1.JVM 内存分配与回收

1.1 对象优先在 Eden 区分配

大多数情况下，对象在新生代中 Eden 区分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。我们来进行实际测试一下。

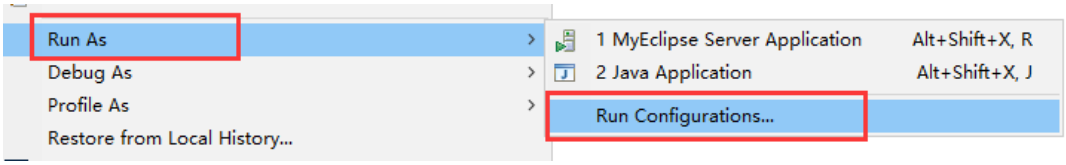
在测试之前我们先来看看 **Minor Gc 和 Full GC 有什么不同呢？**

- **新生代 GC（Minor GC）**:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- **老年代 GC（Major GC/Full GC）**:指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

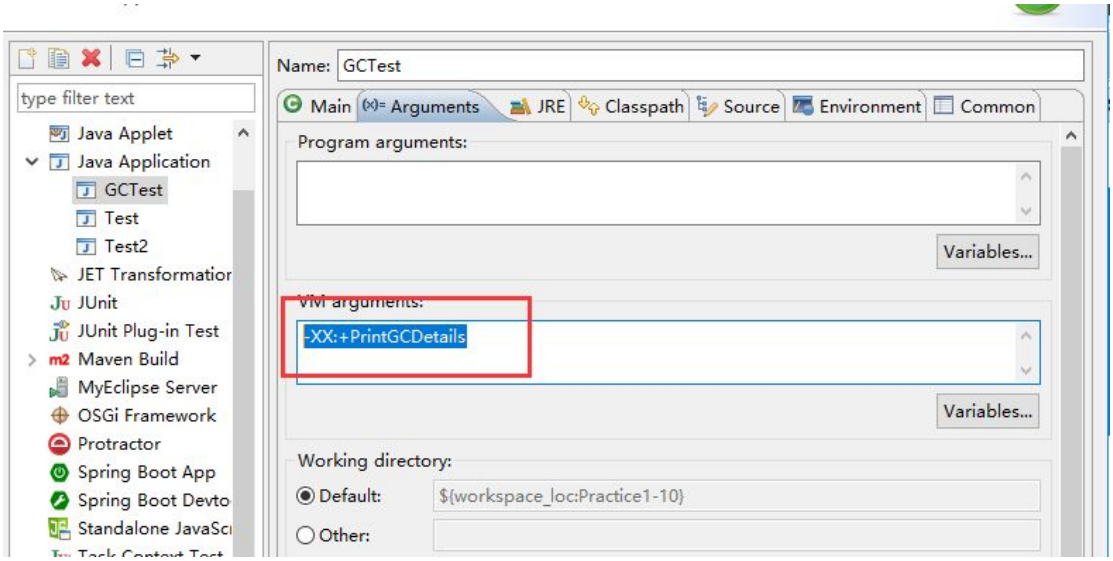
测试：

```
public class GCTest {
    public static void main(String[] args) {
        byte[] allocation1, allocation2;
        allocation1 = new byte[30231 * 1024];
        //allocation2 = new byte[900*1024];
    }
}
```

通过以下方式运行：



添加的参数： **-XX:+PrintGCDetails**



运行结果：

```
Heap
 PSYoungGen      total 38400K, used 33280K [0x00000000d5f00000, 0x00000000d8980000, 0x0000000100000000)
  eden space 33280K, 100% used [0x00000000d5f00000,0x00000000d7f80000,0x00000000d7f80000)
  from space 5120K, 0% used [0x00000000d8480000,0x00000000d8480000,0x00000000d8980000)
  to   space 5120K, 0% used [0x00000000d7f80000,0x00000000d7f80000,0x00000000d8480000)
 ParOldGen       total 87552K, used 0K [0x0000000081c00000, 0x0000000087180000, 0x00000000d5f00000)
  object space 87552K, 0% used [0x0000000081c00000,0x0000000081c00000,0x0000000087180000)
 Metaspace       used 2812K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 302K, capacity 386K, committed 512K, reserved 1048576K
```

从上图我们可以看出 eden 区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用至少 2000 多 k 内存）。**假如我们再为 allocation2 分配内存会出现什么情况呢？**

```
[GC (Allocation Failure) [PSYoungGen: 32893K->944K(38400K)] 32893K->31183K(125952K), 0.0169586 secs] [Times: user=0.08 sys=0.00, real=0.02 secs]
Heap
 PSYoungGen      total 38400K, used 2177K [0x00000000d5f00000, 0x00000000daa00000, 0x0000000100000000)
  eden space 33280K, 3% used [0x00000000d5f00000,0x00000000d60344b8,0x00000000d7f80000)
  from space 5120K, 18% used [0x00000000d7f80000,0x00000000d806c040,0x00000000d8480000)
  to   space 5120K, 0% used [0x00000000da500000,0x00000000da500000,0x00000000daa00000)
 ParOldGen       total 87552K, used 30239K [0x0000000081c00000, 0x0000000087180000, 0x00000000d5f00000)
  object space 87552K, 34% used [0x0000000081c00000,0x0000000083987c10,0x0000000087180000)
 Metaspace       used 2812K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 302K, capacity 386K, committed 512K, reserved 1048576K
```

简单解释一下为什么会出现这种情况： 因为给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，我们刚刚讲了当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.GC 期间虚拟机又发现 allocation1 无法存入 Survior 空间，所以只好通过 **分配担保机制** 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执

行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存。可以执行如下代码验证：

```
public class GCTest {
    public static void main(String[] args) {
        byte[] allocation1, allocation2, allocation3, allocation4, allocation5, allocation6;
        allocation1 = new byte[30231 * 1024];
        allocation2 = new byte[900*1024];
        allocation3 = new byte[1000*1024];
        allocation4 = new byte[1000*1024];
        allocation5 = new byte[1000*1024];
        allocation6 = new byte[1000*1024];
    }
}
```

