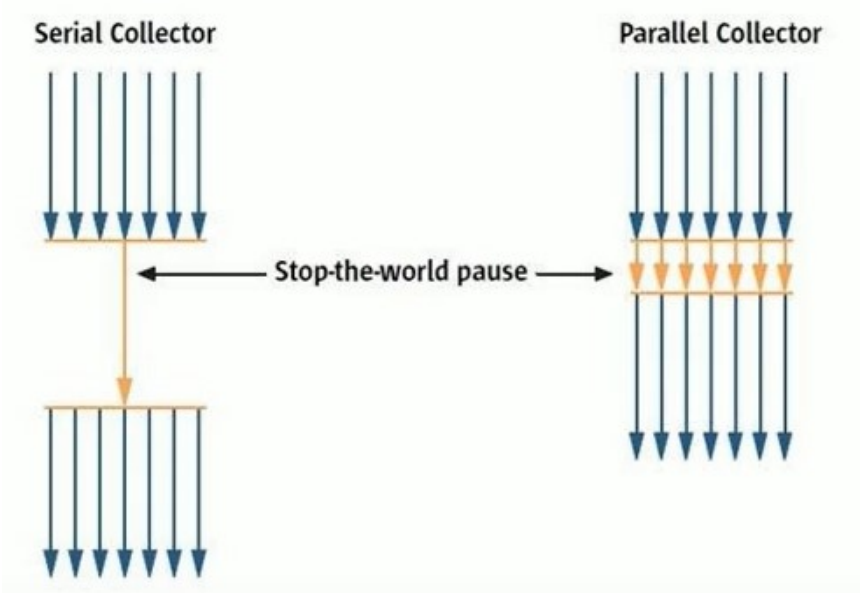
GC分类与性能指标

- 垃圾收集器没有在规范中进行过多的规定，可以由不同的厂商、不同版本的JVM来实现。
- 由于JDK的版本处于高速迭代过程中， 因此Java发展至今已经衍生了众多的GC版本。
- 从不同角度分析垃圾收集器，可以将GC分为不同的类型。

线程数来分

可以分为串行垃圾回收器和并行垃圾回收器



串行回收

指的是在同一时间段内只允许有一个CPU用于执行垃圾回收操作，此时工作线程被暂停，直至垃圾收集工作结束。

- 在诸如单CPU处理器或者较小的应用内存等硬件平台不是特别优越的场合，串行回收器的性能表现可以超过并行回收器和并发回收器。所以， 串行回收默认被应用在客户端的Client模式下的JVM中
- 在并发能力比较强的CPU上， 并行回收器产生的停顿时间要短于串行回收器。

并行回收

和串行回收相反， 并行收集可以运用多个CPU同时执行垃圾回收，

因此提升了应用的吞吐量，不过并行回收仍然与串行回收一样，采用独占式，使用了“Stop-the-world”机制。

工作模式分

按照工作模式分，可以分为并发式垃圾回收器和独占式垃圾回收器。

- 并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间。
- 独占式垃圾回收器(Stop the world) 一旦运行， 就停止应用程序中的所有用户线程，直到垃圾回收过程完全结束。

碎片处理方式分

可分为**压缩式垃圾回收器**和**非压缩式垃圾回收器**。

- 压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片。
    - 在分配对象空间：指针碰撞
- 非压缩式的垃圾回收器不进行这步操作。
    - 在分配对象空间：空闲列表

工作的内存区间分

又可分为**年轻代垃圾回收器**和**老年代垃圾回收器**。

评估GC性能指标

- **吞吐量：运行用户代码的时间占总运行时间的比例**
  - (总运行时间：程序的运行时间+内存回收的时间)
- 垃圾收集开销：吞吐量的补数，垃圾收集所用时间与总运行时间的比例。
- **暂停时间：执行垃圾收集时，程序的工作线程被暂停的时间。**
- 收集频率：相对于应用程序的执行，收集操作发生的频率。
- **内存占用：Java堆区所占的内存大小。**
- 快速：一个对象从诞生到被回收所经历的时间。

红色部分三者

- 这三者共同构成一个“不可能三角”。三者总体的表现会随着技术进步而越来越好。
  一款优秀的收集器通常最多同时满足其中的两项。
- 这三项里，暂停时间的重要性日益凸显。
  因为随着硬件发展，内存占用多些越来越能容忍，硬件性能的提升也有助于降低收集器运行时对应用程序的影响，即提高了吞吐量。
  而内存的扩大，对延迟反而带来负面效果。

简单来说，主要抓住两点：

- 吞吐量
- 暂停时间

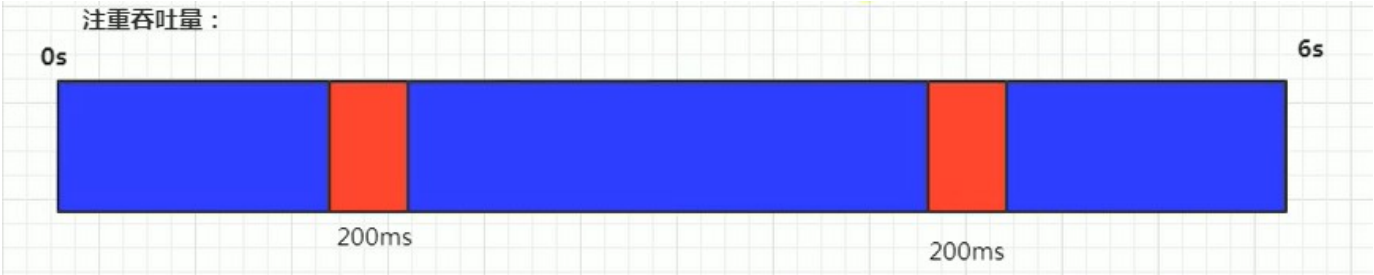
吞吐量

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)。

- 比如：虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

这种情况下，应用程序能容忍较高的暂停时间，因此，高吞吐量的应用程序有更长的时间基准，快速响应是不必考虑的。

吞吐量优先， 意味着在单位时间内， STW的时间最短：0.2+0.2=0.4

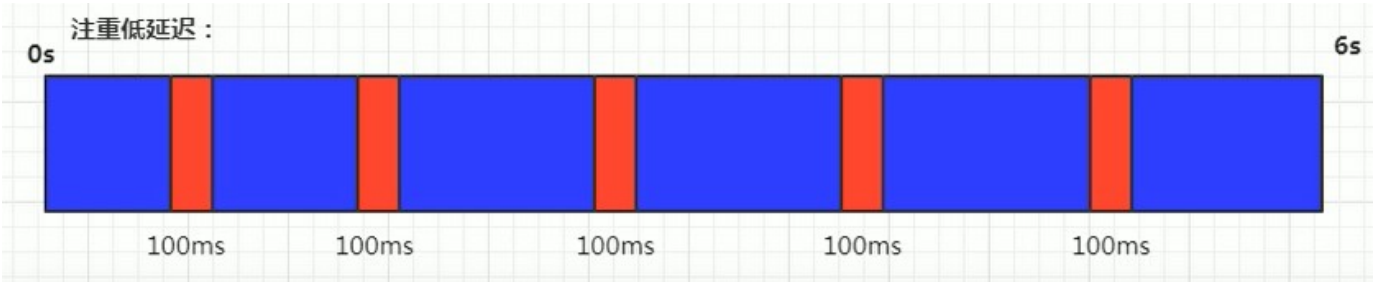


暂停时间

“暂停时间” 是指一个时间段内应用程序线程暂停，让GC线程执行的状态

- 例如，GC期间100毫秒的暂停时间意味着在这100毫秒期间内没有应用程序线程是活动的。

暂停时间优先， 意味着尽可能让单次STW的时间最短：0.1+0.1+0.1+ 0.1 +0.1=0.5



评估

- 高吞吐量较好因为这会让应用程序的最终用户感觉只有应用程序线程在做“生产性”工作。
  直觉上，吞吐量越高程序运行越快。
- 低暂停时间(低延迟)较好因为从最终用户的角度来看不管是GC还是其他原因导致一个应用被挂起始终是不好的。
  这取决于应用程序的类型，有时候甚至短暂的200毫秒暂停都可能打断终端用户体验。
  因此，具有低的较大暂停时间是非常重要的，特别是对于一个交互式应用程序。
- 不幸的是”高吞吐量”和”低暂停时间”是一对相互竞争的目标(矛盾)。
  - 因为如果选择以**吞吐量优先**，那么必然需要降低内存回收的执行频率，
    但是这样会导致GC需要更长的暂停时间来执行内存回收。
  - 相反的，如果选择以**低延迟优先**为原则，那么为了降低每次执行内存回收时的暂停时间，
    也只能频繁地执行内存回收，但这又引起了年轻代内存的缩减和导致程序吞吐量的

下降。

- 在设计(或使用)GC算法时，我们必须确定我们的目标：
  一个GC算法只可能针对两个目标之一(即只专注于较大吞吐量或最小暂停时间)，
  或尝试找到一个二者的折衷。

现在标准：**在最大吞吐量优先的情况下，降低停顿时间**。

不同的垃圾回收器概述

垃圾收集机制是Java的招牌能力， 极大地提高了开发效率。这当然也是面试的热点。

那么， Java常见的垃圾收集器有哪些?

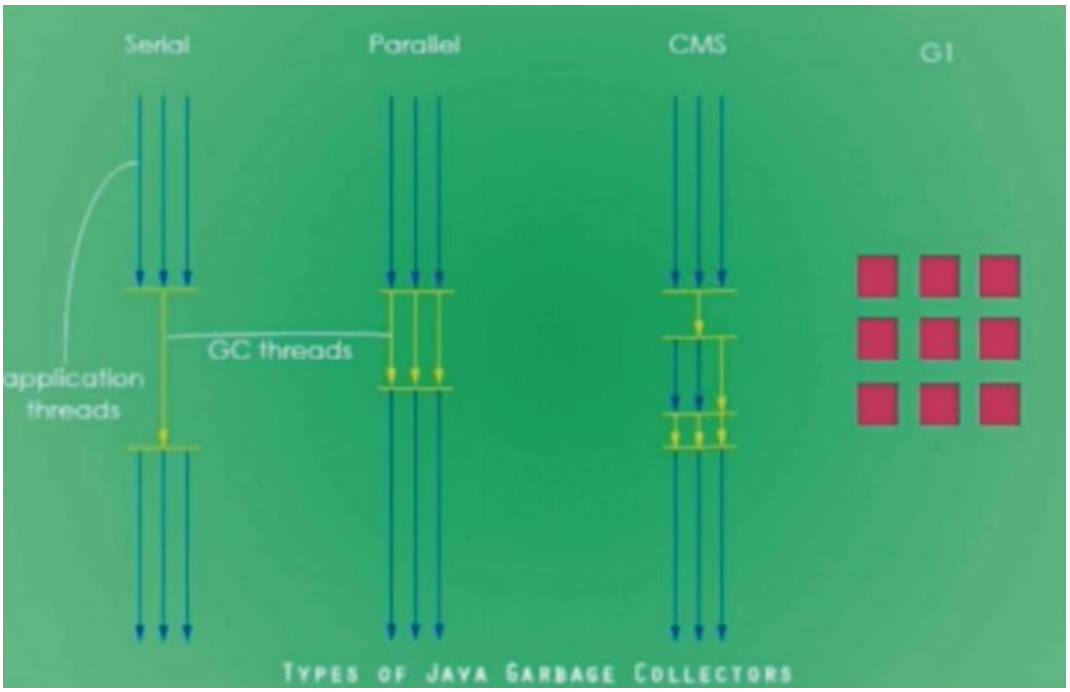
有了虚拟机， 就一定需要收集垃圾的机制， 这就是Garbage Collection， 对应的产品我们称为Garbage Collector。

- 1999年随JDK 1.3.1一起来的是串行方式的SerialGC， 它是第一款GC。Par New垃圾收集器是Serial收集器的多线程版本
- 2002年2月26日， Parallel GC和Concurrent Mark Sweep GC跟随JDK 1.4.2一起发布
- Parallel GC在JDK 6之后成为HotSpot默认GC。
- 2012年， 在JDK 1.7u 4版本中， G 1可用。
- 2017年， JDK 9中G 1变成默认的垃圾收集器， 以替代CMS。
- 2018年3月， JDK 10中G 1垃圾回收器的并行完整垃圾回收， 实现并行性来改善最坏情况下的延迟。
- 2018年9月， JDK 11发布。引入Epsilon垃圾回收器， 又被称为”No-0p(无操作) ”回收器。
  同时， 引入ZGC：可伸缩的低延迟垃圾回收器(Experimental) .
- 2019年3月， JDK 12发布。增强G 1， 自动返回未用堆内存给操作系统。同时， 引入
- Shenandoah GC：低停顿时间的GC(Experimental) .
- 2019年9月， JDK 13发布。增强ZGC， 自动返回未用堆内存给操作系统。
- 2020年3月， JDK 14发布。删除CMS垃圾回收器。扩展ZGC在macos和Windows上的应用

