链滴

JVM_10 垃圾回收 3- 垃圾回收器

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1604060879965

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20200727.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="GC的分类与性能指标">GC 的分类与性能指标</h2>

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。

按线程数分，可以分为串行垃圾回收器和并行垃圾回收器
<img src="https://ld246.com/images/img-loading.svg" alt="172f886ddbb6f49a.png" dat
-src="https://b3logfile.com/file/2020/10/172f886ddbb6f49a-23c15e45.png?imageView2/2/in
erlace/1/format/jpg">

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
，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
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
。
 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。

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，
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。

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
收过程完全结束。

<img src="https://ld246.com/images/img-loading.svg" alt="172f8873dadc3689.png" dat
-src="https://b3logfile.com/file/2020/10/172f8873dadc3689-a0256e4e.png?imageView2/2/in
erlace/1/format/jpg">
按碎片处理方式分，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。

再分配对象空间使用: 指针碰撞

非压缩式的垃圾回收器不进行这步操作。

再分配对象空间使用: 空闲列表

按工作的内存区间分，又可分为年轻代垃圾回收器和老年代垃圾回收器</p

<h3 id="评估GC的性能指标">评估 GC 的性能指标</h3>

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。
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，对延迟反带来负面效果。
简单来说，主要抓住两点：

吞吐量
暂停时间

<h4 id="吞吐量">吞吐量</h4>

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，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

<h4 id="暂停时间">暂停时间</h4>

“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。.
/li>

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

<img src="https://ld246.com/images/img-loading.svg" alt="172f88838b4a0d54.png" dat
-src="https://b3logfile.com/file/2020/10/172f88838b4a0d54-d73b078d.png?imageView2/2/in
erlace/1/format/jpg">

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，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。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。

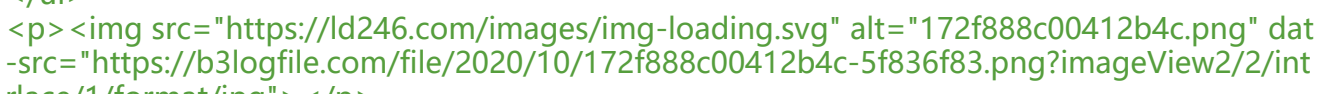
在设计（或使用） GC 算法时，我们必须确定我们的目标： 一个 GC 算法只可能针对两个目标之（即只专注于较大吞吐量或最小暂停时间），或尝试找到一个二者的折衷。
现在标准：在最大吞吐量优先的情况下，降低停顿时间。

<h2 id="不同的垃圾回收器概述">不同的垃圾回收器概述</h2>
□□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？
<h3 id="垃圾收集器发展史">垃圾收集器发展史</h3>
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arbage Collector.

1999 年随 JDK1.3.1 一起来的是串行方式的 Serial GC，它是第一款 GC。ParNew 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 JDK1.4.2 一起发布
Parallel GC 在 JDK6 之后成为 HotSpot 默认 GC。
2012 年，在 JDK1.7u4 版本中，G1 可用。
2017 年，JDK9 中 G1 变成默认的垃圾收集器，以替代 CMS。
2018 年 3 月，JDK10 中 G1 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迟。
------------分水岭------------
2018 年 9 月，JDK11 发布。引入 Epsilon 垃圾回收器，又被称为"No 一 Op （无操作） "回收。同时，引入 ZGC：可伸缩的低延迟垃圾回收器（Experimental）。
2019 年 3 月，JDK12 发布。 增强 G1，自动返回未用堆内存给操作系统。同时，引入 Shenandah GC：低停顿时间的 GC（Experimental）。
2019 年 9 月，JDK13 发布。增强 ZGC，自动返回未用堆内存给操作系统。
2020 年 3 月，JDK14 发布。删除 CMS 垃圾回收器。扩展 ZGC 在 macOS 和 Windows 上的用。