1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

为什么要这样呢？

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

1.3 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别那些对象应放在新生代，那些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1.对象在 Survivor 中每熬过一次 MinorGC,年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 **-XX:MaxTenuringThreshold** 来设置。

2.如何判断对象可以被回收

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。

2.1 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。 所谓对象之间的相互引用问题，如下面代码所示：除了对象 objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为 0，于是引用计数算法无法通知 GC 回收器回收他们。

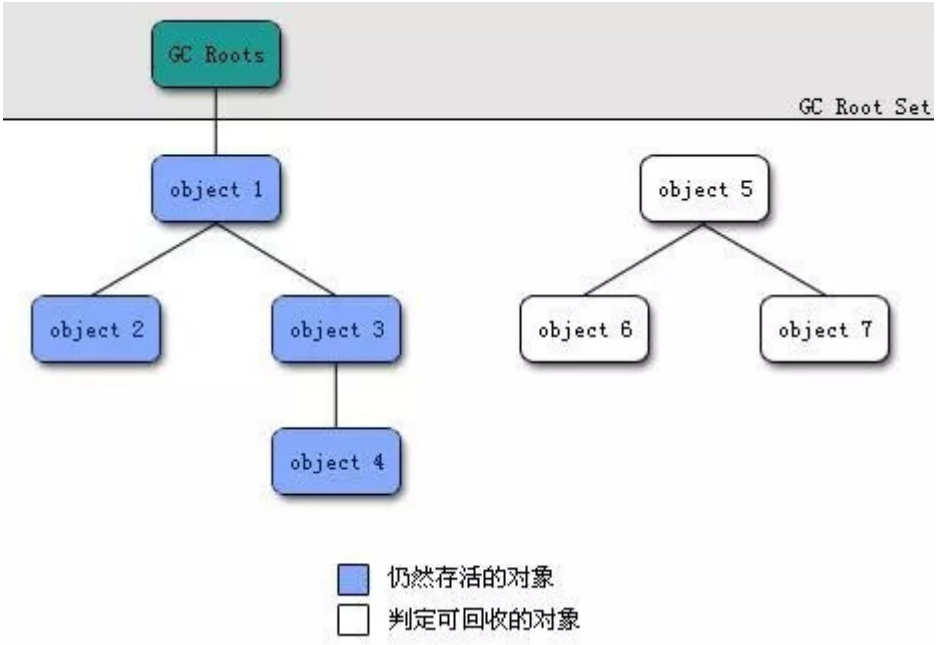
```
public class ReferenceCountingGc {
    Object instance = null;

    public static void main(String[] args) {
        ReferenceCountingGc objA = new ReferenceCountingGc();
        ReferenceCountingGc objB = new ReferenceCountingGc();
        objA.instance = objB;
        objB.instance = objA;
        objA = null;
        objB = null;
    }
}
```

2.2 可达性分析算法

这个算法的基本思想就是通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC

Roots 没有任何引用链相连的话，则证明此对象是不可用的。

GC Roots 根节点：类加载器、Thread、虚拟机栈的本地变量表、static 成员、常量引用、本地方法栈的变量等等



2.3 finalize()方法最终判定对象是否存活

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历再次标记过程。

标记的前提是对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链。

1. 第一次标记并进行一次筛选。

筛选的条件是此对象是否有必要执行 finalize()方法。

当对象没有覆盖 finalize 方法，或者 finzlize 方法已经被虚拟机调用过，虚拟

机将这两种情况都视为“没有必要执行”，对象被回收。

2. 第二次标记

如果这个对象被判定为有必要执行 finalize（）方法，那么这个对象将会被放置在一个名为：F-Queue 的队列之中，并在稍后由一条虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象 finalize（）方法中执行缓慢，或者发生死循环（更极端的情况），将很可能会导致 F-Queue 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。finalize（）方法是对象脱逃死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模标记，如果对象要在 finalize（）中成功拯救自己----只要重新与引用链上的任何的一个对象建立关联即可，譬如把自己赋值给某个类变量或对象的成员变量，那在第二次标记时它将移除出“即将回收”的集合。如果对象这时候还没逃脱，那基本上它就真的被回收了。

见示例程序：

```
public class OOMTest {
    // JVM设置
    // -Xms10M -Xmx10M -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError
    public static void main(String[] args) {
        List<Object> list = new ArrayList<>();
        int i = 0;
        int j = 0;
        while (true) {
            list.add(new User(i++, UUID.randomUUID().toString()));
            new User(j--, UUID.randomUUID().toString());
        }
    }
}
```

2.4 如何判断一个常量是废弃常量

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢?