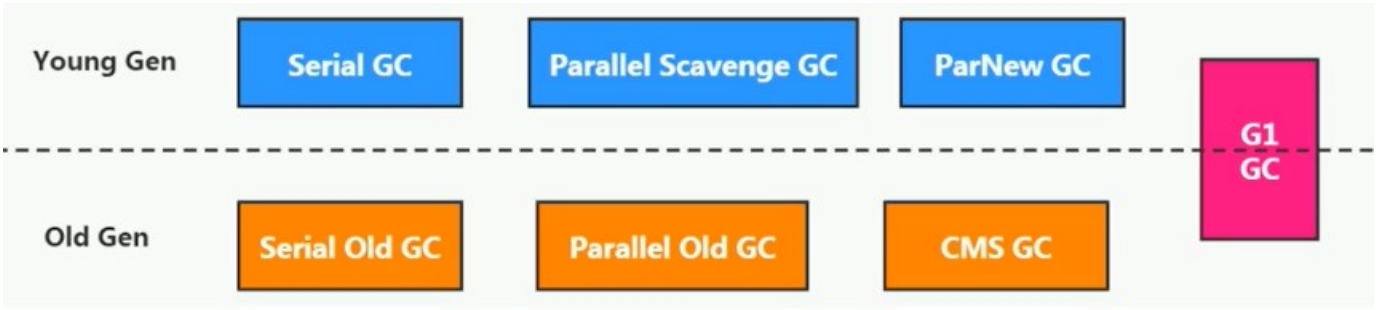
串行回收器：Serial、Serial Old

并行回收器：ParNew、Parallel Scavenge、Parallel Old

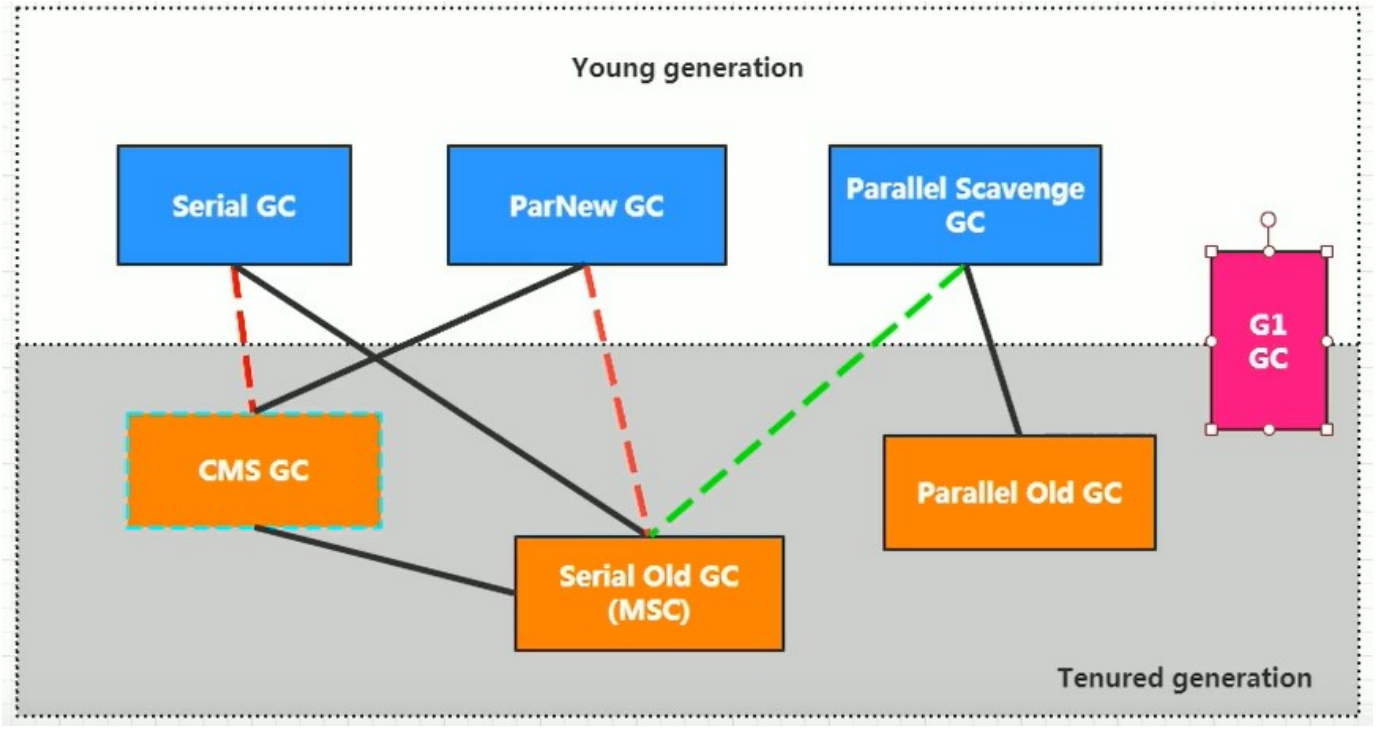
并发回收器：CMS、G1



垃圾回收器的组合关系



- 新生代收集器: Serial、ParNew、Parallel Scavenge ;
- 老年代收集器: Serial Old、Parallel Old、 CMS ;
- 整堆收集器: G1;



JDK8 之前 虚线,都当成 实线 就可以了;

1. 两个收集器间有连线，表明它们可以搭配使用:
   - Serial/Serial old、
   - Serial/CMS、
   - Par New/Serial old、
   - Par New/CMS、
   - Parallel Scavenge/Serial old、
   - Parallel Scavenge/Parallel old、
   - G 1;
2. 其中Serial old作为CMS出现”Concurrent Mode Failure”失败的后备预案。
3. (红色虚线) 由于维护和兼容性测试的成本， 在JDK 8时将Serial+CMS、Par New+Serial old这两个组合声明为废弃(JEP 173)，
   并在JDK 9中完全取消了这些组合的支持(JEP 214)， 即：移除。
4. (绿色虚线) JDK 14中：弃用Parallel Scavenge和Serial old GC组合(JEP 366)
5. (青色虚线) JDK 14中：删除CMS垃圾回收器(JEP 363)

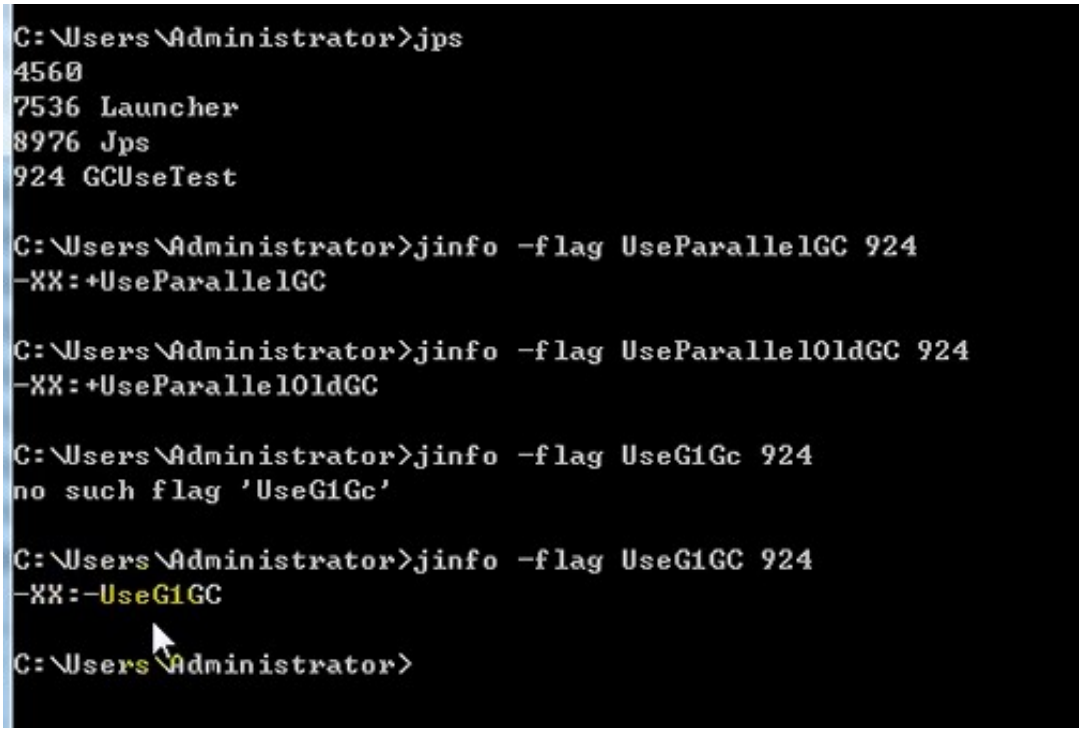
- 为什么要有很多收集器， 一个不够吗?因为Java的使用场景很多， 移动端，服务器

等。
所以就需要针对不同的场景，提供不同的垃圾收集器，提高垃圾收集的性能。

- 虽然我们会对各个收集器进行比较，但并非为了挑选一个最好的收集器出来。
没有一种放之四海皆准、任何场景下都适用的完美收集器存在，更加没有万能的收集器。
所以我们选择的只是对具体应用最合适的收集器。

查看默认垃圾回收器指令

- -XX:+PrintCommandLineFlags: 查看命令行相关参数(包含使用的垃圾收集器)
- 使用命令行指令: jinfo -flag 相关垃圾回收器参数 进程ID



