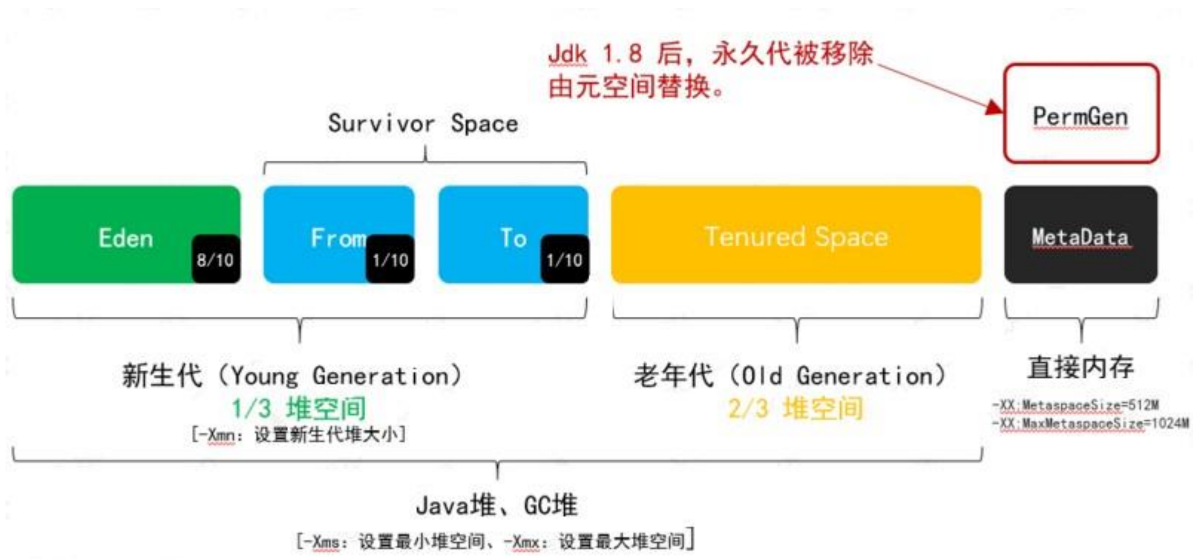


垃圾回收算法：标记清除、复制（多为新生代垃圾回收使用）、标记整理

3. 如何解决线上gc频繁的问题?

1. 查看监控，以了解出现问题的时间点以及当前FGC的频率（可对比正常情况看频率是否正常）
2. 了解该时间点之前有没有程序上线、基础组件升级等情况。
3. 了解JVM的参数设置，包括：堆空间各个区域的大小设置，新生代和老年代分别采用了哪些垃圾收集器，然后分析JVM参数设置是否合理。
4. 再对步骤1中列出的可能原因做排除法，其中元空间被打满、内存泄漏、代码显式调用gc方法比较容易排查。
5. 针对大对象或者长生命周期对象导致的FGC，可通过 jmap -histo 命令并结合dump堆内存文件作进一步分析，需要先定位到可疑对象。
6. 通过可疑对象定位到具体代码再次分析，这时候要结合GC原理和JVM参数设置，弄清楚可疑对象是否满足了进入到老年代的条件才能下结论。

4. 描述一下class初始化过程?

一个类初始化就是执行clinit()方法，过程如下

- 父类初始化
- static变量初始化/static块（按照文本顺序执行）

*Java Language Specification*中，类初始化详细过程如下（最重要的是类初始化是线程安全的）：

1. 每个类都有一个初始化锁LC，进程获取LC（如果没有获取到，就一直等待）
2. 如果C正在被其他线程初始化，释放LC并等待C初始化完成
3. 如果C正在被本线程初始化，即**递归初始化**，释放LC
4. 如果C已经被初始化了，释放LC
5. 如果C处于erroneous状态，释放LC并抛出异常NoClassDefFoundError
6. 否则，将C标记为正在被本线程初始化，释放LC；然后， 初始化那些final且为基础类型的类成员变量
7. 初始化C的父类SC和各个接口SI_n（按照implements子句中的顺序来）；如果SC或SIn初始化过程中抛出异常，则获取LC，将C标记为erroneous，并通知所有线程，然后释放LC，然后再抛出同样的异常。
8. 从classloader处获取assertion是否被打开
9. 接下来， 按照文本顺序执行类变量初始化和静态代码块，或接口的字段初始化，把它们当作是一个个单独的代码块。
10. 如果执行正常，获取LC，标记C为已初始化，并通知所有线程，然后释放LC