假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

2.5 如何判断一个类是无用的类

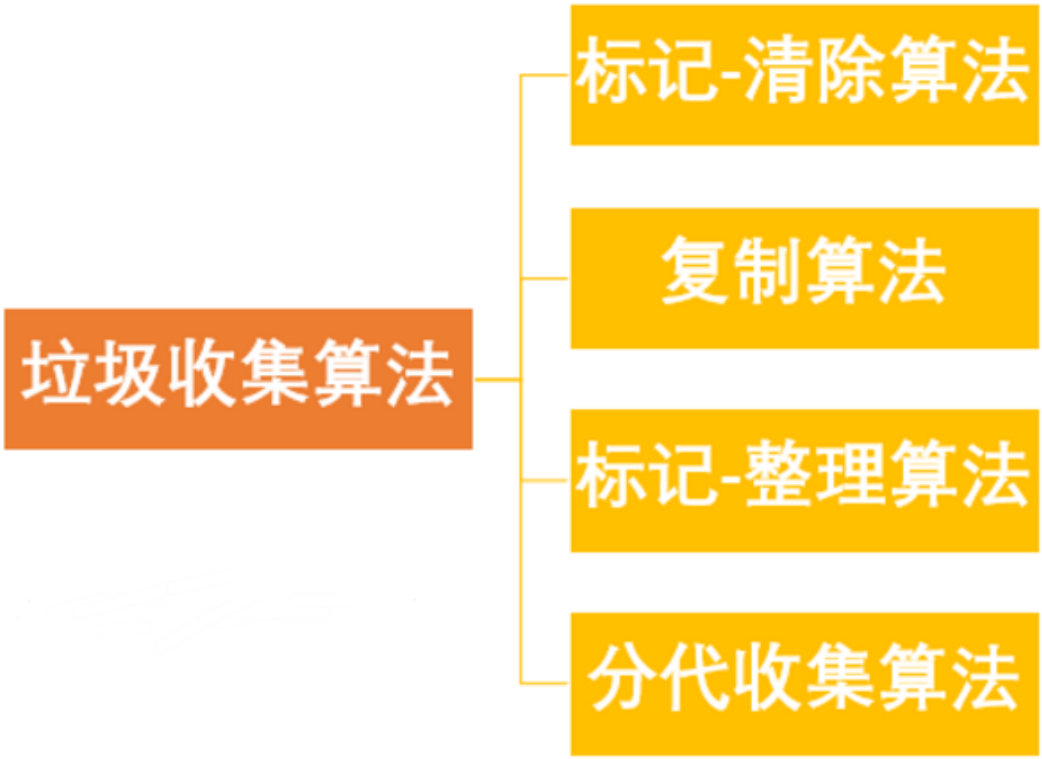
方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢?

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是 **“无用的类”** ：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