代码示例

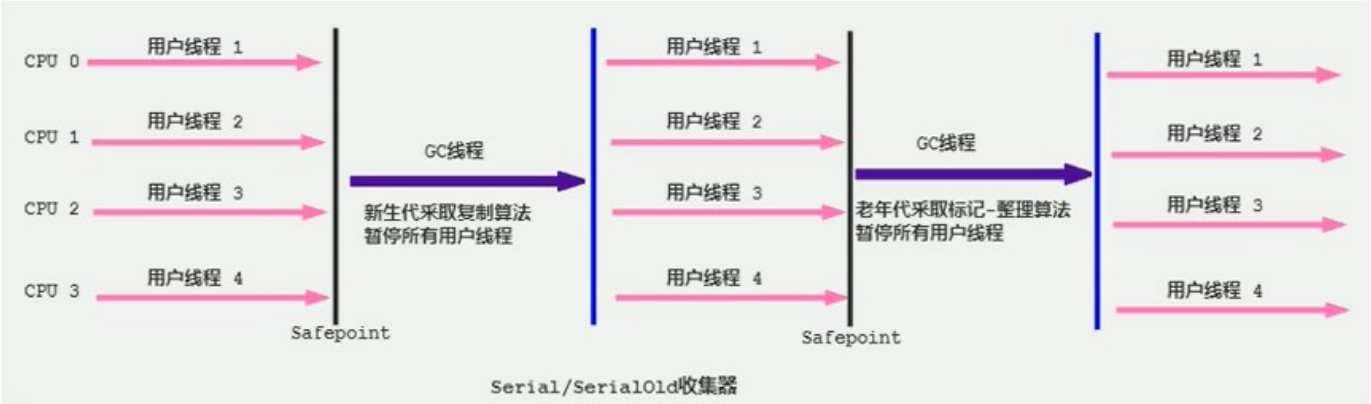
```
import java.util.ArrayList;

/**
 *  -XX:+PrintCommandLineFlags
 *
 *  -XX:+UseSerialGC:表明新生代使用Serial GC ，同时老年代使用Serial Old GC
 *
```

```
 *  -XX:+UseParNewGC：标明新生代使用ParNew GC
 *
 *  -XX:+UseParallelGC:表明新生代使用Parallel GC
 *  -XX:+UseParallelOldGC : 表明老年代使用 Parallel Old GC
 *  说明：二者可以相互激活
 *
 *  -XX:+UseConcMarkSweepGC：表明老年代使用CMS GC。同时，年轻代会触发对ParNew 的使用
 * @author shkstart  shkstart@126.com
 * @create 2020  0:10
 */
public class GCUseTest {
    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();

        while(true){
            byte[] arr = new byte[100];
            list.add(arr);
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Serial回收器: 串行回收

- Serial收集器是最基本、历史最悠久的垃圾收集器了。JDK 1.3之前回收新生代唯一的选择。
- Serial收集器作为HotSpot中Client模式下的默认新生代垃圾收集器。
- Serial收集器采用复制算法、串行回收和”Stop-the-world”机制的方式执行内存回收。
- 除了年轻代之外， Serial收集器还提供用于执行老年代垃圾收集的Serial Old收集器。Serial Old收集器同样也采用了串行回收和”Stop the World”机制， 只不过内存回收算法使用的是标记-压缩算法。

- Serial old是运行在Client模式下默认的老年代的垃圾回收器
- Serialold在Server模式下主要有两个用途：
  ①与新生代的Parallel Scavenge配合使用
  ②作为老年代CMS收集器的后备垃圾收集方案



这个收集器是一个单线程的收集器

但它的“单线程”的意义并不仅仅**说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作**，

更重要的是在它进行垃圾收集时， **必须暂停其他所有的工作线程，** 直到它收集结束(Stop The World) 。

优势

- 优势：**简单而高效**(与其他收集器的单线程比)， 对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销， 专心做垃圾收集自然可以获得最高的单线程收集效率。
  - 运行在client模式下的虚拟机是个不错的选择。
- 在用户的桌面应用场景中，可用内存一般不大(几十MB至一两百MB)，可以在较短时间内完成垃圾收集(几十ms至一百多ms)，只要不频繁发生，使用串行回收器是可以接受的。
- 在HotSpot虚拟机中， 使用-XX：+UseSerialGC 参数可以指定年轻代和老年代都使用串行收集器。
  - 等价于新生代用Serial GC， 且老年代用Serial Old GC

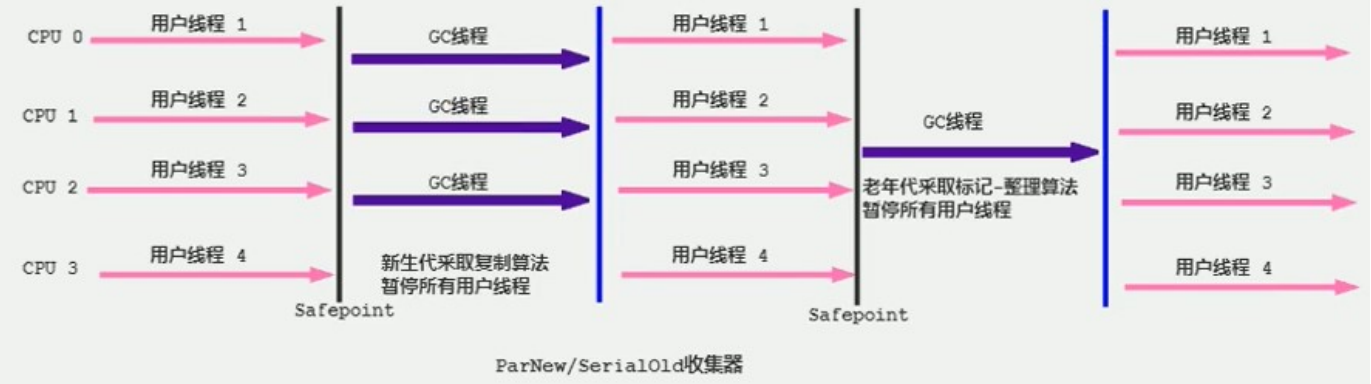
总结:

这种垃圾收集器大家了解， 现在已经不用串行的了。而且在限定单核cpu才可以用。现在都不是单核的了。

对于交互较强的应用而言， 这种垃圾收集器是不能接受的。一般在Java web应用程序中是不会采用串行垃圾收集器的。

ParNew回收器: 并行回收

- 如果说Serial GC是年轻代中的单线程垃圾收集器， 那么Par New收集器则是Serial收集器的多线程版本。
    - Par是Parallel的缩写， New：只能处理的是新生代
- Par New收集器除了采用**并行回收**的方式执行内存回收外， 两款垃圾收集器之间几乎没有任何区别。
  Par New收集器在年轻代中同样也是**采用复制算法、"Stop-the-World"机制。**
- Par New是很多JVM运行在Server模式下新生代的默认垃圾收集器。



- 对于新生代，回收次数频繁，使用并行方式高效。
- 对于老年代，回收次数少，使用串行方式节省资源。(CPU并行需要切换线程，串行可以省去切换线程的资源)

比较

由于Par New收集器是基于并行回收， 那么是否可以断定Par New收集器的回收效率在**任何场景下**都会比Serial收集器更高效?

- Par New收集器运行在多CPU的环境下， 由于可以充分利用多CPU、多核心等物理硬件资源优势，
  可以更快速地完成垃圾收集，提升程序的吞吐量。
- 但是在单个CPU的环境下， Par New收集器不比Serial收集器更高效。

虽然Serial收集器是基于串行回收， 但是由于CPU不需要频繁地做任务切换，
因此可以有效避免多线程交互过程中产生的一些额外开销。

因为除Serial外， 目前只有Par New GC能与CMS收集器配合工作;

设置参数

- 在程序中， 开发人员可以通过选项”-xx：+Use Par New GC”手动指定使用Par New收集器执行内存回收任务。
  它表示年轻代使用并行收集器， 不影响老年代。
- -XX：ParallelGCThreads限制线程数量， **默认开启和CPU数据相同的线程数**。

Parallel回收器: 吞吐量优先

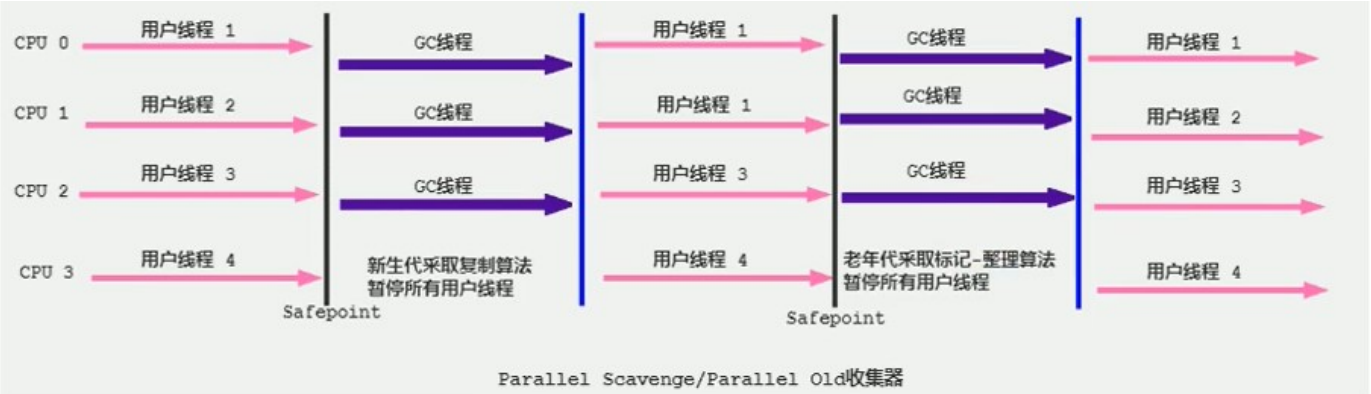
- HotSpot的年轻代中除了拥有Par New收集器是基于并行回收的以外，
  Parallel Scavenge收集器同样也采用了复制算法、并行回收和”Stop the World”机制。

那么Parallel收集器的出现是否多此一举?

- 和Par New收集器不同， Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量(Throughput) ，
  它也被称为吞吐量优先的垃圾收集器。
- **自适应调节策略**也是Parallel Scavenge与Par New一个重要区别。

吞吐量

- 高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。
  因此，常见在服务器环境中使用。**例如，那些执行批量处理、订单处理、工资支付、科学计算的应用程序。**
- Parallel 收集器在JDK 1.6时提供了用于执行老年代垃圾收集的Parallel Old收集器， 用来代替老年代的Serial old收集器。
- Parallel old收集器采用了**标记-压缩算法**， 但同样也是基于并行回收和”Stop-the-World”机制。



优势

- 在程序吞吐量优先的应用场景中，Parallel收集器和Parallel Old收集器的组合， 在Server模式下的内存回收性能很不错。
- 在Java 8中， 默认是此垃圾收集器。

参数设置

使用参数

- **-XX：+UseParallelGC** 手动指定年轻代使用Parallel并行收集器执行内存回收任务。
- **-XX：+UseParallelOldGC** 手动指定老年代都是使用并行回收收集器。
  - 分别适用于新生代和老年代。默认jdk 8是开启的。
  - 上面两个参数，**默认开启一个，另一个也会被开启。(互相激活)**
- **-XX：ParallelGCThreads** 设置年轻代并行收集器的线程数。一般地，最好与CPU数量相等， 以避免过多的线程数影响垃圾收集性能。
  - 在默认情况下，当CPU 数量小于8个，Parallel GC Threads的值等于CPU数量。
  - 当CPU数量大于8个，Parallel GC Threads的值等于3+[5*CPU Count] /8]

一组"矛盾"参数

- **-XX：MaxGCPauseMillis** 设置垃圾收集器最大停顿时间(即STW的时间)。单位是毫秒。
  - 为了尽可能地把停顿时间控制在Max GC Pause Mills以内， 收集器在工作时会调整Java堆大小或者其他一些参数。
  - 对于用户来讲，停顿时间越短体验越好。但是在服务器端，我们注重高并发， 整体的吞吐量。所以服务器端适合Parallel， 进行控制。

    - **该参数使用需谨慎。**
- **-XX：GCTimeRatio**垃圾收集时间占总时间的比例(=1/(N+1))。
  用于衡量吞吐量的大小。
    - 取值范围(0，100)。默认值99，也就是垃圾回收时间不超过1%。
    - 与前一个-XX：MaxGCPauseMillis参数有一定矛盾性。暂停时间越长， Radio参数就容易超过设定的比例。

自适应调节策略参数

- **-XX：+UseAdaptiveSizePolicy** 设置Parallel Scavenge收集器具有自适应调节策略
    - 在这种模式下， 年轻代的大小、Eden和Survivor的比例、**(说的是 8:1:1, 实际 6:1:1)**
      晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。
    - 在手动调优比较困难的场合，可以直接使用这种自适应的方式，
      仅指定虚拟机的最大堆、目标的吞吐量(GC Time Ratio) 和停顿时间(Max GC Pause Mills)， 让虚拟机自己完成调优工作。

CMS回收器: 低延迟

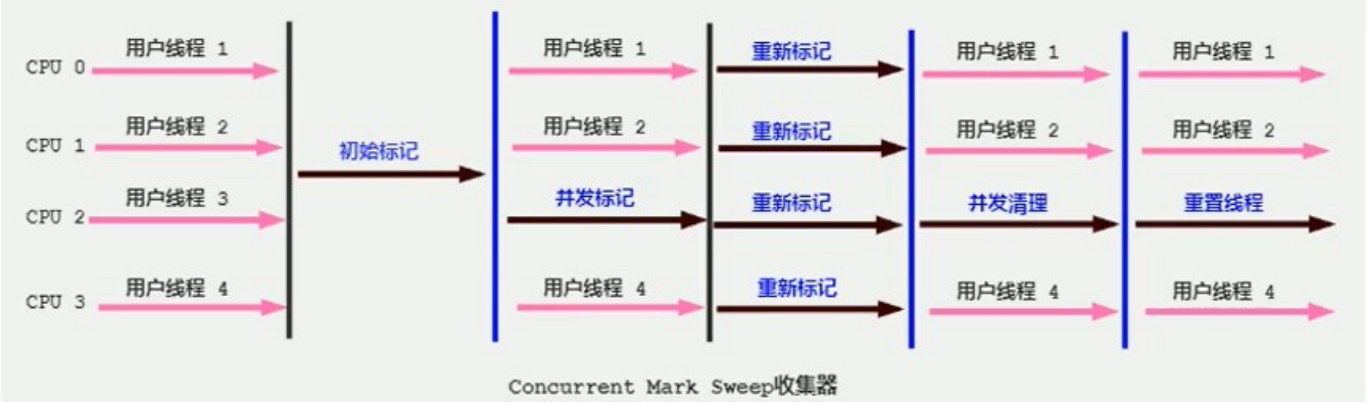
CMS概述

- 在JDK 1.5时期， HotSpot推出了一款在**强交互应用**中几乎可认为有划时代意义的垃圾收集器：CMS(Concurrent-Mark-Sweep) 收集器， **这款收集器是HotSpot虚拟机中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同时工作**。
- CMS收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间。停顿时间越短(低延迟)就越适合与用户交互的程序，良好的响应速度能提升用户体验。
    - **目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，**
      这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。
    - CMS收集器就非常符合这类应用的需求。

- CMS的垃圾收集算法采用标记-清除算法， 并且也会”Stop-the-world”

- 不幸的是， CMS作为老年代的收集器， 却无法与JDK 1.4. 0中已经存在的新生代收集器Parallel Scavenge配合工作，
  所以在JDK 1.5中使用CMS来收集老年代的时候， **新生代只能**选择Par New或者Serial收集器中的一个。
- 在G 1出现之前， CMS使用还是非常广泛的。一直到今天， 仍然有很多系统使用CMS GC。

CMS工作原理



初始化标记时间很短, 并发标记时间相对较长

CMS整个过程比之前的收集器要复杂， 整个过程分为4个主要阶段， 即初始标记阶段、并发标记阶段、重新标记阶段和并发清除阶段。

- **初始标记(Initial-Mark) 阶段**：在这个阶段中， 程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，
  这个阶段的主要任务仅仅只是标记出GC Roots能直接关联到的对象。
  一旦标记完成之后就会恢复之前被暂停的所有应用线程。由于直接关联对象比较小，所以这里的速度非常快。
- **并发标记(Concurrent-Mark) 阶段**：从GC Roots的直接关联对象开始遍历整个对象图的过程，
  这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。
- **重新标记(Remark) 阶段**：由于在并发标记阶段中， 程序的工作线程会和垃圾收集线程同时运行或者交叉运行，
  因此为了修正并发标记期间，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

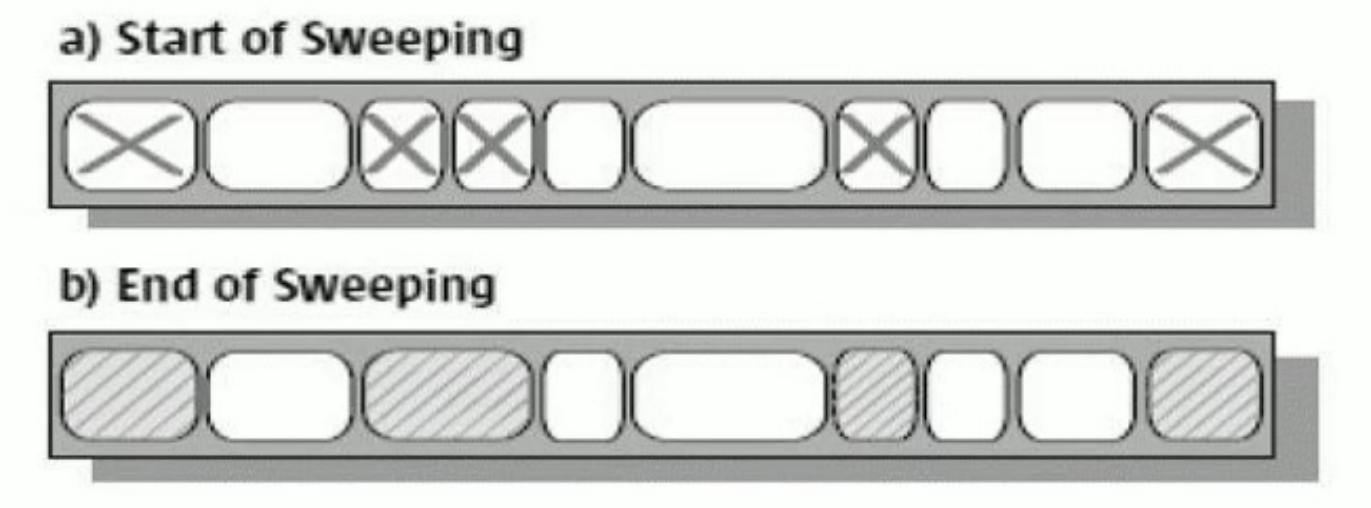
的标记记录，
这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。

- **并发清除(Concurrent-Sweep) 阶段**：此阶段清理删除掉标记阶段判断的已经死亡的对象，释放内存空间。
  由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的

CMS特点与弊端分析

- 尽管CMS收集器采用的是并发回收(非独占式)， 但是在其初始化标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程， 不过暂停时间并不会太长， 因此可以说明目前所有的垃圾收集器都做不到完全不需要“Stop-the-World”， 只是尽可能地缩短暂停时间。
- 由于最耗费时间的并发标记与并发清除阶段都不需要暂停工作，所以整体的回收是低停顿的。
- 另外， 由于在垃圾收集阶段用户线程没有中断， 所以在CMS回收过程中， 还应该确保应用程序用户线程有足够的内存可用。因此， CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，而是当堆内存使用率达到某一阈值时，便开始进行回收， 以确保应用程序在CMS工作过程中依然有足够的空间支持应用程序运行。要是CMS运行期间预留的内存无法满足程序需要， 就会出现一次“Concurrent Mode Failure”失败， 这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