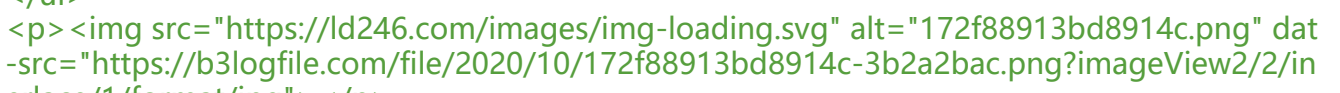
<h3 id="7款经典的垃圾收集器">7 款经典的垃圾收集器</h3>

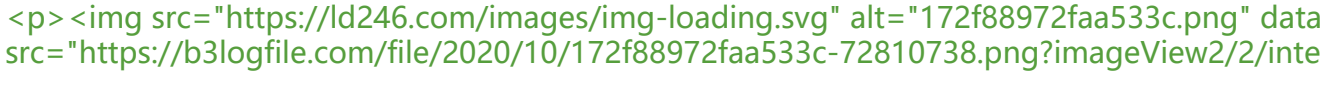
串行回收器：Serial ， Serial Old
并行回收器：ParNew ， Parallel Scavenge ， Parallel Old
并发回收器：CMS ， G1



<h3 id="7款经典的垃圾收集器与垃圾分代之间的关系">7 款经典的垃圾收集器与垃圾分代之间的关</h3>

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；



<h3 id="垃圾收集器的组合关系">垃圾收集器的组合关系</h3>


lace/1/format/jpg">

两个收集器间有连线，表明它们可以搭配使用：
Serial/Serial Old、Serial/CMS、 ParNew/Serial Old、ParNew/CMS、
Parallel Scavenge/Serial Old、Parallel Scavenge/Parallel Old、G1；
其中 Serial Old 作为 CMS 出现"Concurrent Mode Failure"失败的后备预案。
（红色虚线）由于维护和兼容性测试的成本，在 JDK 8 时将 Serial+CMS、ParNew+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（JEP214），即：除。
（绿色虚线）JDK 14 中：弃用 Parallel Scavenge 和 Serial0ld GC 组合（JEP366 ）
（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合适的收集器。

查看默认的垃圾收集器

— 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

<pre><code class="highlight-chroma">/**
 * -XX:+PrintCo mandLineFlags
 *
 * -XX:+UseSerial C:表明新生代使用Serial GC ，同时老年代使用Serial Old GC
 *
 * -XX:+UseParN wGC：标明新生代使用ParNew GC
 *
 * -XX:+UseParalle GC:表明新生代使用Parallel GC
 * -XX:+UseParalle OldGC : 表明老年代使用 Parallel Old GC
 * 说明：二者可相互激活
 *
 * -XX:+UseConc arkSweepGC：表明老年代使用CMS GC。同时，年轻代会触发对ParNew 的使用
 */
public class GCUs Test {
    public static vo d main(String[] args) {
        ArrayList< yte[]> list = new ArrayList<>();

        while(true){
            byte[] arr =

```
new byte[100];
            list.add(arr)

            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

输出