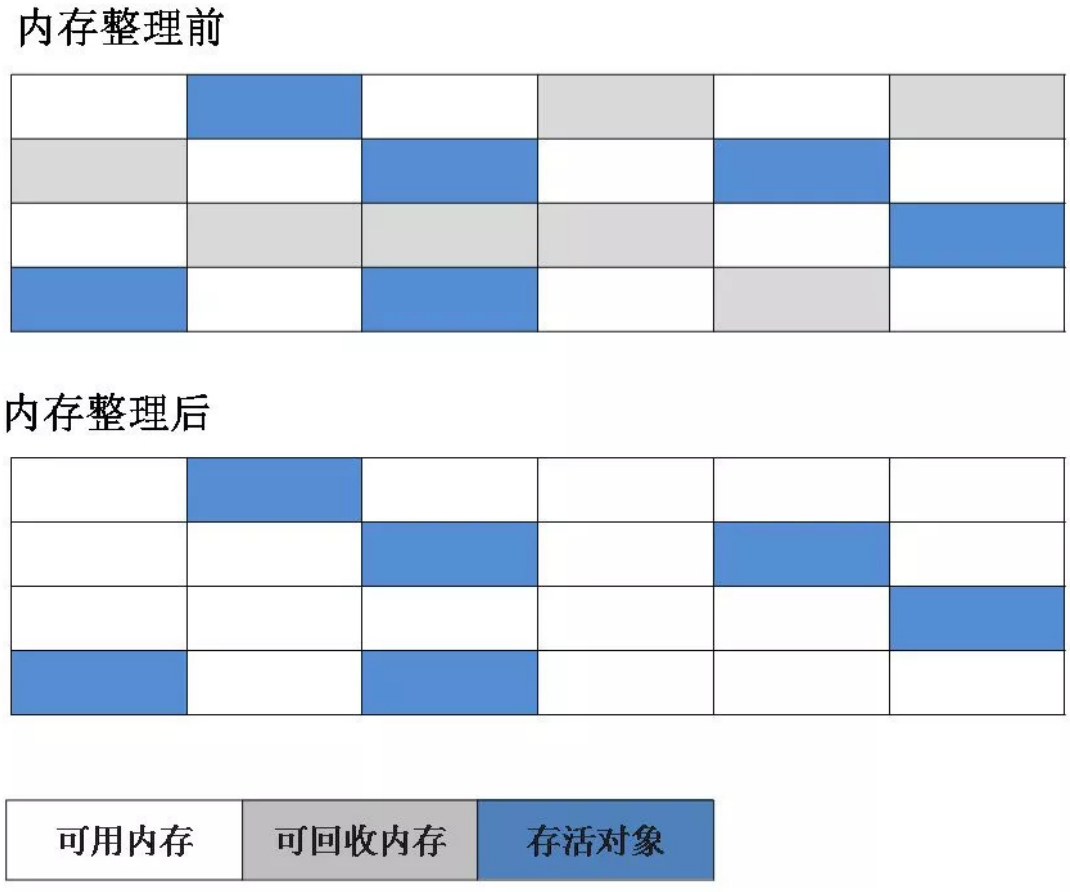
3.垃圾收集算法



3.1 标记-清除算法

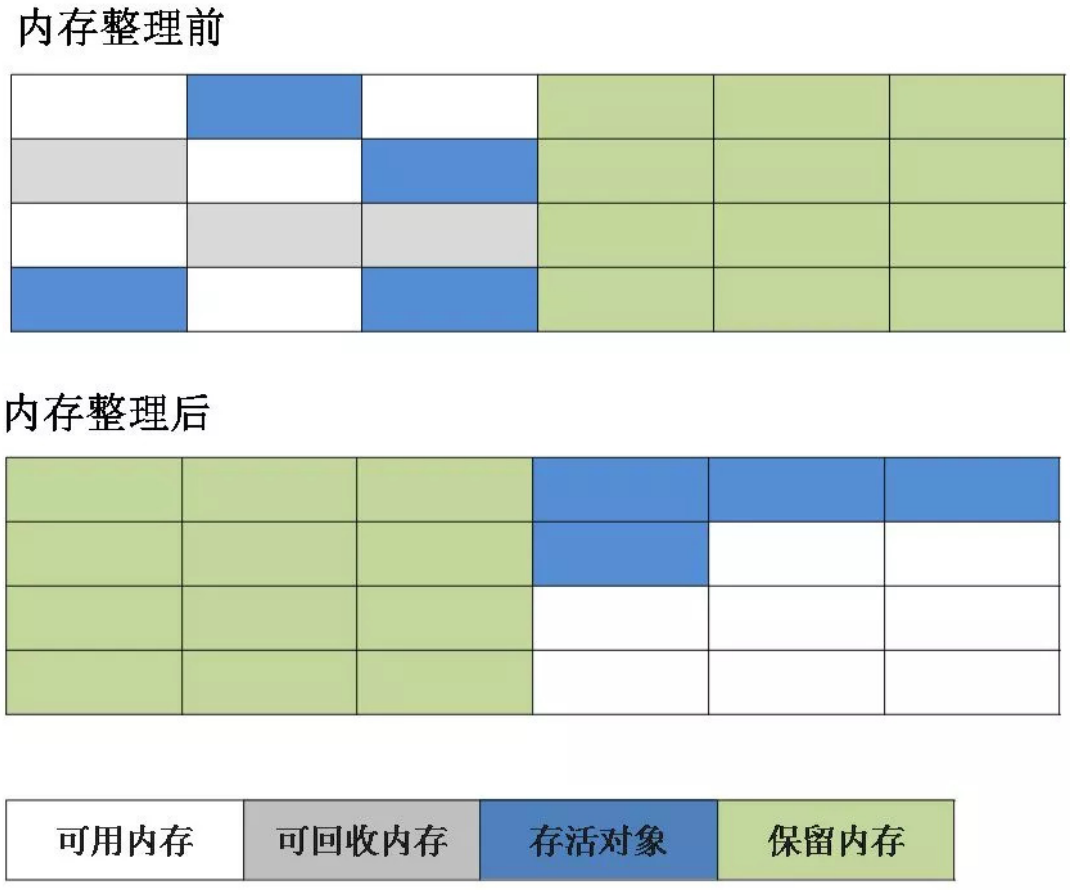
算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：

1. **效率问题**
2. **空间问题（标记清除后会产生大量不连续的碎片）**



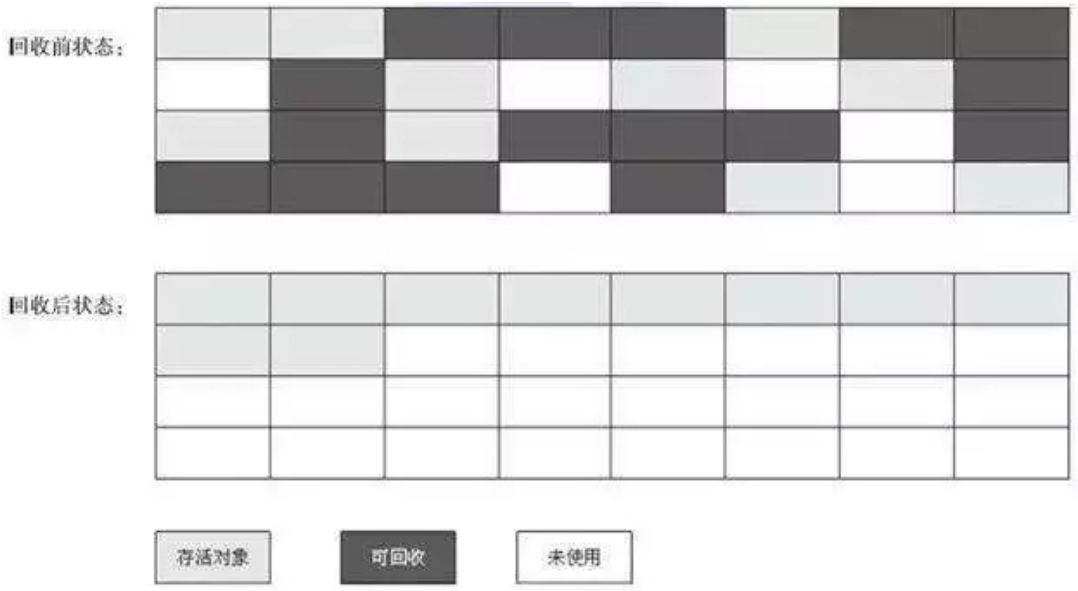
3.2 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。



3.3 标记-整理算法

根据老年代的特点特出的一种标记算法，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回收，而是让所有存活的对象向一段移动，然后直接清理掉端边界以外的内存。