CMS收集器的垃圾收集算法采用的是标记-清除算法， 这意味着每次执行完内存回收后，由于被执行内存回收的无用对象所占用的内存空间极有可能是不连续的一些内存块， 不可避免地将会产生一些内存碎片。那么CMS在为新对象分配内存空间时， 将无法使用指针碰撞(Bump thePointer) 技术， 而只能够选择空闲列表(FreeList) 执行内存分配。



有人会觉得既然Mark Sweep会造成内存碎片， 那么为什么不把算法换成 Mark Compact 呢?

答案其实很简答， 因为当并发清除的时候， 用Compact整理内存的话，
原来的用户线程使用的内存还怎么用呢? 要保证用户线程能继续执行，前提的它运行的资源不受影响嘛。
MarkCompact更适合“Stop the World”这种场景下使用

CMS的优点:

- 并发收集
- 低延迟

CMS的弊端:

1. 会产生内存碎片，导致并发清除后，用户线程可用的空间不足。
   在无法分配大对象的情况下， 不得不提前触发Full GC。
2. CMS收集器对CPU资源非常敏感。在并发阶段， 它虽然不会导致用户停顿，
   但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量会降低。
3. CMS收集器无法处理浮动垃圾。可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。
   在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的， 那么在并发标记阶段如果产生新的垃圾对象，
   CMS将无法对这些垃圾对象进行标记，最终会导致这些新产生的垃圾对象没有被及时回收，
   从而只能在下一次执行GC时释放这些之前未被回收的内存空间。

CMS参数设置

- **-XX:+UseConcMarkSweepGC** 手动指定使用CMS收集器执行内存回收任务。
  - 开启该参数后会自动将-XX+UseParNewGC打开。即：ParNew(Young区用) +CMS(0ld区用) +Serialold的组合。
- **-XX：CMSInitiatingOccupanyFraction** 设置堆内存使用率的阈值，一旦达到该阈值，便开始进行回收。
  - JDK 5及以前版本的默认值为68， 即当老年代的空间使用率达到68号时， 会执行一次CMS回收。**JDK 6及以上版本默认值为92%**
  - 如果内存增长缓慢， 则可以设置一个稍大的值， 大的阈值可以有效降低CMS的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。
    因此**通过该选项便可以有效降低FullGC的执行次数。**
- **-XX：+UseCMSCompactAtFullCollection** 用于指定在执行完Full GC后对内存空间进行压缩整理，以此避免内存碎片的产生。
  不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间变得更长了。
- **-XX: CMSFullGCsBeforeCompaction** 设置在执行多少次FullGC后对内存空间进行压缩整理。
- **-XX: ParallelCMSThreads** 设置CMS的线程数量。
  - CMS默认启动的线程数是(Parallel GC Threads+3) /4，Parallel GC Threads是年轻代并行收集器的线程数。
    当CPU资源比较紧张时， 受到CMS收集器线程的影响， 应用程序的性能在垃圾回收阶段可能会非常糟糕。

CMS小结

HotSpot有这么多的垃圾回收器， 那么如果有人问， Serial GC、Parallel GC、Concurrent Mark Sweep GC这三个GC有什么不同呢?

请记住以下口令:

- 如果你想要最小化地使用内存和并行开销， 请选Serial GC;
- 如果你想要最大化应用程序的吞吐量， 请选Parallel GC;

- 如果你想要最小化GC的中断或停顿时间(低延迟)， 请选CMS GC;

后续变化

- JDK 9新特性：CMS被标记为Deprecate了(JEP 291)
  - 如果对JDK 9及以上版本的HotSpot虚拟机使用参数 -XX:+UseConcMarkSweepGc来开启CMS收集器的话， 用户会收到一个警告信息， 提示CMS未来将会被废弃。
- JDK 14新特性：删除CMS垃圾回收器(JEP 363)
  - 移除了CMS垃圾收集器， 如果在JDK 14中使用-xx：+Use Conc Mark Sweep Gc的话， JVM不会报错， 只是给出一个warning信息， 但是不会exit。JVM会自动回退以默认GC方式启动JVM

JDK 14 报错信息:

Open JDK 64-Bit Server VM warning：Ignoring option Use Conc Mark Sweep GC；

support was removed in 14. 0

and the VM willcontinue execution using the defaultcollector.

G1回收器: 区域化分代式

概述

既然我们已经有了前面几个强大的GC， 为什么还要发布Garbage First(G 1)GC?

- 原因就在于应用程序所应对的业务越来越庞大、复杂，用户越来越多，没有GC就不能保证应用程序正常进行，
  而经常造成STW的GC又跟不上实际的需求， 所以才会不断地尝试对GC进行优化。
  G 1(Garbage-First) 垃圾回收器是在Java 7 update4之后引入的一个新的垃圾回收器，是当今收集器技术发展的最前沿成果之一。
- 与此同时，为了适应现在不断扩大的内存和不断增加的处理器数量，进一步降低暂停时间(pausetime)，同时兼顾良好的吞吐量。
- 官方给G1设定的目标是在延迟可控的情况下获得尽可能高的吞吐量，所以才担当起“全功能收集器”的重任与期望。

为什么名字叫做Garbage First (G1)呢?

- 因为G 1是一个并行回收器， 它把堆内存分割为很多不相关的区域(Region) (物理上不连续的) 。
  使用不同的Region来表示Eden、幸存者0区， 幸存者1区， 老年代等。
- G1 GC有计划地避免在整个Java堆中进行全区域的垃圾收集。
  G 1 跟踪各个Region里面的垃圾堆积的价值大小(回收所获得的空间大小以及回收所需时间的经验值),
  在后台维护一个优先列表， 每次根据允许的收集时间， 优先回收价值最大的Region。
- 由于这种方式的侧重点在于回收垃圾最大量的区间(Region) ， 所以我们给G 1一个名字：垃圾优先(GarbageFirst) 。

G 1(Garbage-First) 是一款面向服务端应用的垃圾收集器， 主要针对配备多核CPU及大容量内存的机器， 以极高概率满足GC停顿时间的同时， 还兼具高吞吐量的性能特征。

在JDK 1.7版本正式启用， 移除了Experimental的标识， 是JDK 9以后的默认垃圾回收器， 取代了CMS回收器以及Parallel+Parallel old组合。被Oracle官方称为“全功能的垃圾收集器”。

与此同时， CMS已经在JDK 9中被标记为废弃(deprecated) 。在jdk 8中还不是默认的垃圾回收器， 需要使用-XX:+UseG1GC来启用。

G1回收器的优势

与其他GC收集器相比，G1使用了全新的分区算法，其特点如下所示：

并行与并发

- 并行性：G1在回收期间，可以有多个GC线程同时工作，有效利用多核计算能力。此时用户线程STW
- 并发性：G1拥有与应用程序交替执行的能力，部分工作可以和应用程序同时执行，因此，一般来说，不会在整个回收阶段发生完全阻塞应用程序的情况

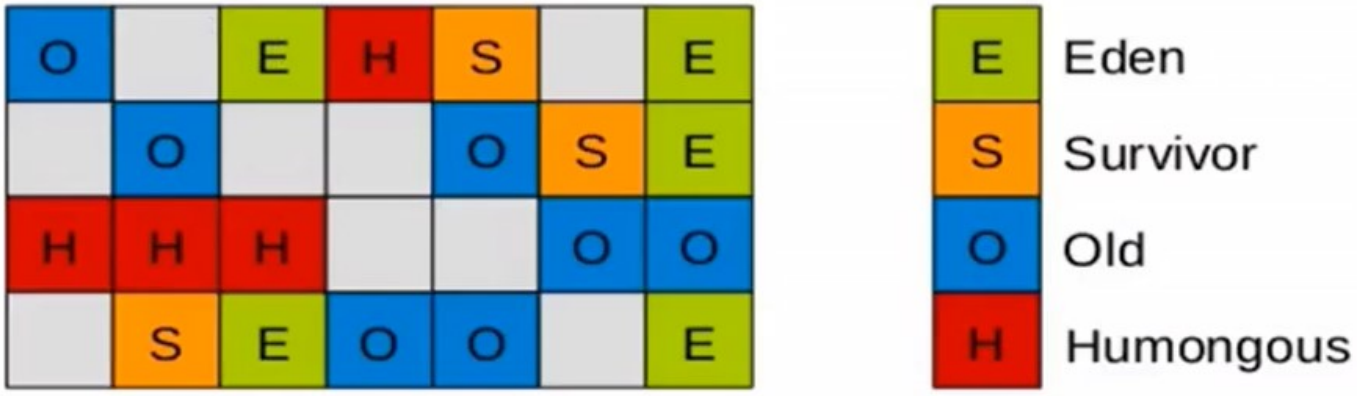
分代收集

- 从分代上看，G1依然属于分代型垃圾回收器，它会区分年轻代和老年代，年轻代依然

有Eden区和Survivor区。
但从堆的结构上看， 它不要求整个Eden区、年轻代或者老年代都是连续的，也不再坚持固定大小和固定数量。

- 将堆空间分为若干个区域(Region)， 这些区域中包含了逻辑上的年轻代和老年代。
- 和之前的各类回收器不同，它同时兼顾年轻代和老年代。对比其他回收器，或者工作在年轻代，或者工作在老年代；

分区思路



空间整合

- CMS：“标记-清除”算法、内存碎片、若干次GC后进行一次碎片整理
- G 1将内存划分为一个个的region。
  内存的回收是以region作为基本单位的。Region之间是复制算法， 但整体上实际可看作是标记-压缩(Mark-Compact)算法，
  两种算法都可以避免内存碎片。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。
  尤其是当Java堆非常大的时候， G 1的优势更加明显。

可预测的停顿时间模型(即：软实时soft real-time)

- 这是G1相对于CMS的另一大优势，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，消耗在垃圾收集上的时间不得超过N毫秒。
  - 由于分区的原因，G1可以只选取部分区域进行内存回收，
    这样宿小了回收的范围，因此对于全局停顿情况的发生也能得到较好的控制。

- G1跟踪各个Region里面的垃圾堆积的价值大小(回收所获得的空间大小以及回收所需时间的经验值)，
  在后台维护一个优先列表，每次根据允许的收集时间， 优先回收价值最大的Region。
  保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。
- 相比于CMS GC，G 1未必能做到CMS在最好情况下的延时停顿， 但是最差情况要好很多。

G1回收器的劣势

- 相较于CMS， G 1还不具备全方位、压倒性优势。比如在用户程序运行过程中，G 1无论是为了垃圾收集产生的内存占用(Footprint) 还是程序运行时的额外执行负载(Overload) 都要比CMS要高。
- 从经验上来说， 在小内存应用上CMS的表现大概率会优于G 1，而G 1在大内存应用上则发挥其优势。平衡点在6-8GB之间。

G1回收器的参数设置

- **-XX：+UseG1GC** 手动指定使用G1收集器执行内存回收任务。
- **-XX：G1HeapRegionSize** 设置每个Region的大小。
  值是2的幂， 范围是1MB到32MB之间， 目标是根据最小的Java堆大小划分出约2048个区域。默认是堆内存的1/2000。
- **-XX：MaxGCPauseMillis** 设置期望达到的最大GC停顿时间指标(JVM会尽力实现，但不保证达到)。默认值是200ms
  说明: 太小, 可能回收少, 导致内存腾出来的空间就少, 容易满 反而进行了 Full GC
- **-XX：ParallelGCThread** 设置sTW工作线程数的值。最多设置为8
- **-XX：ConcGCThreads** 设置并发标记的线程数。将n设置为并行垃圾回收线程数(Parallel GC Threads) 的1/4左右。
- **-XX：InitiatingHeapOccupancyPercent** 设置触发并发GC周期的Java堆占用率阈值。超过此值，就触发GC。默认值是45%。

G1常见操作步骤

G 1的设计原则就是简化JVM性能调优， 开发人员只需要简单的三步即可完成调优：

- 第一步：开启G1垃圾收集器
- 第二步：设置堆的最大内存
- 第三步：设置最大的停顿时间

G 1中提供了三种垃圾回收模式：Young Gc、Mixed GC和Full GC， 在不同的条件下被触发。

G1回收器的适用场景

- 面向服务端应用，针对具有大内存、多处理器的机器。(在普通大小的堆里表现并不惊喜)
- 最主要的应用是需要低GC延迟，并具有大堆的应用程序提供解决方案；
- 如：在堆大小约6GB或更大时，可预测的暂停时间可以低于0.5秒；
  (G1通过每次只清理一部分而不是全部的Region的增量式清理来保证每次GC停顿时间不会过长) 。

- 用来替换掉JDK 1.5中的CMS收集器；

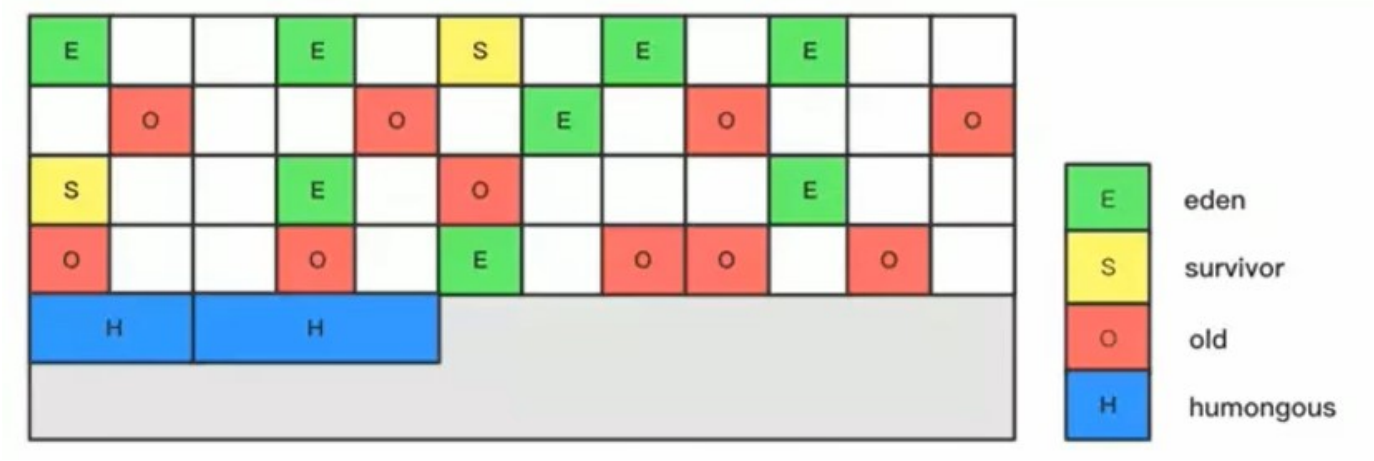
  在下面的情况时， 使用G 1可能比CMS好：

  - ① 超过50%的Java堆被活动数据占用；
  - ② 对象分配频率或年代提升频率变化很大；
  - ③ GC停顿时间过长(长于0.5至1秒)。
- HotSpot垃圾收集器里， 除了G 1以外，
  其他的垃圾收集器使用内置的JVM线程执行GC的多线程操作， 而G1GC可以采用应用线程承担后台运行的GC工作，
  即当JVM的GC线程处理速度慢时，系统会调用应用程序线程帮助加速垃圾回收过程。

G1--分区Region: 化整为零

使用G 1收集器时， 它将整个Java堆划分成约2048个大小相同的独立Region块，
每个Region块大小根据堆空间的实际大小而定， 整体被控制在1MB到32MB之间，且为2的N次幂，即1MB，2MB，4MB，8MB，16MB，32MB。可以通过-XX：G1HeapRegionSize设定。所有的Region大小相同， 且在JVM生命周期内不会被改变。

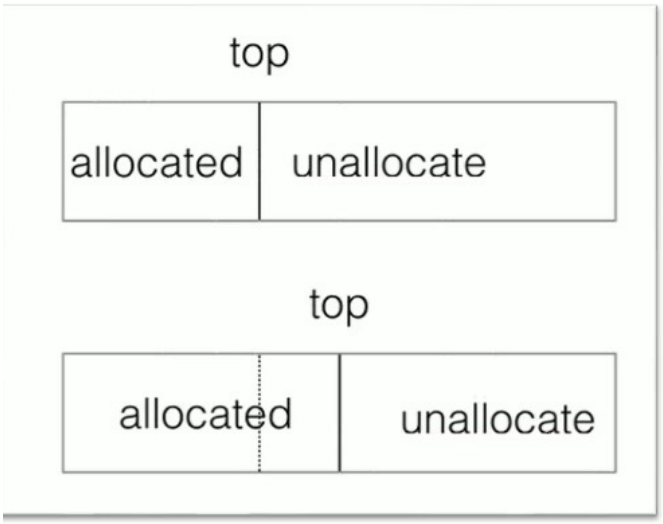
虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，
它们都是一部分Region(不需要连续) 的集合。通过Region的动态分配方式实现逻辑上的连续。



- 一个 region 有可能属于Eden，Survivor或者old/Tenured内存区域。但是一个region只可能属于一个角色。
  图中的E表示该region属于Eden内存区域， s表示属于Survivor内存区域， o表示属于old内存区域。