```
-XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
```

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。JDK1.3 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
  - Serial Old 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 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，直到它收集结束（Stop The World ）。

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

器。

等价于新生代用 Serial GC，且老年代用 Serial Old GC
控制台输出 <code>-XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296 -XX:+
rintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+
seSerialGC</code>

<h3 id="总结">总结</h3>

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
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 Javaweb 应用程序中是不会
用串行垃圾收集器的。

<h2 id="ParNew回收器-并行回收">ParNew 回收器:并行回收</h2>

如果说 Serial GC 是年轻代中的单线程垃圾收集器，那么 ParNew 收集器则是 Serial 收集器的
线程版本。

Par 是 Parallel 的缩写，New： 只能处理的是新生代

ParNew 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
。ParNew 收集器在年轻代中同样也是采用复制算法、"Stop — the — World"机制。
ParNew 是很多 JVM 运行在 Server 模式下新生代的默认垃圾收集器。

-src="https://b3logfile.com/file/2020/10/172f88b5da16f393-2c23829d.png?imageView2/2/in
erlace/1/format/jpg">

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
换线程的资源）
由于 ParNew 收集器是基于并行回收，那么是否可以断定 ParNew 收集器的回收效率在任何场
下都会比 Serial 收集器更高效?

ParNew 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，ParNew 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
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 ParNew GC 能与 CMS 收集器配合工作
在程序中，开发人员可以通过选项"— XX： +UseParNewGC"手动指定使用.ParNew 收集器执
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。

<h2 id="Parallel回收器-吞吐量优先">Parallel 回收器:吞吐量优先</h2>

HotSpot 的年轻代中除了拥有 ParNew 收集器是基于并行回收的以外，Parallel Scavenge 收集
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 ParNew 收集器不同，Parallel Scavenge 收集器的目标则是达到一个可控制的吞量（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 ParNew 一个重要区别。

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、科学计算的应用程序。
Parallel 收集器在 JDK1.6 时提供了用于执行老年代垃圾收集的 Parallel Old 收集器，用来代替年代的 Serial Old 收集器。
Parallel Old 收集器采用了标记一压缩算法，但同样也是基于并行回收和”Stop — The — Worl“机制。

<p>

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 模下的内存回收性能很不错。
在 Java8 中，默认是此垃圾收集器

<h3 id="参数配置">参数配置</h3>

— 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： +UseParallel0ldGC 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 jdk8 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， ParallelGCThreads 的值等于 CPU 数量。
当 CPU 数量大于 8 个， ParallelGCThreads 的值等于 3+[5*CPU_ Count]/8]

— XX ： MaxGCPau3eMillis 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 MaxGCPauseMills 以内，收集器在工作时会调整 Java 堆大或者其他一些参数。
 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。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。

 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 参数就容易超设定的比例。

— 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、晋升老年代的对象年龄等参数会被自调整，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、目标的吐量（GCTimeRatio）和停顿时间（MaxGCPauseMills），让虚拟机自己完成调优工作。

<h2 id="CMS回收器-低延迟">CMS 回收器:低延迟</h2>

在 JDK1.5 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: MS （Concurrent — Mark — Sweep）收集器，这款收集器是 HotSpot 虚拟机中第一款真正意义的并发收集器，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）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需求。

CMS 的垃圾收集算法采用标记一清除算法，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 cavenge 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 ParNew 者 Serial 收集器中的一个。
在 G1 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。</li

<img src="https://ld246.com/images/img-loading.svg" alt="172f88c4cd91d748.png" dat-src="https://b3logfile.com/file/2020/10/172f88c4cd91d748-4762c9ce.png?imageView2/2/interlace/1/format/jpg">
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、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 GCRoots 能直接关联的对象。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

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一清除算法，这意味着每次执行完内存回收后，由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 the Pointer）术，而只能够选择空闲列表（Free List） 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？
<code>答案其实很简单，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。Mark Compact 适合“Stop the World”这种场景”下使用</code>
<h3 id="CMS的优点-">CMS 的优点：</h3>

并发收集
低延迟

<h3 id="CMS的弊端-">CMS 的弊端：</h3>