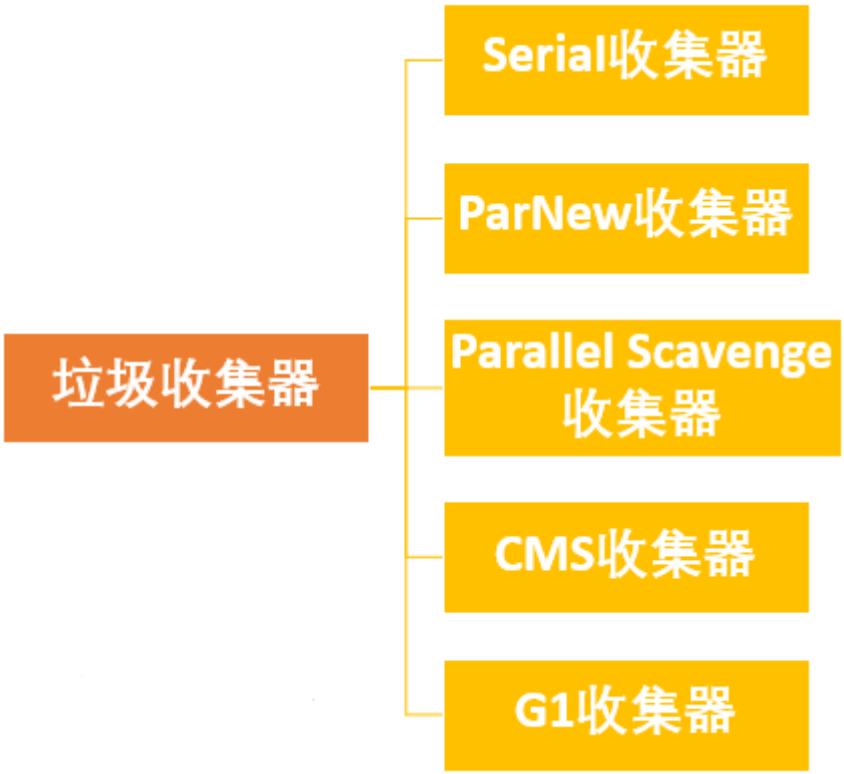


3.4 分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

4.垃圾收集器



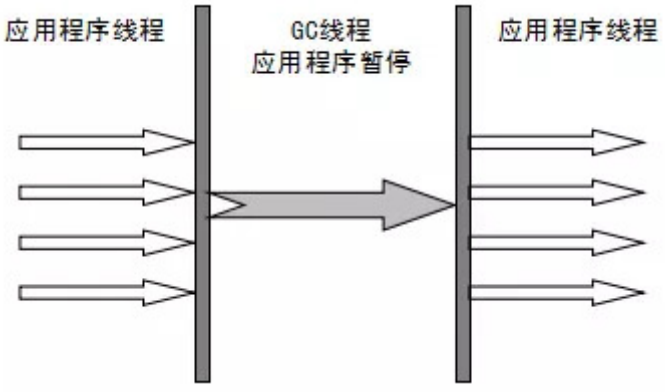
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们对各个收集器进行比较，但并非为了挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，**我们能做的就是根据具体应用场景选择适合自己的垃圾收集器**。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现那么多不同的垃圾收集器了。

4.1 Serial 收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的 **“单线程”** 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ **"Stop The World"** ），直到它收集结束。

新生代采用复制算法，老年代采用标记-整理算法。



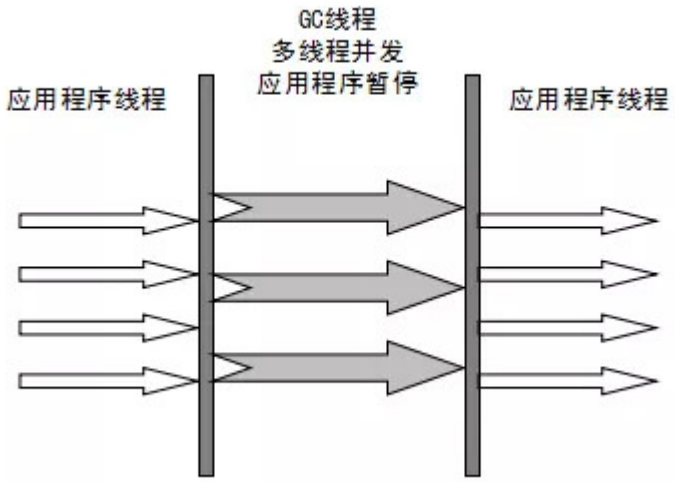
虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是 Serial 收集器有没有优于其他垃圾收集器的地方呢？当然有，它**简单而高效（与其他收集器的单线程相比）**。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。

4.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。

新生代采用复制算法，老年代采用标记-整理算法。



它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

并行和并发概念补充：

- **并行（Parallel）**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。适合科学计算、后台处理等弱交互场景。
- **并发（Concurrent）**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另

一个 CPU 上。适合 Web 应用。

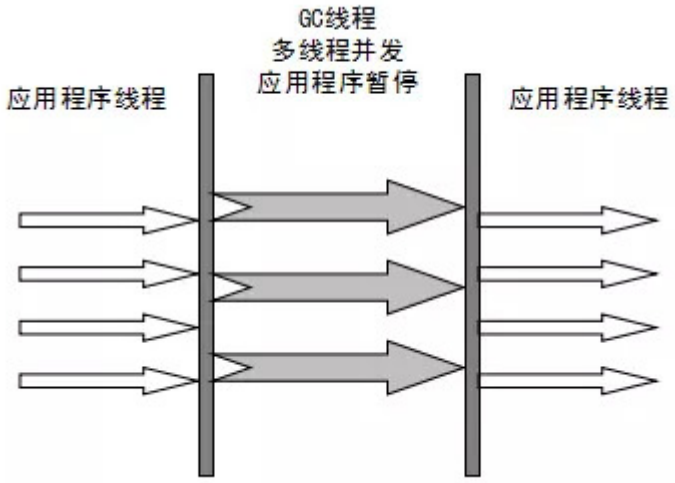
4.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器类似于 ParNew 收集器，是 Server 模式（**内存大于 2G，2 个 cpu**）下的**默认收集器**，**那么它有什么特别之处呢？**

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。

Parallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。



4.4.Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

4.5 Parallel Old 收集器

Parallel Scavenge 收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

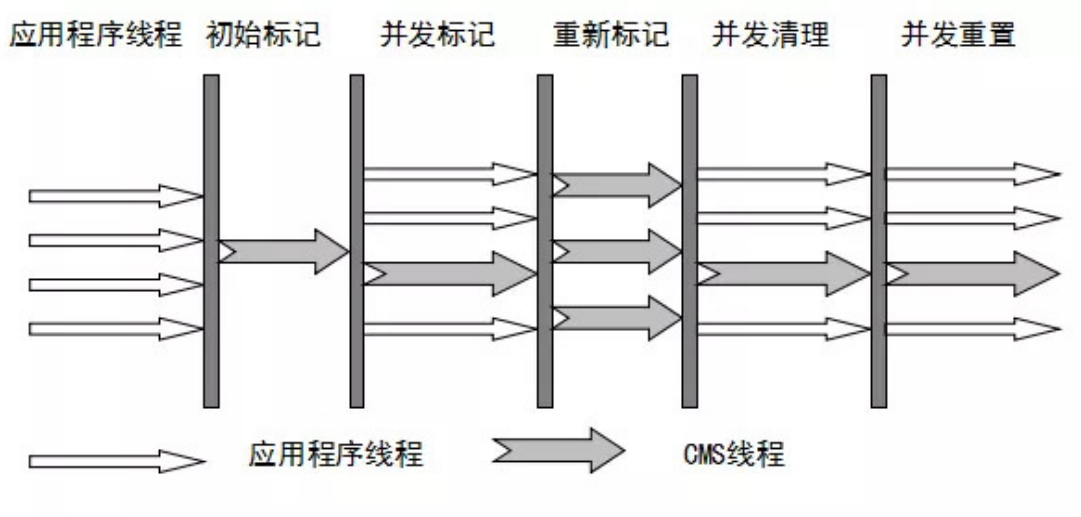
4.6 CMS 收集器(-XX:+UseConcMarkSweepGC(主要是 old 区使用))