  图中空白的表示未使用的内存空间。
- G1垃圾收集器还增加了一种新的内存区域，叫做Humongous内存区域，(巨型对象)
  如图中的H块。主要用于存储大对象， 如果超过1.5个region， 就放到H。

设置H的原因:

对于堆中的大对象，默认直接会被分配到老年代，但是如果它是一个短期存在的大对象，就会对垃圾收集器造成负面影响。
为了解决这个问题， G 1划分了一个Humongous区，它用来专门存放大对象。
如果一个H区装不下一个大对象，那么G1会寻找连续的H区来存储。为了能找到连续的H区，有时候不得不启动Full GC。
G 1的大多数行为都把H区作为老年代的**一部分**来看待。



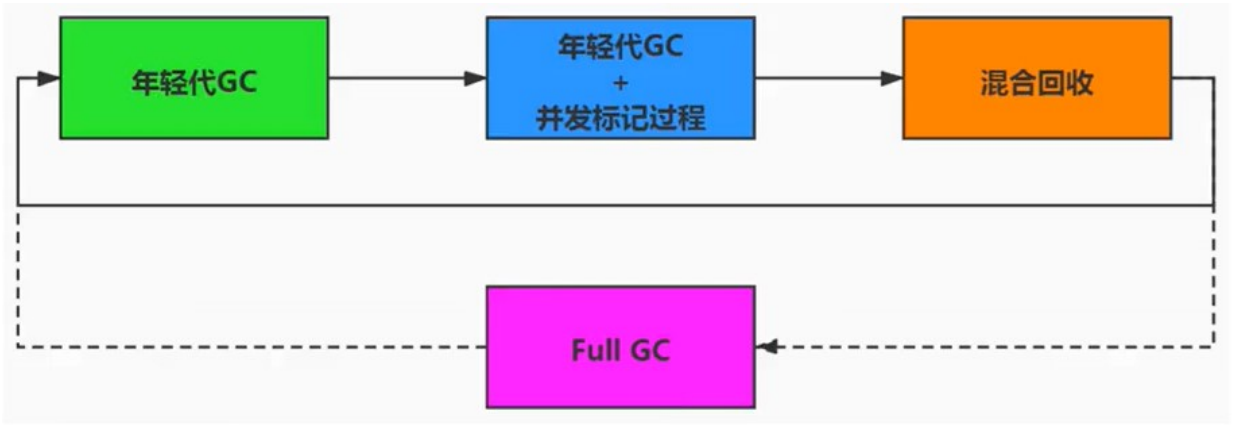
- Bump - the - pointer
即：指针碰撞
- TLAB

G1回收器垃圾回收过程

参考视频: https://www.bilibili.com/video/BV1PJ411n7xZ?p=193

G1　GC的垃圾回收过程主要包括如下三个环节：

- 年轻代GC(Young GC)
- 老年代并发标记过程(Concurrent Marking)
- 混合回收(Mixed GC)
- (如果需要， 单线程、独占式、高强度的Full Gc还是继续存在的。它针对GC的评估失败提供了一种失败保护机制，即强力回收。)



顺时针，young gc -> young gc + concurrent mark-> Mixed GC顺序，
进行垃圾回收。

独占式: STW

- 应用程序分配内存， 当年轻代的Eden区用尽时开始年轻代回收过程； G 1的年轻代收集阶段是一个并行的独占式收集器。
在年轻代回收期，G1GC暂停所有应用程序线程，启动多线程执行年轻代回收。

然后从年轻代区间移动存活对象到Survivor区间或者老年区间， 也有可能是两个区间都会涉及。

- 当堆内存使用达到一定值(默认45%)时，开始老年代并发标记过程。
- 标记完成马上开始混合回收过程。
  对于一个混合回收期，G1GC从老年区间移动存活对象到空闲区间，这些空闲区间也就成为了老年代的一部分。
  和年轻代不同，老年代的G1回收器和其他GC不同，G1的老年代回收器不需要整个老年代被回收，
  一次只需要扫描/回收一小部分老年代的Region就可以了。同时， 这个老年代Region是和年轻代一起被回收的。
- 举个例子：一个Web服务器， Java进程最大堆内存为4G， 每分钟响应1500个请求，每45秒钟会新分配大约2G的内存。
  G1会每45秒钟进行一次年轻代回收，每31个小时整个堆的使用率会达到45%，
  会开始老年代并发标记过程，标记完成后开始四到五次的混合回收。

Remembered Set

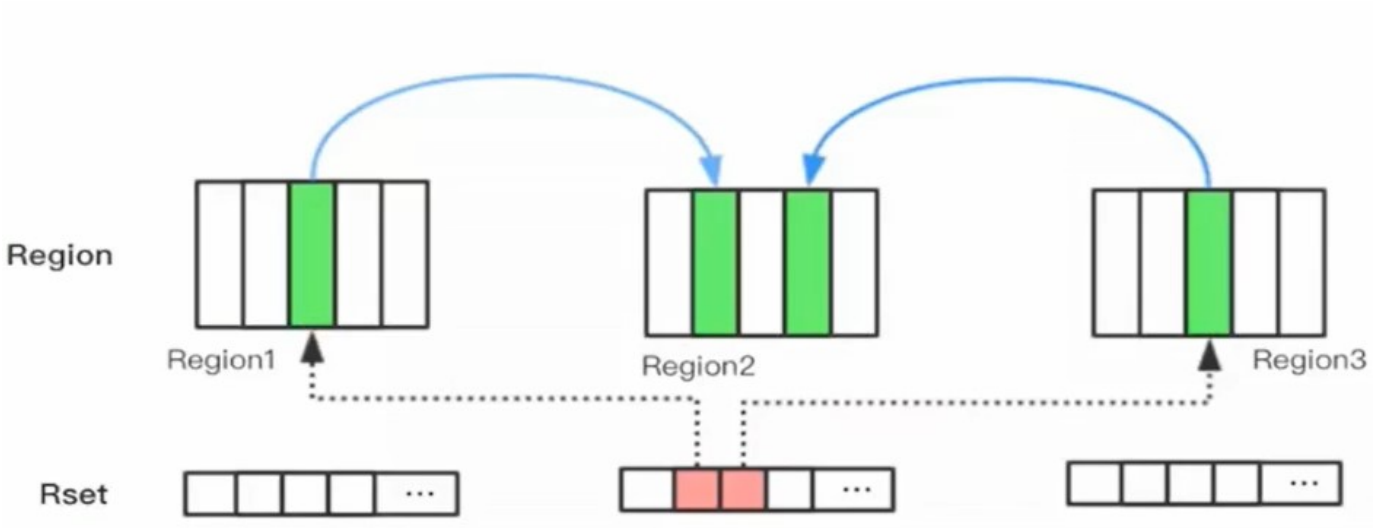
记忆集与写屏障

- 一个对象被不同区域引用的问题
- 一个Region不可能是孤立的， 一个Region中的对象可能被其他任意Region中对象引用，判断对象存活时， 是否需要扫描整个Java堆才能保证准确?
- 在其他的分代收集器，也存在这样的问题(而G1更突出)
- 回收新生代也不得不同时扫描老年代?
- 这样的话会降低Minor GC的效率;

解决方法:

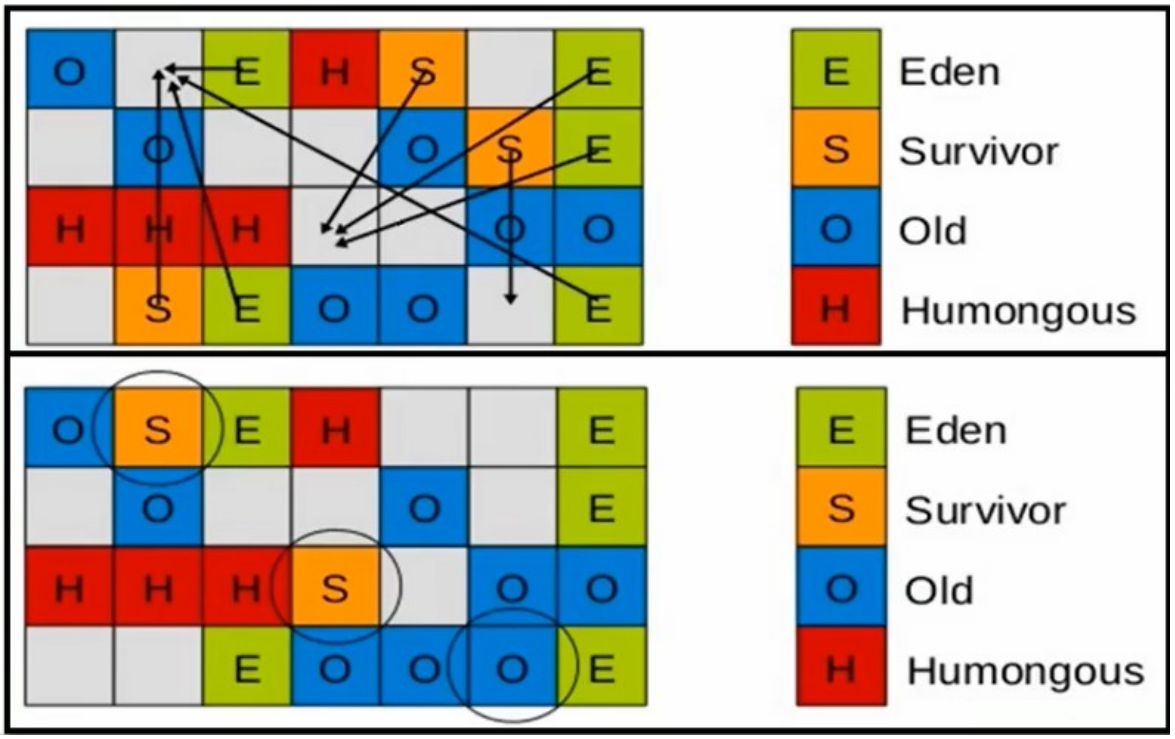
- 无论G 1还是其他分代收集器， JVM都是使用Remembered Set来避免全局扫描:
- 每个Region都有一个对应的Remembered Set;
- 每次Reference类型数据写操作时， 都会产生一个Write Barrier(写屏障)暂时中断操作;
- 然后检查将要写入的引用指向的对象是否和该Reference类型数据在不同的Region(其他收集器：检查老年代对象是否引用了新生代对象);

- 如果不同， 通过Card Table把相关引用信息记录到引用指向对象的所在Region对应的Remembered Set中；
- 当进行垃圾收集时， 在GC根节点的枚举范围加入Remembered Set； 就可以保证不进行全局扫描，也不会有遗漏。



G1回收过程一: 年轻代GC

- JVM启动时， G 1先准备好Eden区， 程序在运行过程中不断创建对象到Eden区， 当Eden空间耗尽时， G 1会启动一次年轻代垃圾回收过程。
- **年轻代垃圾回收只会回收Eden区和Survivor区。**
- Y GC时， 首先G 1停止应用程序的执行(Stop-The-World)， G 1创建回收集(Collection Set)， 回收集是指需要被回收的内存分段的集合， 年轻代回收过程的回收集包含年轻代Eden区和Survivor区所有的内存分段。



然后开始如下回收过程：

第一阶段，扫描根。

根是指static变量指向的对象， 正在执行的方法调用链条上的局部变量等。根引用连同R Set记录的外部引用作为扫描存活对象的入口。

第二阶段，更新R Set。

处理dirty card queue(见备注) 中的card，更新R Set。此阶段完成后， R Set可以准确的反映老年代对所在的内存分段中对象的引用。

第三阶段， 处理R Set。

识别被老年代对象指向的Eden中的对象， 这些被指向的Eden中的对象被认为是存活的对象。

第四阶段，复制对象。

此阶段， 对象树被遍历， Eden区内存段中存活的对象会被复制到Survivor区中空的内存分段，Survivor区内存段中存活的对象如果年龄未达阈值， 年龄会加1， 达到阀值会被会被复制到old区中空的内存分段。如果Survivor空间不够， Eden空间的部分数据会直接晋升到老年代空间。

第五阶段，处理引用。

处理Soft， Weak， Phantom，Final， JNI Weak等引用。最终Eden空间的数据为空， GC停止工作，而目标内存中的对象都是连续存储的，没有碎片，所以复制过程可以达到内存整理的效果，减少碎片。让了

备注:

dirty card queue

对于应用程序的引用赋值语句object.field=object， JVM会在之前和之后执行特殊的操作以在dirty card queue中入队一个保存了对象引用信息的card。在年轻代回收的时候， G 1 对Dirty Card Queue中所有的card进行处理， 以更新R Set， 保证R Set实时准确的反映引用关系。

那为什么不在引用赋值语句处直接更新R Set呢?
这是为了性能的需要， R Set的处理需要线程同步， 开销会很大， 使用队列性能会好很多。

G1回收过程二: 并发标记过程

1. **初始标记阶段**：标记从根节点直接可达的对象。这个阶段是STW的， 并且会触发一次年轻代GC。
2. **根区域扫描(Root Region Scanning)**： G 1 GC扫描Survivor区直接可达的老年代区域对象， 并标记被引用的对象。这一过程必须在young GC之前完成。
3. **并发标记(Concurrent Marking)**： 在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被young GC中断。在并发标记阶段， 若发现区域对象中的所有对象都是垃圾，那这个区域会被立即回收。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。
4. **再次标记(Remark)**： 由于应用程序持续进行， 需要修正上一次的标记结果。是STW的。G 1中采用了比CMS更快的初始快照算法：snapshot-at-the-beginning(SATB) 。
5. **独占清理(cleanup， STW)**： 计算各个区域的存活对象和GC回收比例， 并进行排序，识别可以混合回收的区域。为下阶段做铺垫。是STW的。
    a. 这个阶段并不会实际上去做垃圾的收集
6. **并发清理阶段**：识别并清理完全空闲的区域。

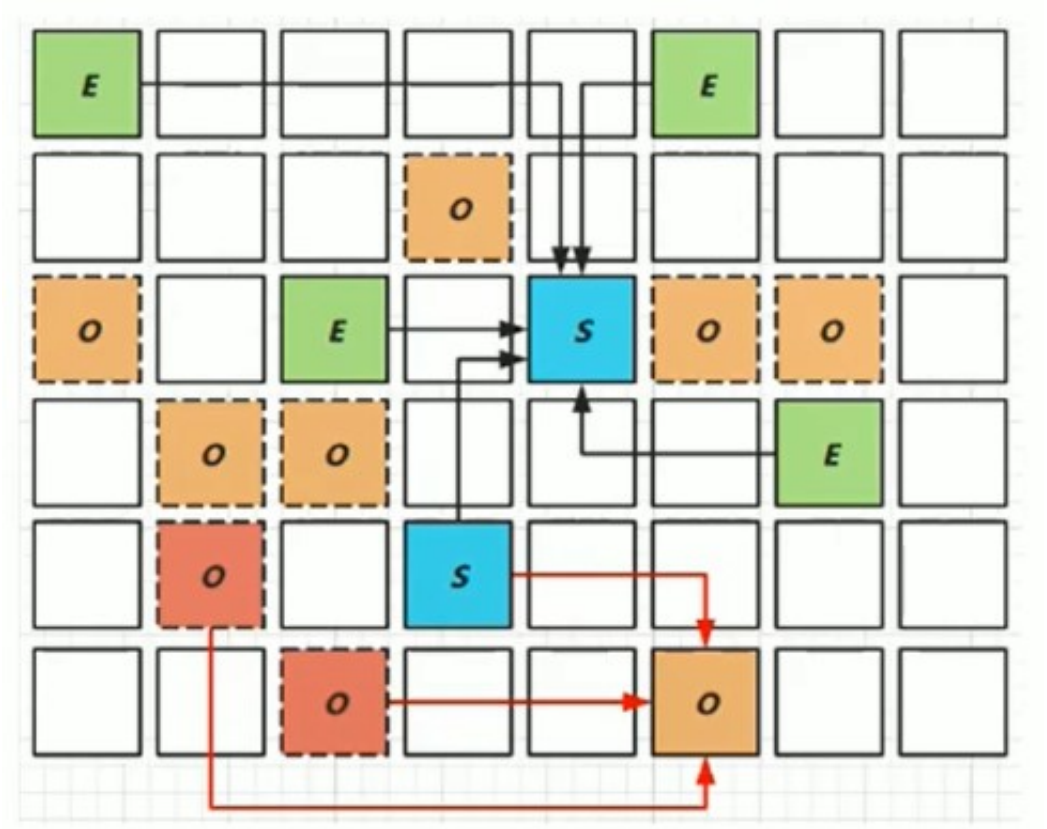
G1回收过程三: 混合回收

当越来越多的对象晋升到老年代old region时， 为了避免堆内存被耗尽，虚拟机会触发一个混合的垃圾收集器，即Mixed GC，

该算法并不是一个old GC， 除了回收整个Young Region，还会回收一部分的Old Region。
这里需要注意：**是一部分老年代，而不是全部老年代**。
可以选择哪些Old Region进行收集， 从而可以对垃圾回收的耗时时间进行控制。也要注意的是Mixed GC并不是Full GC。

过程

- 并发标记结束以后，老年代中百分百为垃圾的内存分段被回收了，部分为垃圾的内存分段被计算了出来。
  默认情况下，这些老年代的内存分段会分8次(可以通过-XX：G1MixedGCCountTarget设置)被回收。
- 混合回收的回收集(Collection Set) 包括八分之一的老年代内存分段， Eden区内存分段， Survivor区内存分段。
  混合回收的算法和年轻代回收的算法完全一样， 只是回收集多了老年代的内存分段。
  具体过程请参考上面的年轻代回收过程。
- 由于老年代中的内存分段默认分8次回收，G1会优先回收垃圾多的内存分段。垃圾占内存分段比例越高的，越会被先回收。
  并且有一个阈值会决定内存分段是否被回收，-XX：G1MixedGC Live Threshold Percent， 默认为65%，
  意思是垃圾占内存分段比例要达到65%才会被回收。
  如果垃圾占比太低，意味着存活的对象占比高，在复制的时候会花费更多的时间。
- 混合回收并不一定要进行8次。有一个阈值-XX：G1Heap Waste Percent， 默认值为10%，
  意思是允许整个堆内存中有10%的空间被浪费，意味着如果发现可以回收的垃圾占堆内存的比例低于10%，
  则不再进行混合回收。因为GC会花费很多的时间但是回收到的内存却很少。



G1回收可选的过程四: FULL GC

G 1的初衷就是要避免Full GC的出现。但是如果上述方式不能正常工作， G 1 会停止应用程序的执行(Stop-The-World)， 使用单线程的内存回收算法进行垃圾回收，性能会非常差，应用程序停顿时间会很长。

要避免Full GC的发生， 一旦发生需要进行调整。什么时候会发生Full Gc呢?比如堆内存太小，当G1在复制存活对象的时候没有空的内存分段可用，则会回退到full gc， 这种情况可以通过增大内存解决。

导致G1FullGc的原因可能有两个:

- 1.Evacuation的时候没有足够的to-space来存放晋升的对象;
- 2.并发处理过程完成之前空间耗尽。

G1回收过程: 补充

从Oracle官方透露出来的信息可获知， 回收阶段(Evacuation) 其实本也有想过设计成与用户程序一起并发执行，
但这件事情做起来比较复杂， 考虑到G 1只是回收一部分Region， 停顿时间是用户可控制的，

所以并不迫切去实现，而选择把这个特性放到了G1之后出现的低延迟垃圾收集器(即ZGC)中。
另外， 还考虑到G 1不是仅仅面向低延迟， 停顿用户线程能够最大幅度提高垃圾收集效率，
为了保证吞吐量所以才选择了完全暂停用户线程的实现方案。

G1回收器优化建议

年轻代大小

- 避免使用-X mn或-Xx：NewRatio等相关选项显式设置年轻代大小
- 固定年轻代的大小会覆盖暂停时间目标

暂停时间目标不要太过严苛

- G1 GC的吞吐量目标是90%的应用程序时间和10%的垃圾回收时间
- 评估G1 GC的吞吐量时，暂停时间目标不要太严苛。目标太过严苛表示你愿意承受更多的垃圾回收开销，而这些会直接影响到吞吐量。

垃圾回收器总结

7种垃圾回收器对比

截止JDK1.8，一共有7款不同的垃圾收集器。每一款不同的垃圾收集器都有不同的特点，在具体使用的时候，需要根据具体的情况选用不同的垃圾收集器。

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B/S业务
G1	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

GC 发展阶段:

Serial => Parallel(并行) => CMS(并发) => G1 => ZGC

怎么选择垃圾收集器?

Java垃圾收集器的配置对于JVM优化来说是一个很重要的选择， 选择合适的垃圾收集器可以让JVM的性能有一个很大的提升。

1. 优先调整堆的大小让JVM自适应完成。
2. 如果内存小于100M，使用串行收集器。
3. 如果是单核、单机程序，并且没有停顿时间的要求，串行收集器。
4. 如果是多CPU、需要高吞吐量、允许停顿时间超过1秒， 选择并行或者JVM自己选择。
5. 如果是多CPU、追求低停顿时间， 需快速响应 (比如延迟不能超过1秒， 如互联网应用)，使用并发收集器。

官方推荐G1，性能高。现在互联网的项目，基本都是使用G1。

最后需要明确一个观点

1. 没有最好的收集器，更没有万能的收集；
2. 调优永远是针对特定场景、特定需求，不存在一劳永逸的收集器

对于垃圾收集，面试官可以循序渐进从理论、实践各种角度深入，也未必是要求面试者什么都懂。但如果你懂得原理，一定会成为面试中的加分项。

这里较通用、基础性的部分如下：

- 垃圾收集的算法有哪些?如何判断一个对象是否可以回收?
- 垃圾收集器工作的基本流程。

另外，大家需要多关注垃圾回收器这一章的各种常用的参数。

GC日志分析

通过阅读GC日志， 我们可以了解Java虚拟机内存分配与回收策略。

内存分配与垃圾回收的参数列表

- -XX：+PrintGC 输出GC日志。类似：-verbose：gc
- -XX：+PrintGCDetails 输出GC的详细日志
- -XX：+PrintGCTimeStamps 输出GC的时间戳 (以基准时间的形式)
- -XX：+PrintGCDateStamps 输出GC的时间戳 (以日期的形式，如2013-05-04 T 21：53：59.234 +0800)
- -XX：+PrintHeapAtGC 在进行GC的前后打印出堆的信息
- -xlog gc：../logs/gc.log 日志文件的输出路径

参数日志展示

- 打开GC日志：