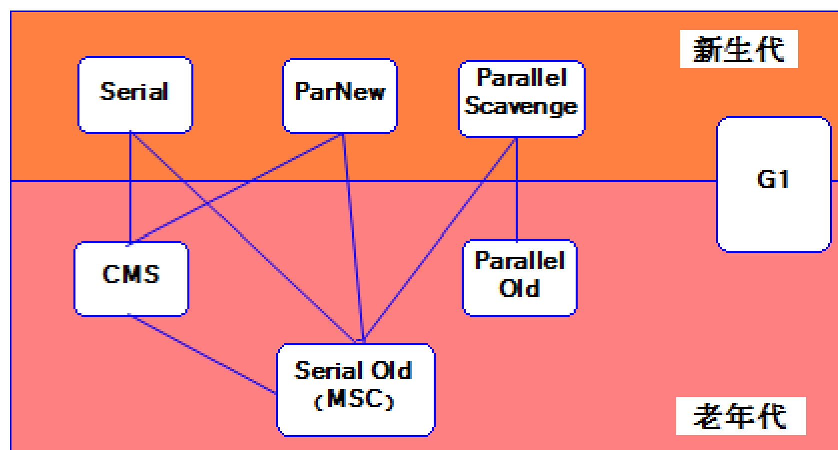
11. 否则，如果抛出了异常E。若E不是Error，则以E为参数创建新的异常ExceptionInInitializerError作为E。如果因为OutOfMemoryError导致无法创建ExceptionInInitializerError，则将OutOfMemoryError作为E。
12. 获取LC，将C标记为erroneous，通知所有等待的线程，释放LC，并抛出异常E。

5. 简述一下内存溢出的原因，如何排查线上问题？

内存溢出的原因

- java.lang.OutOfMemoryError: ......java heap space. ......堆栈溢出，代码问题的可能性极大
- java.lang.OutOfMemoryError: GC over head limit exceeded 系统处于高频的GC状态，而且回收的效果依然不佳的情况，就会开始报这个错误，这种情况一般是产生了很多不可以被释放的对象，有可能是引用使用不当导致，或申请大对象导致，但是java heap space的内存溢出有可能提前不会报这个错误，也就是可能内存就直接不够导致，而不是高频GC.
- java.lang.OutOfMemoryError: PermGen space jdk1.7之前才会出现的问题 ，原因是系统的代码非常多或引用的第三方包非常多、或代码中使用了大量的常量、或通过intern注入常量、或者通过动态代码加载等方法，导致常量池的膨胀
- java.lang.OutOfMemoryError: Direct buffer memory 直接内存不足，因为jvm垃圾回收不会回收掉直接内存这部分的内存，所以可能原因是直接或间接使用了ByteBuffer中的allocateDirect方法的时候，而没有做clear
- java.lang.StackOverflowError - Xss设置的太小了
- java.lang.OutOfMemoryError: unable to create new native thread 堆外内存不足，无法为线程分配内存区域
- java.lang.OutOfMemoryError: request {} byte for {}out of swap 地址空间不够

6. jvm有哪些垃圾回收器，实际中如何选择？



图中展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，则说明它们可以搭配使用。虚拟机所处的区域则表示它是属于新生代还是老年代收集器。
新生代收集器（全部的都是复制算法）：Serial、ParNew、Parallel Scavenge
老年代收集器：CMS（标记-清理）、Serial Old（标记-整理）、Parallel Old（标记整理）
整堆收集器： G1（一个Region中是标记-清除算法，2个Region之间是复制算法）