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。它而非常符合在注重用户体验的应用上使用，它是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的 **Mark Sweep** 这两个词可以看出，CMS 收集器是一种 **“标记-清除”算法**实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记：** 暂停所有的其他线程(**STW**)，并记录下直接与 root 相连的对象，速度很快；
- **并发标记：** 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
- **重新标记：** 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短

- **并发清除：** 开启用户线程，同时 GC 线程开始对未标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

- **对 CPU 资源敏感（会和服务抢资源）；**
- **无法处理浮动垃圾(在 java 业务程序线程与垃圾收集线程并发执行过程中又产生的垃圾，这种浮动垃圾只能等到下一次 gc 再清理了)；**
- **它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。**

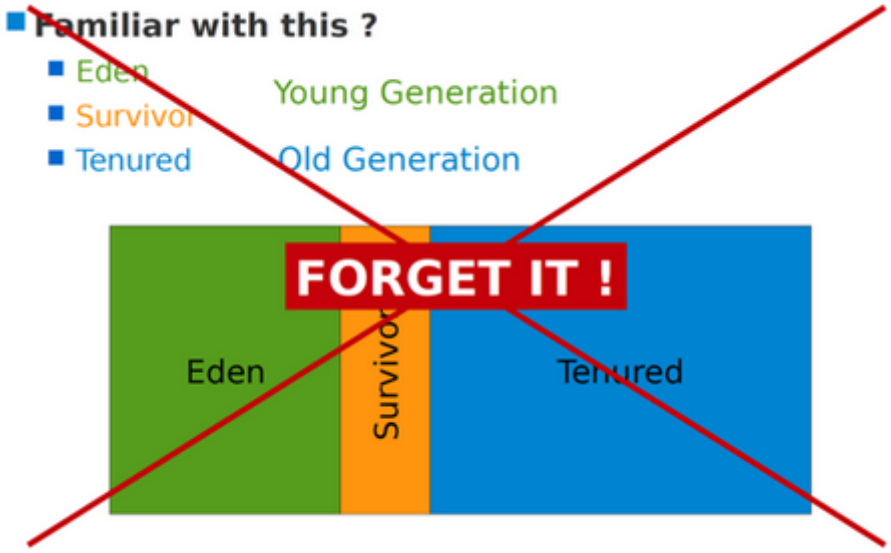
CMS 的相关参数

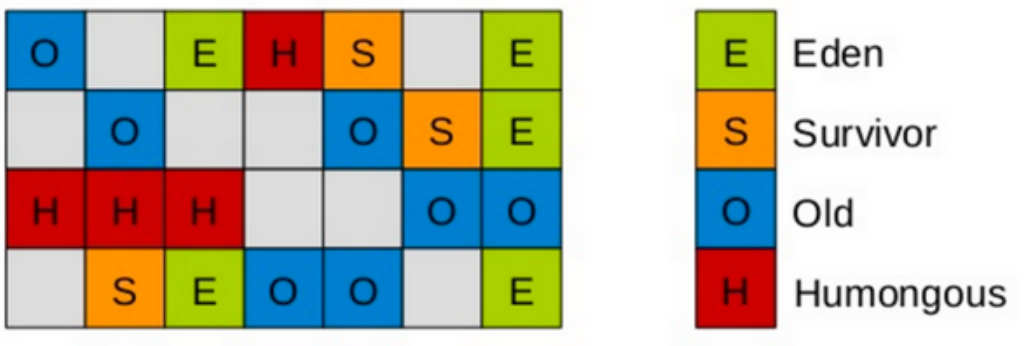
3. -XX:+UseConcMarkSweepGC 启用 cms
4. -XX:ConcGCThreads:并发的 GC 线程数（并非 STW 时间，而是和服务一起执行的线程数）
5. -XX:+UseCMSCompactAtFullCollection:FullGC 之后做压缩（减少碎片）
6. -XX:CMSFullGCsBeforeCompaction:多少次 FullGC 之后压缩一次（因压缩非常的消耗时间，所以不能每次 FullGC 都做）
7. -XX:CMSInitiatingOccupancyFraction:触发 FulGC 条件（默认是 92）

8. -XX:+UseCMSInitiatingOccupancyOnly:是否动态调节
9. -XX:+CMSScavengeBeforeRemark:FullGC 之前先做 YGC（一般这个参数是打开的）
10. -XX:+CMSClassUnloadingEnabled:启用回收 Perm 区（jdk1.7 及以前）

4.7 G1 收集器(-XX:+UseG1GC)

G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.