```
-verbose:gc
```

- 这个只会显示总的GC堆的变化， 如下：

```
[GC (Allocation Failure) 80832K->19298K(227840K), 0.0084018 secs]
[GC (Metadata GC Threshold) 109499K->21465K(228352K), 0.0184066 secs]
[Full GC (Metadata GC Threshold) 21465K->16716K(201728K), 0.0619261 secs]
```

- 参数解析：

```
GC、Full GC: GC的类型， GC只在新生代上进行， Full GC包括永生代， 新生代， 老年代。
Allocation Failure: GC发生的原因。
80832K->19298K: 堆在GC前的大小和GC后的大小。
228840k: 现在的堆大小。
0.0084018 secs: GC持续的时间。
```

==========

- 打开GC日志：

```
-verbose:gc -XX:+PrintGCDetails
```

- 输入信息如下：

```
[GC (Allocation Failure) [PSYoungGen: 70640K->10116K(141312K)] 80541K->20017K(227328K), 0.0172573
secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
[GC (Metadata GC Threshold) [PSYoungGen: 98859K->8154K(142336K)] 108760K->21261K(228352K),
0.0151573 secs] [Times: user=0.00 sys=0.01, real=0.02 secs]
[Full GC (Metadata GC Threshold) [PSYoungGen: 8154K->0K(142336K)] [ParOldGen: 13107K-
>16809K(62464K)] 21261K->16809K(204800K), [Metaspace: 20599K->20599K(1067008K)], 0.0639732 secs]
[Times: user=0.14 sys=0.00, real=0.06 secs]
```

- 参数解析：

```
GC, Full FC: 同样是GC的类型
Allocation Failure: GC原因
PSYoungGen: 使用了Parallel Scavenge并行垃圾收集器的新生代GC前后大小的变化
ParOldGen: 使用了Parallel Old并行垃圾收集器的老年代GC前后大小的变化
Metaspace: 元数据区GC前后大小的变化， JDK1.8中引入了元数据区以替代永久代
xxx secs: 指GC花费的时间
Times: user: 指的是垃圾收集器花费的所有CPU时间， sys: 花费在等待系统调用或系统事件的时间， real:
GC从开始到结束的时间，包括其他进程占用时间片的实际时间。
```

==========

- 打开GC日志：

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
```

- 输入信息如下：

```
2019-09-24T22:15:24.518+0800: 3.287: [GC (Allocation Failure) [PSYoungGen: 136162K-
>5113K(136192K)] 141425K->17632K(222208K), 0.0248249 secs] [Times: user=0.05
sys=0.00, real=0.03 secs]

2019-09-24T22:15:25.559+0800: 4.329: [GC (Metadata GC Threshold) [PSYoungGen:
97578K->10068K(274944K)] 110096K->22658K(360960K), 0.0094071 secs] [Times: user=0.00
sys=0.00, real=0.01 secs]

2019-09-24T22:15:25.569+0800: 4.338: [Full GC (Metadata GC Threshold) [PSYoungGen:
10068K->0K(274944K)] [ParOldGen: 12590K->13564K(56320K)] 22658K->13564K(331264K),
[Metaspace: 20590K->20590K(1067008K)], 0.0494875 secs] [Times: user=0.17 sys=0.02,
real=0.05 secs]
```

