已全面支持 G1 回收器功能。G1 收集器主要为有如下需求的程序设计：

1） 可以像 CMS 收集器一样能同时和应用线程一起并发的执行；
2） 减少整理内存空间时的停顿时间；
3） 要满足可预测的 GC 停顿时间需求；
4） 不能牺牲太多的吞吐性能；

未来 G1 计划要全面取代 CMS。G1 相比 CMS 有更多的优势，G1 是压缩型收集器，可以实现更有效的空间压缩，消除大部分潜在的内存碎片问题。G1 提供了更精准的可预测的垃圾停顿时间设置,可满足用户在指定垃圾回收时间上的需求。

在 G1 中,堆不再分成连续的年轻代和老年代空间,而是划分为多个(通常是 2048 个)可以存放对象的小堆区(smaller heap regions)。每个小堆区都可能是 Eden 区， Survivor 区或者 Old 区． 在逻辑上， 所有的 Eden 区和 Survivor 区合起来就是年轻代，所有的 Old 区拼在一起那就是老年代，如下图所示：

这样的划分使得 GC 不必每次都去收集整个堆空间， 而是以增量的方式来处理。GC 时每次只处理一部分小堆区,称为此次的回收集(collection set). GC 事件的每次暂停都会收集所有年轻代的小堆区， 同时也可能包含一部分老年代小堆区，如下图所示:



G1 在并发阶段估算每个小堆区存活对象的总数，垃圾最多的小堆区会被优先收集，这也是 G1 名称的由来。

G1 以一种和 CMS 相似的方式执行垃圾回收。G1 在并发标记阶段估算每个小堆区存活对象的总数，垃圾最多的小堆区会被优先收集，这也是 G1 名称的由来。顾名思义，G1 将其收集和压缩活动集中在堆中可能充满可回收对象（即垃圾）的区域上。G1 通过用停顿预测模型来满足用户自定义的停顿时间目标，它基于设定的停顿时间来选择要回收的 regions 数量。

G1 基于标记，清理对应的 regions 时,会将对象从一个或多个 region 里复制到另一个 region 里，在这个过程中会伴随着压缩和释放内存。清理过程在多核机器上都采用并行执行，来降低停顿时间，增加吞吐量。因此 G1 在持续的运行中能减少碎片，满足用户自定义停顿时间需求。这种能力是以往的回收器所不具备的（例如 CMS 回收器不能进行碎片压缩，ParallelOld 只能进行整堆的压缩，会导致较长的停顿时间）。

再次强调：G1 不是一个实时的收集器，它只是最大可能的来满足设定的停顿时间。G1 会基于以往的收集数据，来评估用户指定的停顿时间可以回收多少 regions，需要花费的时间，然后确定停顿时间内可以回收多少个 regions。

G1 收集器特点:

1）将 java 堆均分成大小相同的多个区域（region，1M-32M，最多 2000 个，最大支持堆内存 64G）。

2）内存应用具备极大地弹性(一个或多个不连续的区域共同组成 eden、survivor 或 old 区，但大小不再固定。)
3）相对 CMS 有着更加可控的暂停时间(pause time) 和 更大的吞吐量(throughput)以及更少的碎片（标记整理）
4）支持并行与并发，可充分利用多 CPU，多核优势，降低延迟，提高响应速度。