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 ull 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。

<h3 id="参数设置">参数设置</h3>

— 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（Old 区用） +Serial Old 的组合。

— XX：CMSInitiatingOccupanyFraction 设置堆内存使用率的阈值，一旦达到该阈值，便开始行回收。

JDK5 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% 时，会执行一次 CMS 回。 JDK6 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 Full GC 的执次数。

— XX： +UseCMSCompactAtFullCollection 用于指定在执行完 Full GC 后对内存空间进行压

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间变得更长了。
— XX：CMSFullGCsBeforeCompaction 设置在执行多少次 Full GC 后对内存空间进行压缩整。
— XX：ParallelCMSThreads 设置 CMS 的线程数量。

CMS 默认启动的线程数是（ParallelGCThreads+3） /4，
ParallelGCThreads 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 收集器线的影响，应用程序的性能在垃圾回收阶段可能会非常糟糕。

<h3 id="小结-">小结：</h3>
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，请选 CMS GC。
<h3 id="JDK-后续版本中CMS的变化">JDK 后续版本中 CMS 的变化</h3>

JDK9 新特性： CMS 被标记为 Deprecate 了（JEP291）

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。

JDK14 新特性： 删除 CMS 垃圾回收器（JEP363）

移除了 CMS 垃圾收集器，如果在 JDK14 中使用— XX： +UseConcMarkSweepGC 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

<h2 id="G1回收器-区域化分代式">G1 回收器:区域化分代式</h2>
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 （G1）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 G1 （Garbage — First） 垃圾回收器是在 Java7 update4 之后引入的一个新的垃圾回收器，是当今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（pause time） ，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 G1 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）。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。G1 跟踪各个 Region 里面的垃堆积的价值大小（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，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 G1 一个名字：垃圾

先（Garbage First）。
G1 （Garbage 一 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 JDK1. 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”。</l>
与此同时，CMS 已经在 JDK 9 中被标记为废弃（deprecated）。在 jdk8 中还不是默认的垃圾收器，需要使用一 XX： +UseG1GC 来启用。

<h3 id="优势-">优势</h3>
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 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。

<img src="https://ld246.com/images/img-loading.svg" alt="172f88dad2adfacc.png" dat-src="https://b3logfile.com/file/2020/10/172f88dad2adfacc-8412f1d9.png?imageView2/2/interlace/1/format/jpg">

空间整合

CMS： “标记一清除”算法、内存碎片、若干次 GC 后进行一次碎片整理
G1 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 之间是复算法，但整体上实际可看作是标记一压缩（Mark 一 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 G。尤其是当 Java 堆非常大的时候，G1 的优势更加明显。

可预测的停顿时间模型（即：软实时 soft real 一 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，这样缩小了回收的范围，因此对于全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 CMSGC，G1 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。</l>

>

<h3 id="缺点">缺点</h3>

相较于 CMS，G1 还不具备全方位、压倒性优势。比如在用户程序运行过程中，G1 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。
从经验上来说，在小内存应用上 CMS 的表现大概率会优于 G1，而 G1 在大内存应用，上则发其优势。平衡点在 6 一 8GB 之间。

<h3 id="参数设置-">参数设置</h3>

一 XX：+UseG1GC 手动指定使用 G1 收集器执行内存回收任务。
一 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。</l>
一 XX：MaxGCPauseMillis 设置期望达到的最大 Gc 停顿时间指标（JVM 会尽力实现，但不保达到）。默认值是 200ms
一 XX：ParallelGCThread 设置 STW 工作线程数的值。最多设置为 8
一 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（ParallelGChreads）的 1/4 左右。
一 XX：InitiatingHeapOccupancyPercent 设置触发并发 GC 周期的 Java 堆占用率阈值。超过值，就触发 GC。默认值是 45。

<h3 id="G1回收器的常见操作步骤">G1 回收器的常见操作步骤</h3>
G1 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

G1 中提供了三种垃圾回收模式： YoungGC、 Mixed GC 和 Full GC， 在不同的条件下被触发
<h3 id="适用场景">适用场景</h3>

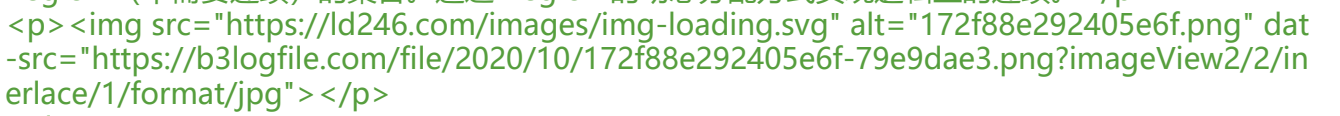
面向服务端应用，针对具有大内存、多处理器的机器。（在普通大小的堆里表现并不.惊喜）

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 JDK1.5 中的 CMS 收集器；
在下面的情况时，使用 G1 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；
③GC 停顿时间过长（长于 0. 5 至 1 秒）。

HotSpot 垃圾收集器里，除了 G1 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 G1 GC 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。

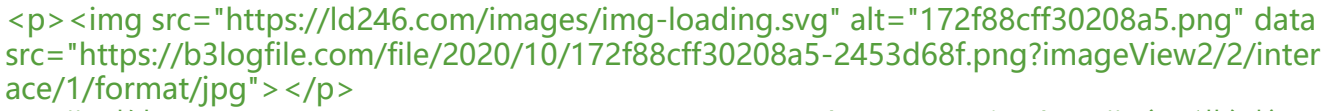
<h3 id="分区region-化整为零">分区 region,化整为零</h3>
使用 G1 收集器时，它将整个 Java 堆划分成约 2048 个大小相同的独立 Region 块，每个 Region 块大小根据堆空间的实际大小而定，整体被控制在 1MB 到 32MB 之间，且为 2 的 N 次幂，即 1MB， 2MB， 4MB， 8MB， 1 6MB， 32MB。可以通过一 XX：G1HeapRegionSize 设定。所有的 Region 大小相同，且在 JVM 生命周期内不会被改变。
虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region （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，如图中的 H 块。主用于存储大对象，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，G1 划分了一个 Humongous 区，它用来专门存放大对。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，时候不得不启动 Full GC。G1 的大多数行为都把 H 区作为老年代的一部分来看待。

<h3 id="G1回收器垃圾回收过程">G1 回收器垃圾回收过程</h3>
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，进行垃回收。

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； G1 的年轻代收集阶段是个并行的独占式收集器。在年轻代回收期，G1 GC 暂停所有应用程序线程，启动多线程执行年轻代回。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。</l
>
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，G1 GC 从老年区间移动存活对象到空闲间，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

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。

<h3 id="记忆集与写屏障">记忆集与写屏障</h3>

一个对象被不同区域引用的问题(分代引用问题)
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 MinorGC 的效率。
解决方法：

 无论 G1 还是其他分代收集器，JVM 都是使用 RememberedSet 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 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 （其他收器：检查老年代对象是否引用了新生代对象） ；
 如果不同，通过 CardTable 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描，也不会有遗漏。

<img src="https://ld246.com/images/img-loading.svg" alt="172f88f075afe7cf.png" data-rc="https://b3logfile.com/file/2020/10/172f88f075afe7cf-738684a7.png?imageView2/2/interlce/1/format/jpg">
<h3 id="G1回收过程详解">G1 回收过程详解</h3>
<h4 id="1--年轻代GC">1. 年轻代 GC</h4>

JVM 启动时，G1 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 空耗尽时，G1 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。
YGC 时，首先 G1 停止应用程序的执行（Stop — The — World），G1 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。

<img src="https://ld246.com/images/img-loading.svg" alt="172f88f5d8bec04e.png" dat-src="https://b3logfile.com/file/2020/10/172f88f5d8bec04e-2cf28ea6.png?imageView2/2/interlace/1/format/jpg">
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 RSet 记的外部引用作为扫描存活对象的入口。

第二阶段，更新 RSet。
处理 dirty card queue（见备注）中的 card，更新 RSet。 此阶段完成后，RSet 可以准确的反老年代对所在的内存分段中对象的引用。

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 会对 Dirty Card Queue 中所有的 card 进行处理，以更新 RSet，保证 RSet 实时准确的反映引用关。那为什么不在引用赋值语句处直接更新 RSet 呢？这是为了性能的需要，RSet 的处理需要线程同步开销会很大，使用队列性能会好很多。

第三阶段，处理 RSet。
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，urvivor 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 01d 区中的内存分段。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 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，少碎片。

<h4 id="2--并发标记过程">2. 并发标记过程</h4>

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 G。
根区域扫描（Root Region Scanning）： G1 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。
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）。
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。G1 采用了比 CMS 更快的初始快照算法：snapshot — at — the — beginning （SATB）。
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，识别可混合回收的区域。为下阶段做铺垫。是 STW 的。

 这个阶段并不会实际上去做垃圾的收集

并发清理阶段：识别并清理完全空闲的区域。

<h4 id="3--混合回收">3. 混合回收</h4>

当越来越多的对象晋升到老年代 oldregion 时，为了避免堆内存被耗尽，虚拟机会触发一个混合垃圾收集器，即 Mixed GC， 该算法并不是一个 OldGC，除了回收整个 Young Region，还会回收部分的 OldRegion。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 OldRegio 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。</>

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（可以通过一 XX： G1MixedGCCountTarget 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，Survor 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，一 XX:G1MixedGCLiveThreholdPercent，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，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。有一个阈值一 XX： G1HeapWastePercent，默认值为 10%，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 1%，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。

<h4 id="4--Full-GC">4. Full GC</h4>
G1 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，G1 会停止应用程序执行（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，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内存解决。
导致 G1Full GC 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。

<h4 id="补充">补充</h4>
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序一起并发执行，但这件事情做起来比较复杂，考虑到 G1 只是回收一部分 Region， 停顿时间是用可控制的，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 ZG）中。另外，还考虑到 G1 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，为保证吞吐量所以才选择了完全暂停用户线程的实现方案。
<h3 id="优化建议">优化建议</h3>

年轻代大小

 避免使用一 Xmn 或一 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。

<h2 id="垃圾回收器总结">垃圾回收器总结</h2>