说明：带上了日期和时间

==========

- 如果想把GC日志存到文件的话， 是下面这个参数：

```
-Xloggc:/path/to/gc.log
```

代码示例

```
1 import java.util.ArrayList;
2
```

```java
/**
 * -Xms60m -Xmx60m -XX:SurvivorRatio=8 -XX:+PrintGCDetails -Xloggc:./logs/gc.lo
 *
 * @author shkstart  shkstart@126.com
 * @create 2020  18:12
 */
public class GCLogTest {
    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();

        for (int i = 0; i < 500; i++) {
            byte[] arr = new byte[1024 * 100];//100KB
            list.add(arr);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

日志补充说明

- ”[GC”和”[Full GC”说明了这次垃圾收集的停顿类型， 如果有”Full”则说明GC发生了”Stop The World”
- 使用Serial收集器在新生代的名字是Default New Generation， 因此显示的是”[Def New”
- 使用Par New收集器在新生代的名字会变成”[Par New”， 意思是”ParallelNew Generation”
- 使用ParallelScavenge收集器在新生代的名字是”[PS Young Gen”
- 老年代的收集和新生代道理一样，名字也是收集器决定的
- 使用G 1收集器的话， 会显示为”garbage-first heap”

- Allocation Failure
  - 表明本次引起GC的原因是因为在年轻代中没有足够的空间能够存储新的数据了。
- [PS Young Gen：5986K->696K(8704K)]5986K->704K(9216K)
  - 中括号内：GC回收前年轻代大小，回收后大小，(年轻代总大小)
  - 括号外：GC回收前年轻代和老年代大小，回收后大小，(年轻代和老年代总大小)
- user代表用户态回收耗时， sys内核态回收耗时， rea实际耗时。由于多核的原因， 时间总和可能会超过real 时间

Heap(堆)

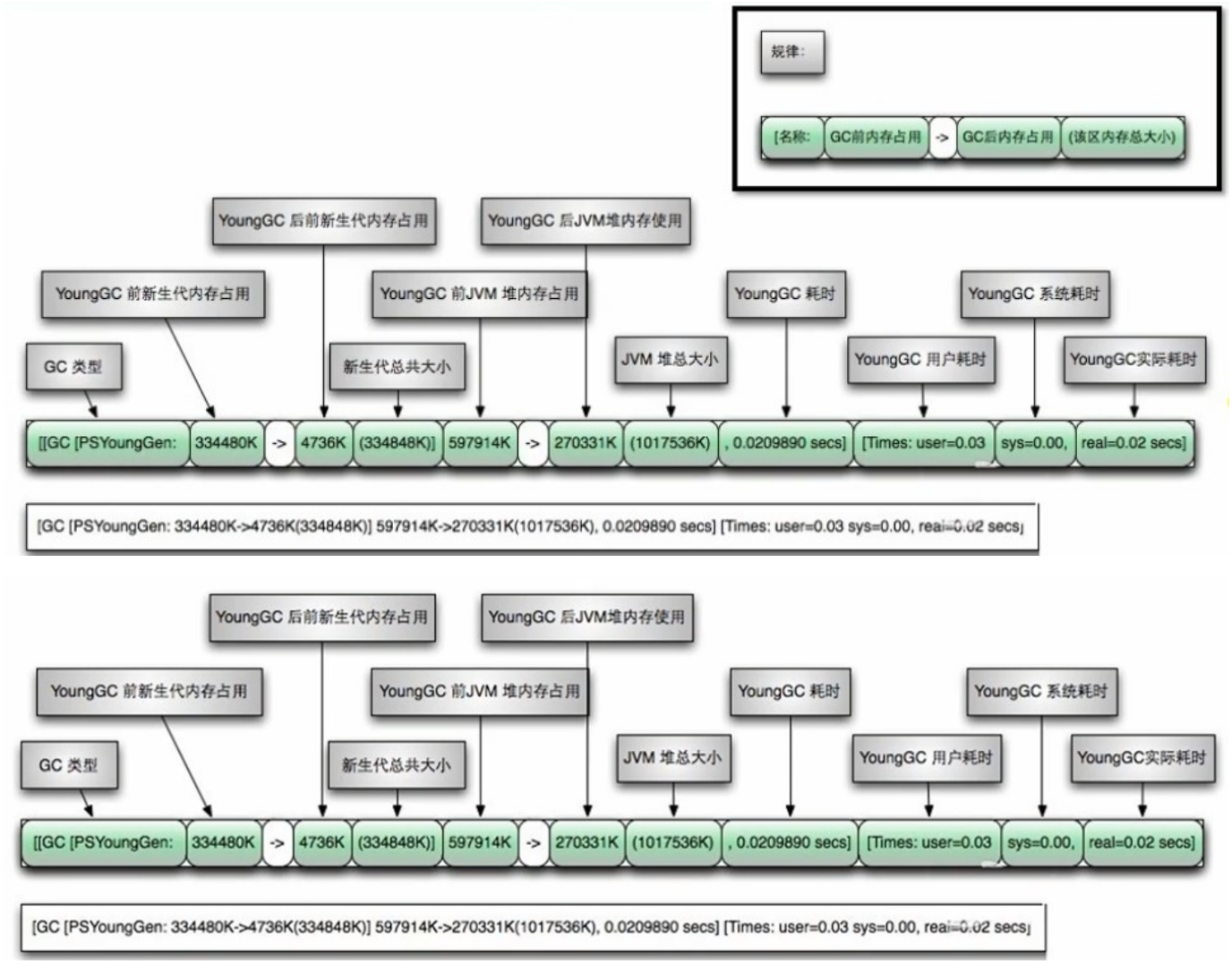
- **PSYoungGen**(Parallel Scavenge收集器新生代) total 9216K，used 6234K
  - [0x00000000ff600000，0x0000000100000000，0x0000000100000000)
- **eden space**(堆中的Eden区默认占比是8) 8192K， 76% used
  - [0x00000000ff 600000， 0x00000000ffc16b 08， 0x00000000ffe 00000)
- **from space**(堆中的Survivor， 这里是From Survivor区默认占比是1) 1024K，0% used
  - [0x0000000fff00000，0x00000000fff00000，0x0000000100000000)
- **to space**(堆中的Survivor， 这里是to Survivor区默认占比是1， 需要先了解一下堆的分配策略)1024K，0%used
  - [0x00000000ffe 00000， 0x00000000ffe 00000， 0x00000000fff 00000)

- **ParOldGen**(老年代总大小和使用大小) total10240K，used 7001K
  - [0x00000000fec 00000，0x00000000ff600000，0x00000000ff600000)
- **object space**(显示个使用百分比)10240K，68%used
  - [0x00000000fec 00000， 0x00000000ff2d 6630， 0x00000000ff 600000)

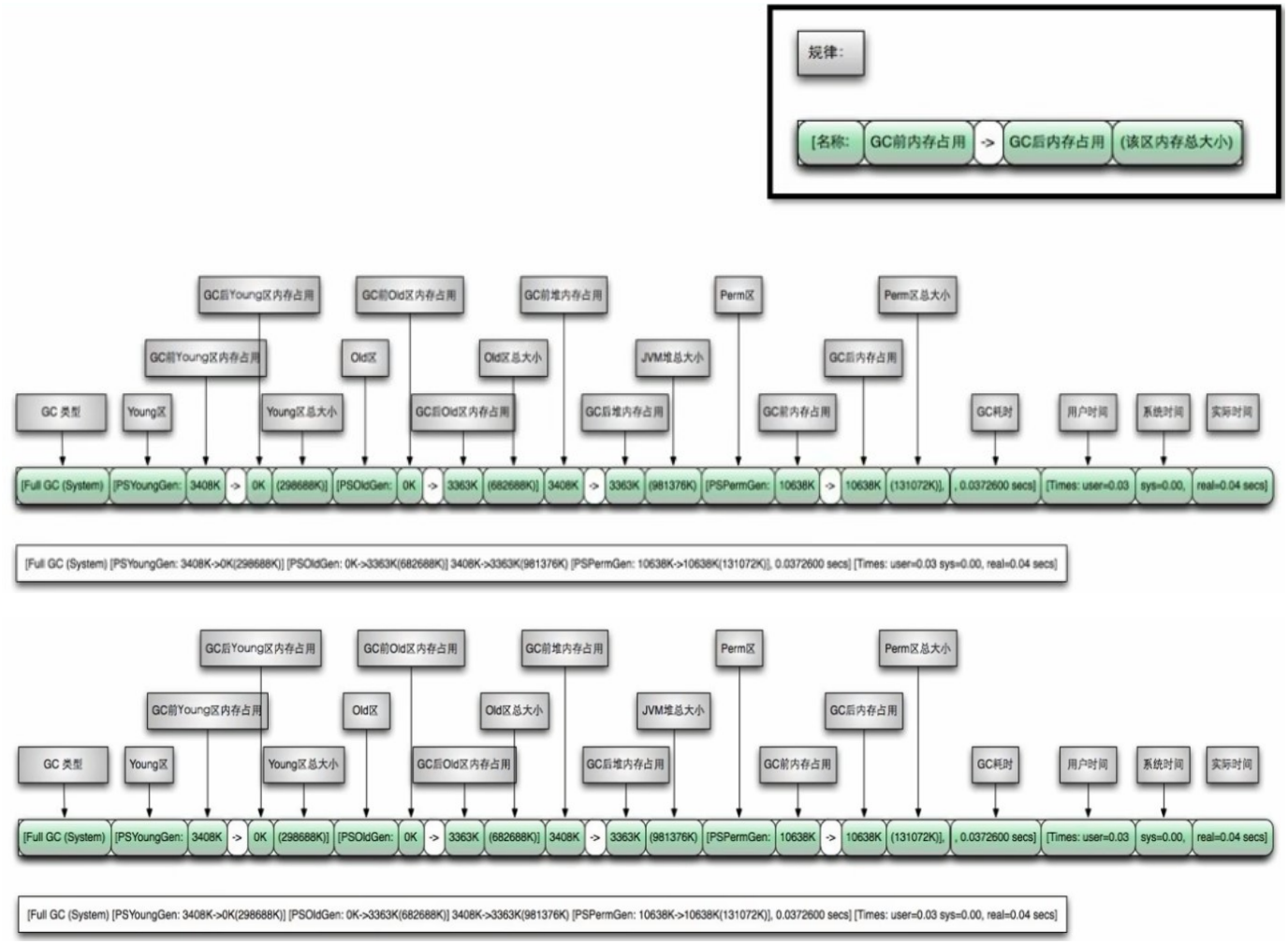
- **PSPermGen**(永久代总大小和使用大小) total21504K，used 4949K
  - [0x00000000f9a00000，0x00000000faf 00000， 0x00000000fec 00000)
- **object space**(显示个使用百分比，自己能算出来)21504K， 23%used
  - [0x00000000f9a 00000， 0x00000000f9ed55e 0， 0x00000000faf 00000)

GC 日志详解

Minor GC日志



FULL GC日志



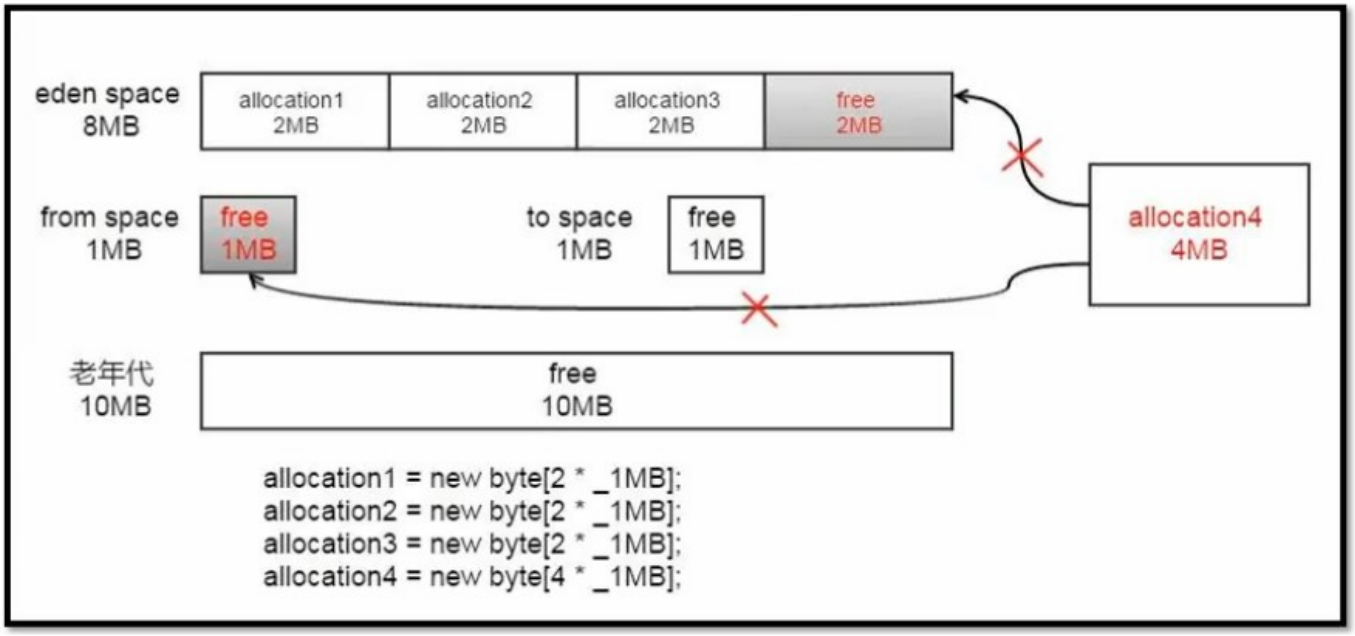
日志中堆空间数据解读

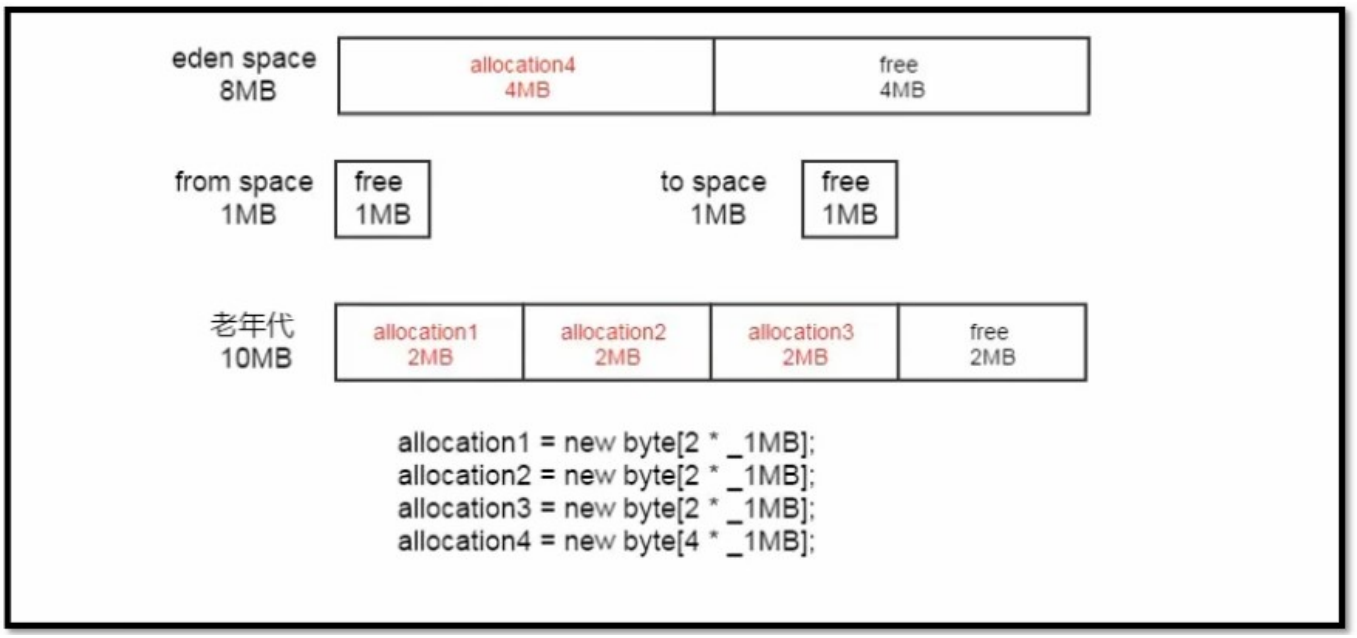
代码示例

```
/**
 * 在jdk7 和 jdk8中分别执行
 * -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
 * @author shkstart  shkstart@126.com
 * @create 2020  0:12
 */
public class GCLogTest1 {
    private static final int _1MB = 1024 * 1024;

    public static void testAllocation() {
        byte[] allocation1, allocation2, allocation3, allocation4;
        allocation1 = new byte[2 * _1MB];
```

```
13          allocation2 = new byte[2 * _1MB];
14          allocation3 = new byte[2 * _1MB];
15          allocation4 = new byte[4 * _1MB];
16      }
17
18      public static void main(String[] agrs) {
19          testAllocation();
20      }
21  }
```

JDK 7 里面执行





2m 2m 2m 这时候来一个 4m 放不下, 触发GC 但是 from to 区也放不下, 于是直接进入老年代, 2m 2m 2m　4M进入eden区

JDK8 执行

大对象，直接进入老年代，并且根据参数，发生了GC

日志分析工具

可以用一些工具去分析这些gc日志。

常用的日志分析工具有：GCViewer、GCEasy、GCHisto、GCLogViewer、Hp jmeter、garbagecat等。

-Xloggc:./logs/gc.log

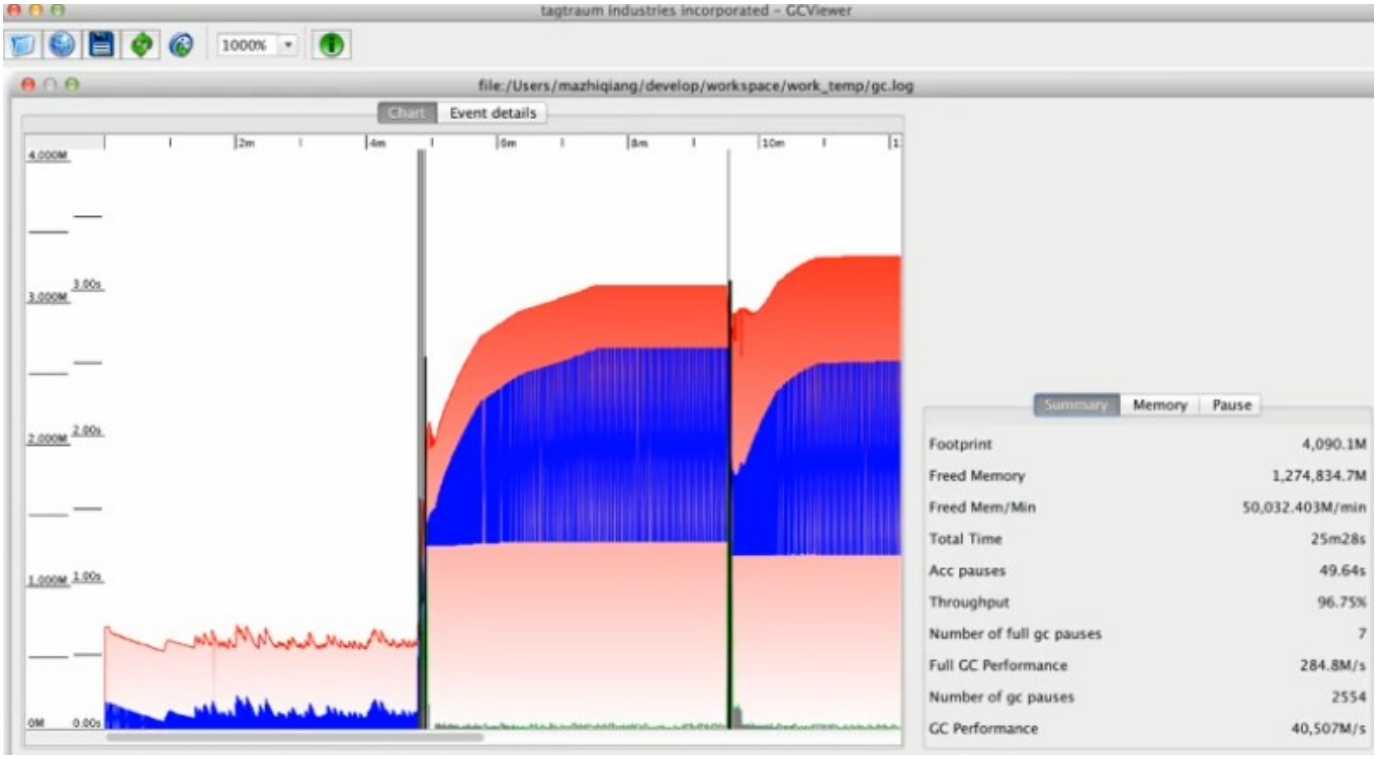
```
import java.util.ArrayList;

/**
 * -Xms60m -Xmx60m -XX:SurvivorRatio=8 -XX:+PrintGCDetails -Xloggc:./logs/gc.lo
 *
 * @author shkstart  shkstart@126.com
 * @create 2020  18:12
```

```
 */
public class GCLogTest {
    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();

        for (int i = 0; i < 500; i++) {
            byte[] arr = new byte[1024 * 100];//100KB
            list.add(arr);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