同时，先解释几个名词：

1，**并行（Parallel）**：多个垃圾收集线程并行工作，此时用户线程处于等待状态

2，**并发（Concurrent）**：用户线程和垃圾收集线程同时执行

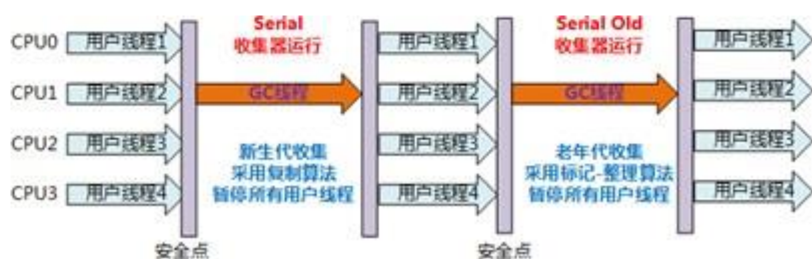
3，**吞吐量**：运行用户代码时间／（运行用户代码时间＋垃圾回收时间）

1.Serial收集器是最基本的、发展历史最悠久的收集器。**

特点：单线程、简单高效（与其他收集器的单线程相比），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手机效率。收集器进行垃圾回收时，必须暂停其他所有的工作线程，直到它结束（Stop The World）。

应用场景：适用于Client模式下的虚拟机。

Serial / Serial Old收集器运行示意图



2.**ParNew收集器其实就是Serial收集器的多线程版本。**

除了使用多线程外其余行为均和Serial收集器一模一样（参数控制、收集算法、Stop The World、对象分配规则、回收策略等）。

特点：多线程、ParNew收集器默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境中，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

和Serial收集器一样存在Stop The World问题

应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为它是除了Serial收集器外，唯一一个能与CMS收集器配合工作的。

*ParNew/Serial Old*组合收集器运行示意图如下：

3.**Parallel Scavenge 收集器与吞吐量关系密切，故也称为吞吐量优先收集器。**

特点：属于新生代收集器也是采用复制算法的收集器，又是并行的多线程收集器（与ParNew收集器类似）。

该收集器的目标是达到一个可控制的吞吐量。还有一个值得关注的点是：GC自适应调节策略（与ParNew收集器最重要的一个区别）

GC自适应调节策略：Parallel Scavenge收集器可设置-XX:+UseAdptiveSizePolicy参数。当开关打开时不需要手动指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThreshold）等，虚拟机会根据系统的运行状况收集性能监控信息，动态设置这些参数以提供最优的停顿时间和最高的吞吐量，这种调节方式称为GC的自适应调节策略。

Parallel Scavenge收集器使用两个参数控制吞吐量：

- XX:MaxGCPauseMillis 控制最大的垃圾收集停顿时间
- XX:GCRatio 直接设置吞吐量的大小。

4. **Serial Old是Serial收集器的老年代版本。**

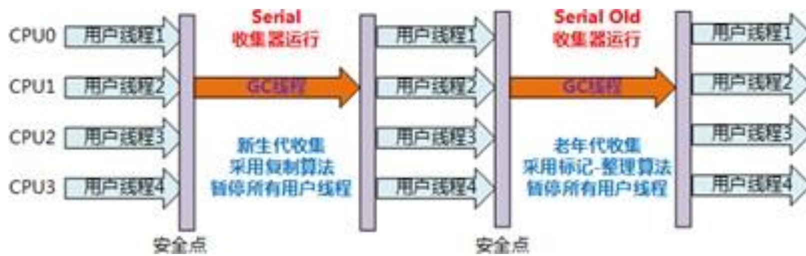
特点：同样是单线程收集器，采用标记-整理算法。

应用场景：主要也是使用在Client模式下的虚拟机中。也可在Server模式下使用。

Server模式下主要的两大用途（在后续中详细讲解···）：

1. 在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用。
2. 作为CMS收集器的后备方案，在并发收集Concurent Mode Failure时使用。

Serial / Serial Old收集器工作过程图（Serial收集器图示相同）：



5.**Parallel Old是Parallel Scavenge收集器的老年代版本。**

特点：多线程，采用标记-整理算法。

应用场景：注重高吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge+Parallel Old 收集器。