截止 JDK 1.8，一共有 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，在具使用的时候，需要根据具体的情况选用不同的垃圾收集器。

rlace/1/format/jpg">
GC 发展阶段： Serial => Parallel （并行） => CMS （并发） => G1 =&g
; ZGC
怎么选择垃圾回收器

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 J
M 的性能有一个很大的提升。
怎么选择垃圾收集器?

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
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

<h2 id="GC日志分析">GC 日志分析</h2>
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
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 — 04T21 ：
53：59.234+0800 ）
— XX： +PrintHeapAtGC 在进行 GC 的前后打印出堆的信息
— Xloggc：. . /logs/gc. log 日志文件的输出路径

<h3 id="-PrintGC">+PrintGC</h3>

打开 GC 日志：— verbose：gc
这个只会显示总的 GC 堆的变化， 如下：

<pre><code class="highlight-chroma"><span class="highlight
cl">[GC （Allocation Failure） 80832K—>19298K（227840K），0.0084018 secs]
[GC （Metadata
C Threshold） 109499K—>21465K （228352K），0.0184066 secs]
[Full GC （Metada
a GC Threshold） 21 465K—>16716K （201728K），0.0619261 secs]
</code></pre>

参数解析：

<pre><code class="highlight-chroma"><span class="highlight
cl">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 Failur
： GC发生的原因。
80832K—> 19
98K：堆在GC前的大小和GC后的大小。
228840k：现在的
大小。
0.0084018 secs：
C持续的时间。
</code></pre>
<h3 id="PrintGCDetails">PrintGCDetails</h3>
-打开 GC 日志： <code>—verbose：gc—XX： +PrintGCDetaiis</code>

输入信息如下：

<pre><code class="highlight-chroma"><span class="highlight
cl">[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
C Threshold） [PSYoungGen：98859K—>8154K（142336K）] 108760K—>21261K （22
352K），
0.0151573 secs] [T
mes： user=0.00 sys=0.01， real=0.02 secs]
[Full GC （Metada
a GC Threshold） [PSYoungGen： 8154K—>0K（142336K）] [ParOldGen： 13107K—>
6809K（62464K）] 21261K—>16809K （204800K），[Metaspace： 20599K—>20599K
（1067008K）]，0.0639732 secs]
[Times： user=0.1
sys=0.00， real=0.06 secs]
</code></pre>

参数解析：

<pre><code class="highlight-chroma"><span class="highlight
cl">GC，Full GC：同样是GC的类型
Allocation Failur
： GC原因
PSYoungGen：
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
了Parallel Old并行垃圾收集器的老年代GC前后大小的变化
Metaspace： 元
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
费的时间
Times：user： 指
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
到结束的时间，包括其他进程占用时间片的实际时间。
</code></pre>
<h3 id="PrintGCTimeStamps">PrintGCTimeStamps</h3>

打开 GC 日志：<code>—verbose：gc —XX： +PrintGCDetails —XX：+PrintGCTimeStamp
— XX： +PrintGCDateStamps</code>
输入信息如下：

<pre><code class="highlight-chroma">2019—09—24T22：15：24.518+0800：3.287： [GC（Allocation Failure） [PSYoungGen： 1361 62K—>5113K（136192K）] 141425K—>17632K （222208K） ，0.0248249 secs] [T mes： user=0.05sys=0.00， real=0.03 secs]
2019—09—24T2 ：15：25.559+0800：4.329： [GC（Metadata GC Threshold）[PSYoungGen：97578K—>1 068K（274944K）] 110096K—>22658K （360960K），0.0094071 secs]
[Times： user=0. 0sys=0.00， real=0. 01 secs]
2019—09—24T2 ：15：25.569+0800：4.338： [Full GC （Metadata GC Threshold）[PSYoungGen：10068K— gt;0K（274944K）] [ParoldGen： 12590K—>13564K （56320K）] 22658K—>13564K 331264K） ，
[Metaspace： 205 0K—>20590K（1067008K）]， 0. 0494875 secs]
[Times： user=0.1 sys=0. 02，real=0.05 secs]
</code></pre>
说明：带上了日期和时间
<h3 id="补充说明">补充说明</h3>

"[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"StopThe W rld"
使用 Serial 收集器在新生代的名字是 De fault New Generation， 因此显示的是" [DefNew"</l >
使用 ParNew 收集器在新生代的名字会变成" 【ParNew"，意思是"Parallel New Generation"</l >
使用 Parallel Scavenge 收 集器在新生代的名字是" 【PSYoungGen"
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 G1 收集器的话，会显示为"garbage 一 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。
[PSYoungGen： 5986K 一 >696K（8704K）] 5986K 一 > 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 会超过 real 时间

<img src="https://ld246.com/images/img-loading.svg" alt="172f892280ce063f.png" dat -src="https://b3logfile.com/file/2020/10/172f892280ce063f-27dbf0a7.png?imageView2/2/int rlace/1/format/jpg">
<h4 id="Minor-GC">Minor GC</h4>
<img src="https://ld246.com/images/img-loading.svg" alt="172f892ade579903.png" dat -src="https://b3logfile.com/file/2020/10/172f892ade579903-a851f28f.png?imageView2/2/int rlace/1/format/jpg">
<h4 id="Full-GC">Full GC</h4>
<img src="https://ld246.com/images/img-loading.svg" alt="172f892f35470b6d.png" dat -src="https://b3logfile.com/file/2020/10/172f892f35470b6d-2965e33a.png?imageView2/2/in erlace/1/format/jpg">
<h4 id="例">例</h4>
<pre><code class="highlight-chroma">/**
 * 在jdk7 和 jdk8 分别执行