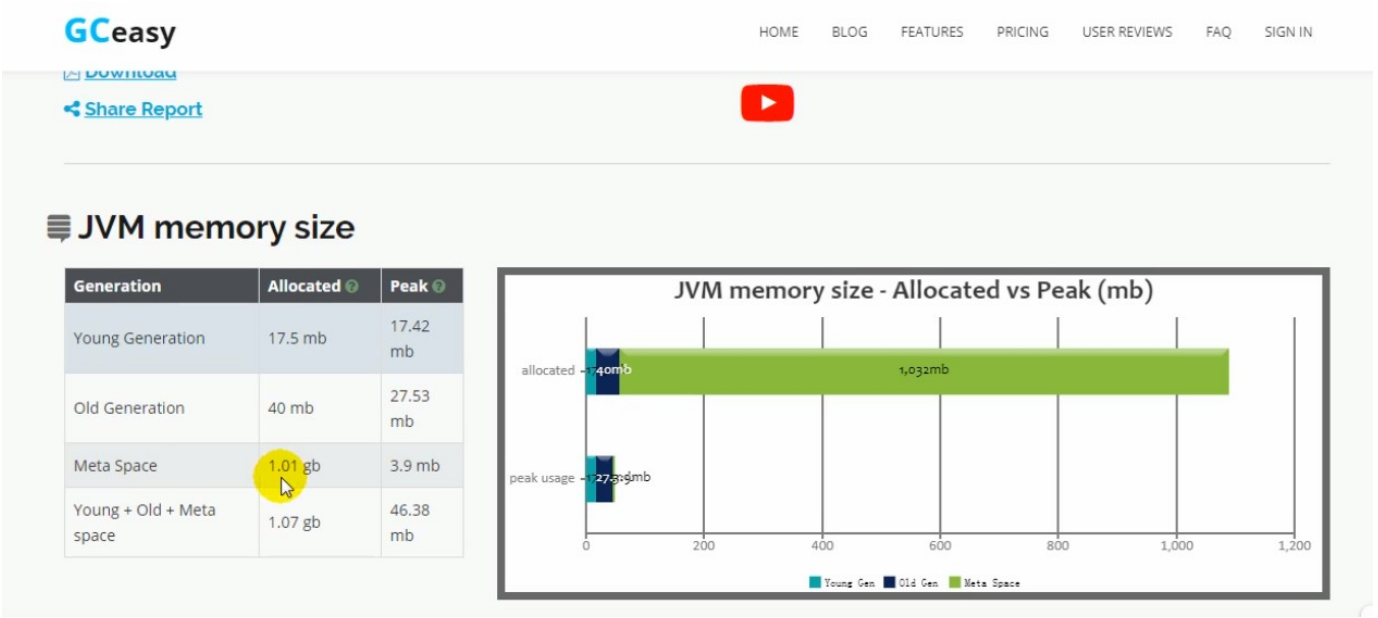
gc.log

```
Java HotSpot(TM) 64-Bit Server VM (25.131-b11) for windows-amd64 JRE (1.8.0_131
Memory: 4k page, physical 16688008k(10560772k free), swap 33374156k(27498644k f
CommandLine flags: -XX:InitialHeapSize=62914560 -XX:MaxHeapSize=62914560 -XX:+P
6.110: [GC (Allocation Failure) [PSYoungGen: 15287K->2552K(17920K)] 15287K->130
13.726: [GC (Allocation Failure) [PSYoungGen: 17842K->2536K(17920K)] 28364K->28
13.736: [Full GC (Ergonomics) [PSYoungGen: 2536K->0K(17920K)] [ParOldGen: 25731
21.214: [Full GC (Ergonomics) [PSYoungGen: 15303K->2500K(17920K)] [ParOldGen: 2
Heap
 PSYoungGen      total 17920K, used 10648K [0x00000000fec00000, 0x0000000100000
  eden space 15360K, 69% used [0x00000000fec00000,0x00000000ff6663f0,0x00000000
  from space 2560K, 0% used [0x00000000ffd80000,0x00000000ffd80000,0x0000000100
  to   space 2560K, 0% used [0x00000000ffb00000,0x00000000ffb00000,0x00000000ff
 ParOldGen       total 40960K, used 40597K [0x00000000fc400000, 0x00000000fec00
  object space 40960K, 99% used [0x00000000fc400000,0x00000000feba57a0,0x000000
 Metaspace       used 3999K, capacity 4568K, committed 4864K, reserved 1056768K
  class space    used 447K, capacity 460K, committed 512K, reserved 1048576K

```

GCViewer



GCeasy.io



垃圾回收器的新发展

- GC仍然处于飞速发展之中，目前的默认选项G1GC在不断的进行改进，
  很多我们原来认为的缺点， 例如串行的Full GC、Card Table扫描的低效等， 都已经被

大幅改进，
例如， JDK 10以后，Full Gc已经是并行运行， 在很多场景下， 其表现还略优于Parallel GC的并行Full GC实现。

- 即使是Serial GC， 虽然比较古老， 但是简单的设计和实现未必就是过时的，
它本身的开销，不管是GC相关数据结构的开销，还是线程的开销，都是非常小的，
所以随着云计算的兴起， **在Serverless等新的应用场景下， Serial GC找到了新的舞台**。
- 比较不幸的是CMS GC， 因为其算法的理论缺陷等原因，
虽然现在还有非常大的用户群体， 但在JDK 9中已经被标记为废弃， 并在JDK 14版本中移除。

JDK11 新特性

新GC

- **JEP318：**
Epsilon: A No-Op Garbage Collector（Epsilon 垃圾回收器，"No-Op（无操作）"回收器）
http://openjdk.java.net/jeps/318

- **JEP333：**
ZGC: A Scalable Low-Latency Garbage Collector (Experimental)（ZGC：可伸缩的低延迟垃圾回收器，处于实验性阶段）

181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
335: Deprecate the Nashorn JavaScript Engine
336: Deprecate the Pack200 Tools and API

JDK12

现在G1回收器已成为默认回收器好几年了。

我们还看到了引入了两个新的收集器：**ZGC(JDK 11出现)** 和**Shenandoah(OpenJDK 12)**。

主打特点：低停顿时间

Shenandoah GC

Open JDK 12的Shenandoah GC：低停顿时间的GC(实验性)

- Shenandoah， 无疑是众多GC中最孤独的一个。
  是第一款不由Oracle公司团队领导开发的HotSpot垃圾收集器。不可避免的受到官方的排挤。
  比如号称Open JDK和Oracle JDK没有区别的Oracle公司仍拒绝在Oracle JDK 12中支持Shenandoah。
- Shenandoah垃圾回收器最初由RedHat进行的一项垃圾收集器研究项目Pause less GC的实现，
  旨在针对JVM上的内存回收实现低停顿的需求。在2014年贡献给Open JDK。
- RedHat研发Shenandoah团队对外宣称， Shenandoah垃圾回收器的暂停时间与堆大小无关，
  这意味着无论将堆设置为200MB还是200GB，99.9%的目标都可以把垃圾收集的停顿时间限制在十毫秒以内。
  不过实际使用性能将取决于实际工作堆的大小和工作负载。

Shenandoah开发团队在实际应用中的测试数据

收 集 器	运行时间	总 停 顿	最大停顿	平均停顿
Shenandoah	387.602s	320ms	89.79ms	53.01ms
G1	312.052s	11.7s	1.24s	450.12ms
CMS	285.264s	12.78s	4.39s	852.26ms
Parallel Scavenge	260.092s	6.59s	3.04s	823.75ms

这是RedHat在2016年发表的论文数据， 测试内容是使用ES对200GB的维基百科数据进行索引。从结果看：

- 停顿时间比其他几款收集器确实有了质的飞跃，但也未实现最大停顿时间控制在十毫秒以内的目标。
- 而吞吐量方面出现了明显的下降，总运行时间是所有测试收集器里最长的。

总结:

Shenandoah GC的弱项：高运行负担下的吞吐量下降。

ShenandoahGC的强项：低延迟时间。

Shenandoah GC的工作过程大致分为九个阶段， 这里就不再赘述。在之前Java 12新特性视频里有过介绍。

【Java l 2新特性地址】

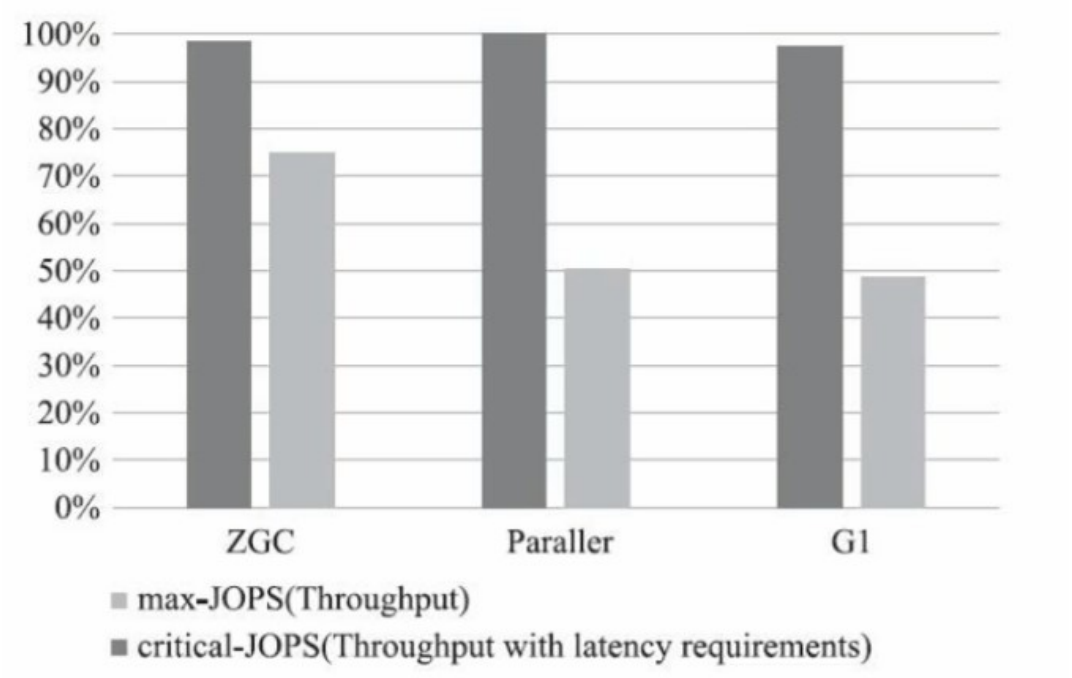
https://www.bilibili.com/video/BV1jJ411M7kQ/

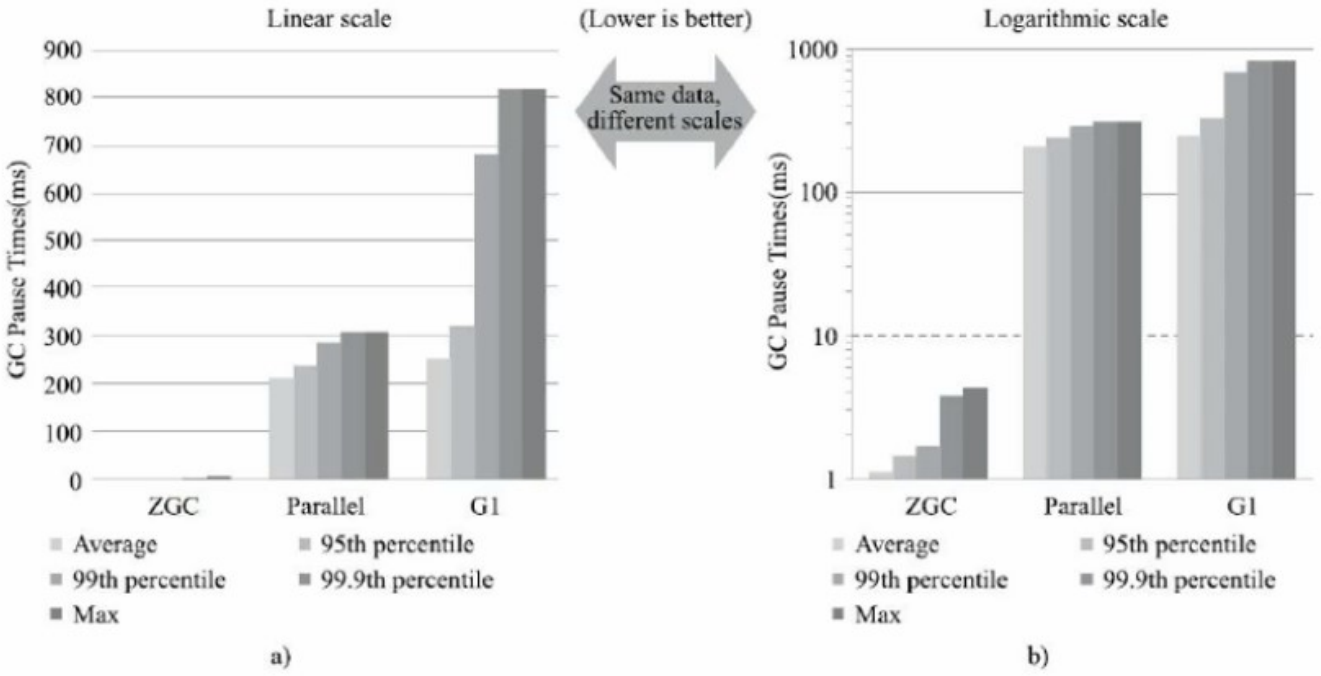
明日之星：ZGC

https://docs.oracle.com/en/java/javase/12/gctuning/

- ZGC与Shenandoah目标高度相似， 在尽可能对吞吐量影响不大的前提下，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。
- 《深入理解Java虚拟机》一书中这样定义ZGC：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-压缩算法的，以低延迟为首要目标的一款垃圾收集器。
- ZGC的工作过程可以分为4个阶段：并发标记 -> 并发预备重分配 -> 并发重分配 -> 并发重映射等。
- ZGC几乎在所有地方并发执行的， 除了初始标记的是STW的。
  所以停顿时间几乎就耗费在初始标记上，这部分的实际时间是非常少的。

测试数据





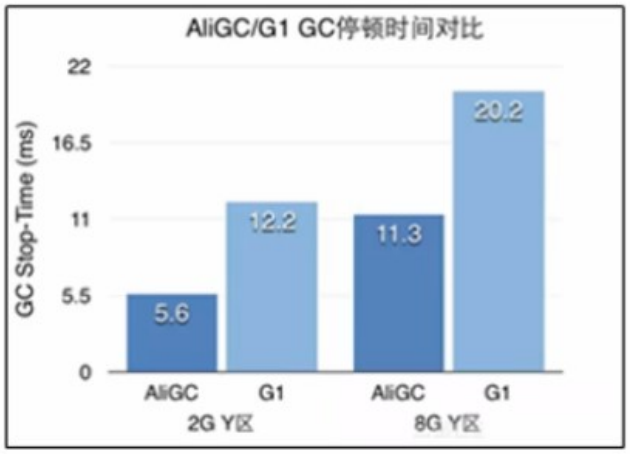
- **在ZGC的强项停顿时间测试上， 它毫不留情的将Parallel、G 1拉开了两个数量级的差距。**
- **无论平均停顿、95%停顿、99%停顿、99.9%停顿，还是最大停顿时间， ZGC都能毫不费劲控制在10毫秒以内。**
- 虽然ZGC还在试验状态， 没有完成所有特性，但此时性能已经相当亮眼，用“令人震惊、革命性”来形容，不为过。
- **未来将在服务端、大内存、低延迟应用的首选垃圾收集器。**

JDK14 新特性

- JEP 364：ZGC应用在macos上
- JEP 365：ZGC应用在Windows上
- JDK 14之前， zGc仅Linux才支持。
- 尽管许多使用zGC的用户都使用类Linux的环境， 但在Windows和macos上， 人们也需要ZGC进行开发部署和测试。许多桌面应用也可以从ZGC中受益。因此， ZGC特性被移植到了Windows和macos上。
- 现在mac或Windows上也能使用zGc了， 示例如下：-XX：+UnlockExperimentalVMOptions **-XX：+UseZGC**

Other 垃圾回收器

- **Ali GC**是阿里巴巴JVM团队基于G 1算法，指定场景下的对比：**面向大堆(Large Heap)应用场景**。



- 当然，其他厂商也提供了各种独具一格的GC实现，例如比较有名的低延迟GC，Zing(https：//www.infoq.com/articles/azul_gc_in_detail)， 有兴趣可以参考提供的链接。