G1 将 Java 堆划分为多个大小相等的独立区域（**Region**），虽保留新生代和

老年代的概念，但不再是物理隔阂了，它们都是（可以不连续）Region 的集合。

分配大对象（直接进 Humongous 区，专门存放短期巨型对象，不用直接进老年代，避免 Full GC 的大量开销）不会因为无法找到连续空间而提前触发下一次 GC。

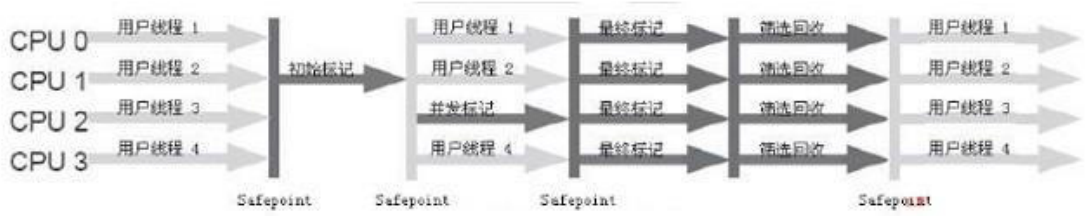
被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备以下特点：

- **并行与并发**：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程来执行 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。
- **分代收集**：虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- **空间整合**：与 CMS 的“标记--清理”算法不同，G1 从整体来看是基于“**标记整理**”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿**：这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内完成垃圾收集。

G1 收集器的运作大致分为以下几个步骤：

- **初始标记**（initial mark，STW）：在此阶段，G1 GC 对根进行标记。该阶段与常规的 (STW) 年轻代垃圾回收密切相关。

- **并发标记**（Concurrent Marking）：G1 GC 在整个堆中查找可访问的（存活的）对象。
- **最终标记**（Remark，STW）：该阶段是 STW 回收，帮助完成标记周期。
- **筛选回收**（Cleanup，STW）：筛选回收阶段首先对各个 Region 的回收价值和成本进行排序，**根据用户所期望的 GC 停顿时间来制定回收计划**，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。



G1 收集器运行示意图

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 GF 收集器在有限时间内可以尽可能高的收集效率。

G1 垃圾收集分类

YoungGC

11. 新对象进入 Eden 区
12. 存活对象拷贝到 Survivor 区
13. 存活时间达到年龄阈值时，对象晋升到 Old 区

MixedGC

14. 不是 FullGC，回收所有的 Young 和部分 Old(根据期望的 GC 停顿时间确

定 old 区垃圾收集的优先顺序)

15. global concurrent marking （全局并发标记）
    a) Initial marking phase:标记 GC Root，STW
    b) Root region scanning phase：标记存活 Region
    c) Concurrent marking phase：标记存活的对象
    d) Remark phase :重新标记,STW
    e) Cleanup phase:部分 STW
16. 相关参数
    a) G1MixedGCLiveThresholdPercent Old 区的 region 被回收的时候的存活对象占比
    b) G1MixedGCCountTarget：一次 global concurrent marking 之后，最多执行 Mixed GC 的次数
    c) G1OldCSetRegionThresholdPercent 一次 Mixed GC 中能被选入 CSet 的最多 old 区的 region 数量
17. **触发的时机**
    a) **InitiatingHeapOccupancyPercent**:堆占有率达到这个值则触发 global concurrent marking，默认 45%
    b) G1HeapWastePercent:在 global concurrent marking 结束之后，可以知道区有多少空间要被回收，在每次 YGC 之后和再次发生 Mixed GC 之前，会检查垃圾占比是否达到了此参数，只有达到了，下次才会发生 Mixed GC

5. 如何选择垃圾收集器

18. 优先调整堆的大小让服务器自己来选择

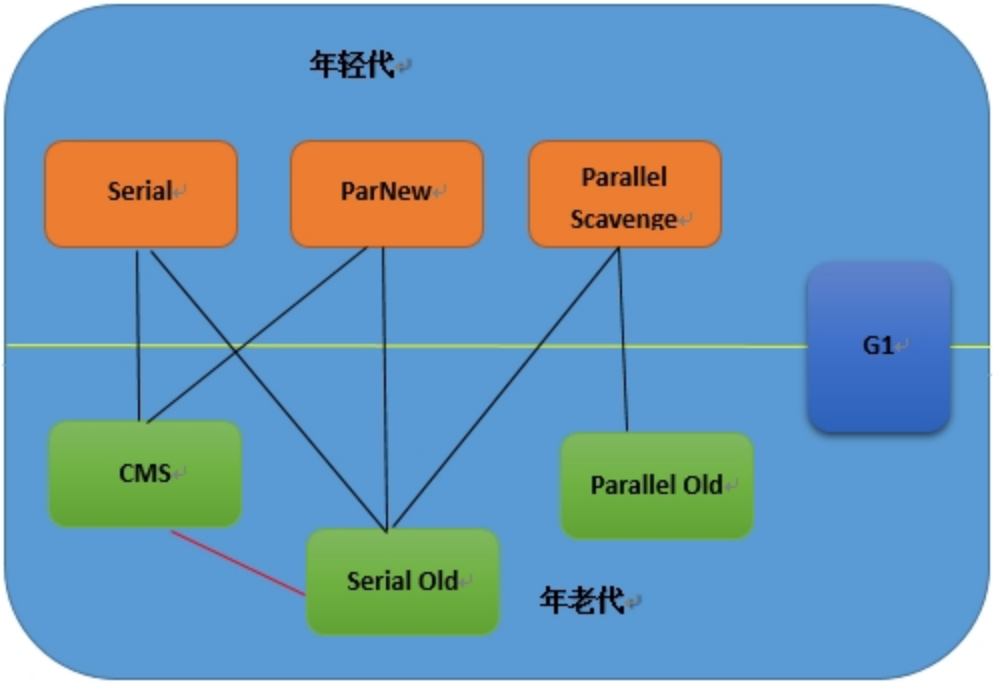
19. 如果内存小于 100M，使用串行收集器

20. 如果是单核，并且没有停顿时间的要求，串行或 JVM 自己选择

21. 如果允许停顿时间超过 1 秒，选择并行或者 JVM 自己选

22. 如果响应时间最重要，并且不能超过 1 秒，使用并发收集器

下图有连线的可以搭配使用，官方推荐使用 G1，因为性能高



6. 实战调优

JVM 调优主要就是调整下面两个指标

停顿时间：垃圾收集器做垃圾回收中断应用执行的时间。-

XX:MaxGCPauseMillis

吞吐量：花在垃圾收集的时间和花在应用时间的占比 -

XX:GCTimeRatio=<n>,垃圾收集时间占比：1/(1+n)