```
</span></span><span class="highlight-line"><span class="highlight-cl"> * -verbose:gc -Xs20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+UseSerialGC
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class GCLoTest1 {
</span></span><span class="highlight-line"><span class="highlight-cl">    private static final int _1MB = 1024 * 1024;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static void testAllocation() {
</span></span><span class="highlight-line"><span class="highlight-cl">        byte[] allocation1, allocation2, allocation3, allocation4;
</span></span><span class="highlight-line"><span class="highlight-cl">        allocation1 = new byte[2 * _1MB];
</span></span><span class="highlight-line"><span class="highlight-cl">        allocation2 = new byte[2 * _1MB];
</span></span><span class="highlight-line"><span class="highlight-cl">        allocation3 = new byte[2 * _1MB];
</span></span><span class="highlight-line"><span class="highlight-cl">        allocation4 = new byte[4 * _1MB];
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static void main(String[] agrs) {
</span></span><span class="highlight-line"><span class="highlight-cl">        testAllocation();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>
<p>
<h3 id="日志分析工具使用">日志分析工具使用</h3>
<p>可以用一些工具去分析这些 gc 日志。<br>
常用的日志分析.工具有： GCViewer、GCEasy、GCHisto、GCLogViewer 、Hpjmeter、garbageca 等。</p>
<h2 id="垃圾回收器的新发展">垃圾回收器的新发展</h2>
<p>GC 仍然处于飞速发展之中，目前的默认选项 G1 GC 在不断的进行改进，很多我们原来认为的点，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。</p>
<p>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，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。</p>
<p>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 JDK9 中已经被标记为废弃，并在 JDK14 版本中移除。</p>
<h3 id="JDK11-新特性">JDK11 新特性</h3>
<ul>
<li>JEP318 :<br>
Epsilon: A No — Op Garbage<br>
Collector (Epsilon 垃圾回收器,"No — Op (无操作) "回收器)<br>
http: / /openidk.java.net/ieps/318</li>
```

JEP333:
ZGC: A Scalable Low — Latency ;Garbage Collector
(Experimental) (ZGC:可伸縮的低延退竝坂回收器，处于试验性阶段)