G1 场景应用分析：

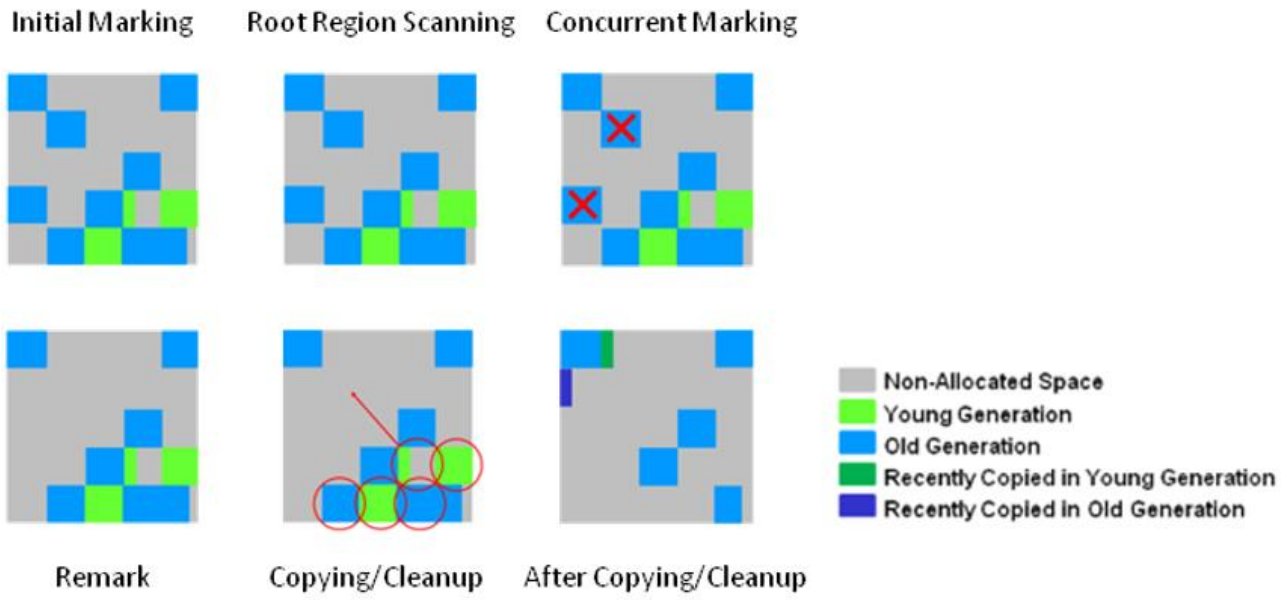
1）FullGC 发生相对比较频繁或消耗的总时长过长
2）对象分配率或对象升级至老年代的比例波动较大
3）较长时间的内存整理停顿
说明：如果你现在用 CMS 或者 ParallelOldGC ，并且你的程序运行很好，没有经历长时间垃圾回收停顿，建议就不用迁移。

G1 算法应用分析：

1）年轻代标记复制算法
2）老年代标记清除整理算法。

G1 关键步骤应用分析： (Old Generation)
1）初始标记(Initial Mark)：属于 Young GC 范畴，是 stop-the-world 活动。对持有老年代对象引用的 Survivor 区（Root 区）进行标记。
2）根区扫描(Root Region Scan)：并发执行，扫描那些对 old 区有引用的 Survivor 区，在 yongGC 发生之前该阶段必须完成。
3）并发标记(Concurrent Mark)：并发执行，找出整个堆中存活的对象，将空区域标记为”X”,此阶段也可能会被 Young GC 中断。
4）再次标记(Remark)：完全完成对 heap 存活对象的标记。采用 snapshot-at-the-beginning (SATB) 算法完成，比 CMS 用的算法更快。
5）清理(cleanup)：并发执行，统计小堆区中所有存活的对象，并对小堆区进行排序,优先清理垃圾多的小堆区，释放内存。
6）复制/清理(copy/clean)：对小堆区未被清理对象对象进行复制，然后再清理。

G1 实践应用分析：



其应用参数配置：-XX:+UseG1GC 表示启用 GC 收集器

其它参数配置：

1) -XX:MaxGCPauseMillis=200 - 设置最大 GC 停顿时间(GC pause time)指标(target). 这是一个软性指标(soft goal), JVM 会尽力去达成这个目标. 所以有时候这个目标并不能达成. 默认值为 200 毫秒.
2) -XX:InitiatingHeapOccupancyPercent=45 - 启动并发 GC 时的堆内存占用百分比. G1 用它来触发并发 GC 周期,基于整个堆的使用率,而不只是某一代内存的使用比例。值为 0 则表示 “一直执行 GC 循环)'. 默认值为 45 (表示堆使用了 45%).

总之：

G1 是 HotSpot 中最先进的准产品级(production-ready)垃圾收集器。重要的是， HotSpot 工程师的主要精力都放在不断改进 G1 上面， 在新的 java 版本中,将会带来新的功能和优化。