GC 调优步骤

- 打印 GC 日志

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -

XX:+PrintGCDateStamps -Xloggc:./gc.log

- 分析日志得到关键性指标
- 分析 GC 原因，调优 JVM 参数

1、Parallel Scavenge 收集器(默认)

分析 parallel-gc.log

第一次调优，设置 Metaspace 大小：增大元空间大小-XX:MetaspaceSize=64M

-XX:MaxMetaspaceSize=64M

第二次调优，添加吞吐量和停顿时间参数：-XX:MaxGCPauseMillis=100 -

XX:GCTimeRatio=99

第三次调优，修改动态扩容增量：-XX:YoungGenerationSizeIncrement=30

2、配置 CMS 收集器

-XX:+UseConcMarkSweepGC

分析 cms-gc.log

3、配置 G1 收集器

-XX:+UseG1GC

分析 g1-gc.log

查看发生 MixedGC 的阈值：jinfo -flag InitiatingHeapOccupancyPercent

进程 id

分析工具：**gceasy**，GCViewer

G1 调优相关

23. 常用参数
    a) -XX:+UseG1GC 开启 G1
    b) -XX:G1HeapRegionSize=n,region 的大小，1-32M，2048 个
    c) -XX:MaxGCPauseMillis=200 最大停顿时间
    d) -XX:G1NewSizePercent -XX:G1MaxNewSizePercent
    e) -XX:G1ReservePercent=10 保留防止 to space 溢出（）
    f) -XX:ParallelGCThreads=n SWT 线程数（停止应用程序）
    g) -XX:ConcGCThreads=n 并发线程数=1/4*并行
24. 最佳实践
    a) 年轻代大小：避免使用-Xmn、-XX:NewRatio 等显示设置 Young 区大小，会覆盖暂停时间目标（常用参数 3）
    b) 暂停时间目标：暂停时间不要太严苛，其吞吐量目标是 90%的应用程序时间和 10%的垃圾回收时间，太严苛会直接影响到吞吐量

25. 是否需要切换到 G1

    a) 50%以上的堆被存活对象占用

    b) 对象分配和晋升的速度变化非常大

    c) 垃圾回收时间特别长，超过 1 秒

26. G1 调优目标

    a) 6GB 以上内存

    b) 停顿时间是 500ms 以内

    c) 吞吐量是 90%以上

GC 常用参数

堆栈设置

-Xss:每个线程的栈大小

-Xms:初始堆大小，默认物理内存的 1/64

-Xmx:最大堆大小，默认物理内存的 1/4

-Xmn:新生代大小

-XX:NewSize:设置新生代初始大小

-XX:NewRatio:默认 2 表示新生代占年老代的 1/2，占整个堆内存的 1/3。

-XX:SurvivorRatio:默认 8 表示一个 survivor 区占用 1/8 的 Eden 内存，即 1/10 的新生代内存。

-XX:MetaspaceSize:设置元空间大小

-XX:MaxMetaspaceSize:设置元空间最大允许大小，默认不受限制，JVM Metaspace 会进行动态扩展。

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

收集器设置

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParallelOldGC:老年代使用并行回收收集器

-XX:+UseParNewGC:在新生代使用并行收集器

-XX:+UseParalledlOldGC:设置并行老年代收集器

-XX:+UseConcMarkSweepGC:设置 CMS 并发收集器

-XX:+UseG1GC:设置 G1 收集器

-XX:ParallelGCThreads:设置用于垃圾回收的线程数

并行收集器设置

-XX:ParallelGCThreads:设置并行收集器收集时使用的 CPU 数。并行收集线程数。

-XX:MaxGCPauseMillis:设置并行收集最大暂停时间

-XX:GCTimeRatio:设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

CMS 收集器设置

-XX:+UseConcMarkSweepGC:设置 CMS 并发收集器

-XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况。

-XX:ParallelGCThreads:设置并发收集器新生代收集方式为并行收集时，使用的 CPU 数。

并行收集线程数。

-XX:CMSFullGCsBeforeCompaction:设定进行多少次 CMS 垃圾回收后，进行一次内存压缩

-XX:+CMSClassUnloadingEnabled:允许对类元数据进行回收

-XX:UseCMSInitiatingOccupancyOnly:表示只在到达阀值的时候，才进行 CMS 回收

-XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况

-XX:ParallelCMSThreads:设定 CMS 的线程数量

-XX:CMSInitiatingOccupancyFraction:设置 CMS 收集器在老年代空间被使用多少后触发

-XX:+UseCMSCompactAtFullCollection:设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片的整理

G1 收集器设置

-XX:+UseG1GC:使用 G1 收集器

-XX:ParallelGCThreads:指定 GC 工作的线程数量

-XX:G1HeapRegionSize:指定分区大小(1MB~32MB，且必须是 2 的幂)，默认将整堆划分为 2048 个分区

-XX:GCTimeRatio:吞吐量大小，0-100 的整数(默认 9)，值为 n 则系统将花费不超过 1/(1+n)的时间用于垃圾收集

-XX:MaxGCPauseMillis:目标暂停时间(默认 200ms)

-XX:G1NewSizePercent:新生代内存初始空间(默认整堆 5%)

-XX:G1MaxNewSizePercent:新生代内存最大空间

-XX:TargetSurvivorRatio:Survivor 填充容量(默认 50%)

-XX:MaxTenuringThreshold:最大任期阈值(默认 15)

-XX:InitiatingHeapOccupancyPercen:老年代占用空间超过整堆比 IHOP 阈值(默认 45%),超过则执行混合收集

-XX:G1HeapWastePercent:堆废物百分比(默认 5%)

-XX:G1MixedGCCountTarget:参数混合周期的最大总次数(默认 8)