<h3 id="Open-JDK12的Shenandoah-GC">Open JDK12 的 Shenandoah GC</h3>

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 （JDK11 出现）和 Shenandoah（Open JDK12）。

 主打特点：低停顿时间

Open JDK12 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 Hotpot 垃圾收集器。不可避免的受到官方的排挤。比如号称 OpenJDK 和 OracleJDK 没有区别的 Oracle 公司仍拒绝在 OracleJDK12 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 PauselessGC 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 OpenJDK。
Red Hat 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。

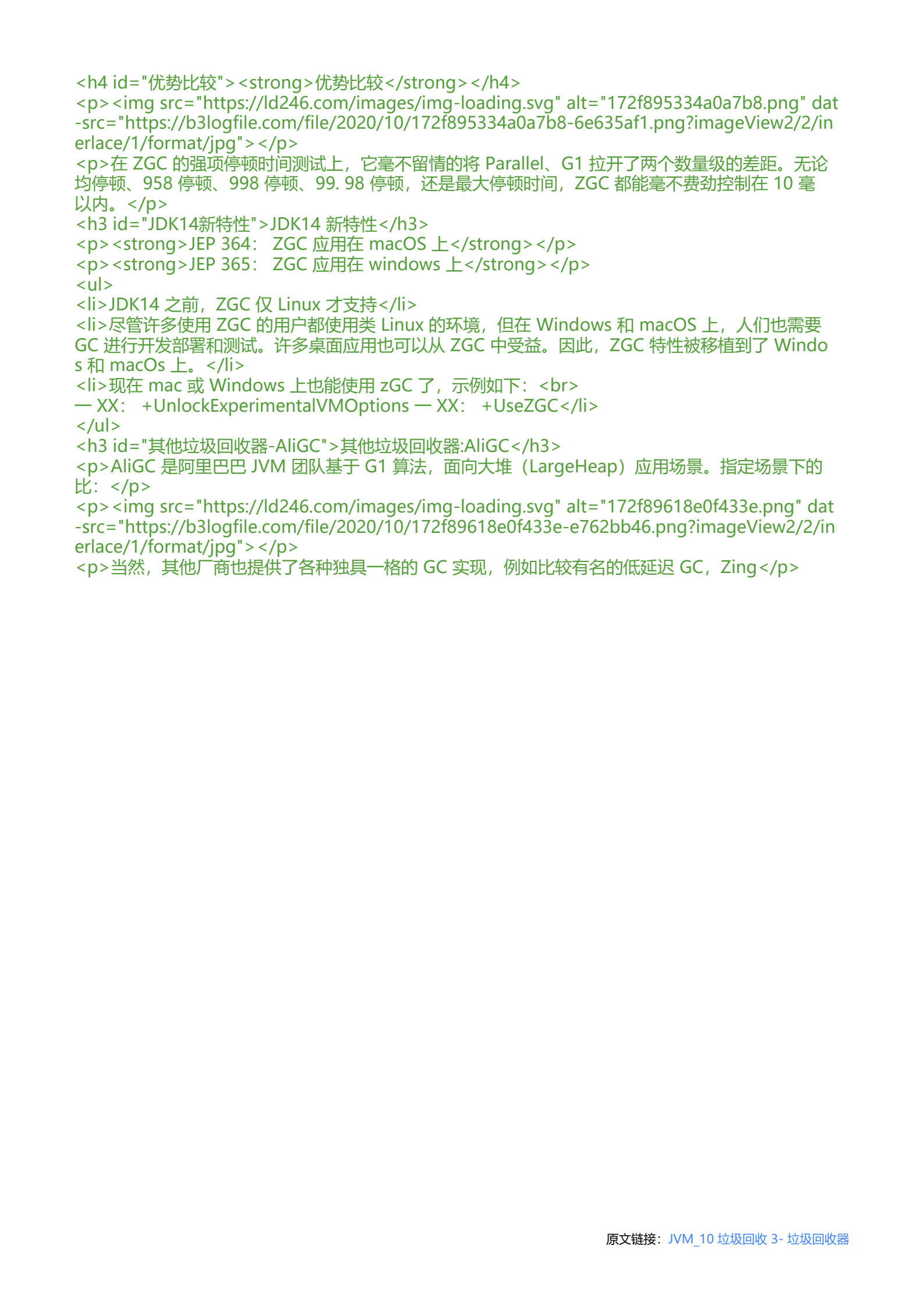
<img src="https://ld246.com/images/img-loading.svg" alt="172f8949968d9d3e.png" data-src="https://b3logfile.com/file/2020/10/172f8949968d9d3e-5f17a068.png?imageView2/2/interlace/1/format/jpg">

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。
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。

Shenandoah GC 的弱项：高运行负担下的吞吐量下降。
Shenandoah GC 的强项：低延迟时间。

<h3 id="革命性的ZGC">革命性的 ZGC</h3>
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，（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一压缩法的，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：并发标记一并发预备重分配一并发重分配一并发重映射等
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标记上，这部分的实际时间是非常少的。
测试数据如图:
<img src="https://ld246.com/images/img-loading.svg" alt="172f894ed354442d.png" data-src="https://b3logfile.com/file/2020/10/172f894ed354442d-24ad5363.png?imageView2/2/nterlace/1/format/jpg">

<h4 id="优势比较">优势比较</h4>

在 ZGC 的强项停顿时间测试上，它毫不留情的将 Parallel、G1 拉开了两个数量级的差距。无论均停顿、958 停顿、998 停顿、99. 98 停顿，还是最大停顿时间，ZGC 都能毫不费劲控制在 10 毫以内。
<h3 id="JDK14新特性">JDK14 新特性</h3>
JEP 364： ZGC 应用在 macOS 上
JEP 365： ZGC 应用在 windows 上

JDK14 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 GC 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 Windos 和 macOs 上。
现在 mac 或 Windows 上也能使用 zGC 了，示例如下：
— XX： +UnlockExperimentalVMOptions — XX： +UseZGC

<h3 id="其他垃圾回收器-AliGC">其他垃圾回收器:AliGC</h3>
AliGC 是阿里巴巴 JVM 团队基于 G1 算法，面向大堆（LargeHeap）应用场景。指定场景下的比：

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