*Parallel Scavenge/Parallel Old*收集器工作过程图：

6.**CMS收集器是一种以获取最短回收停顿时间为目标的收集器。**

特点：基于标记-清除算法实现。并发收集、低停顿。

应用场景：适用于注重服务的响应速度，希望系统停顿时间最短，给用户带来更好的体验等场景下。如web程序、b/s服务。

CMS收集器的运行过程分为下列4步：

初始标记：标记GC Roots能直接到的对象。速度很快但是仍存在Stop The World问题。

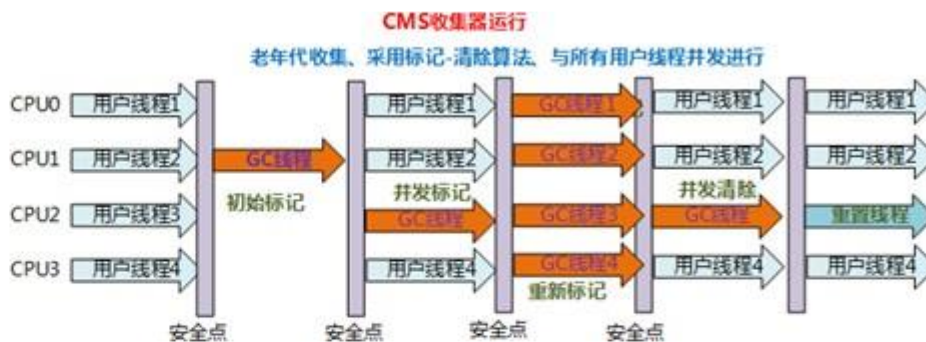
并发标记：进行GC Roots Tracing 的过程，找出存活对象且用户线程可并发执行。

重新标记：为了修正并发标记期间因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。仍然存在Stop The World问题。

并发清除：对标记的对象进行清除回收。

CMS收集器的内存回收过程是与用户线程一起并发执行的。

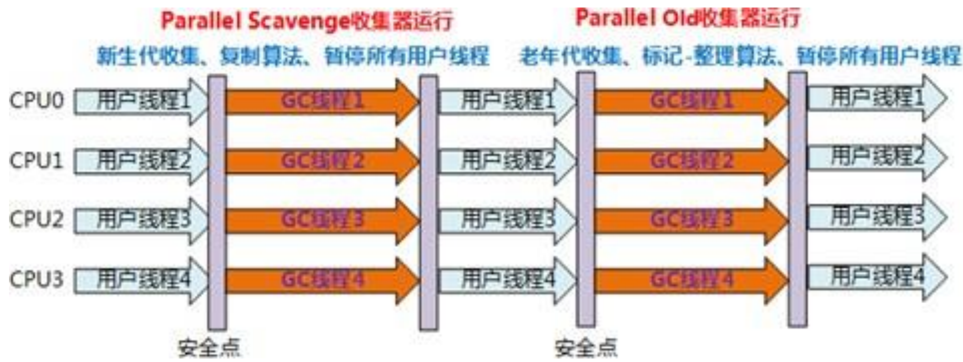
CMS收集器的工作过程图：



CMS收集器的缺点：

- 对CPU资源非常敏感。

- 无法处理浮动垃圾，可能出现Concurrent Model Failure失败而导致另一次Full GC的产生。
- 因为采用标记-清除算法所以会存在空间碎片的问题，导致大对象无法分配空间，不得不提前触发一次Full GC。



**

**

7.**G1收集器一款面向服务端应用的垃圾收集器。**

特点如下：

并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分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程序继续运行。

分代收集：G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整合：G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。

可预测的停顿：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毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

G1收集器运行示意图：

**

**

关于gc的选择
除非应用程序有非常严格的暂停时间要求，否则请先运行应用程序并允许VM选择收集器（如果没有特别要求。使用VM提供给的默认GC就好）。
如有必要，调整堆大小以提高性能。 如果性能仍然不能满足目标，请使用以下准则作为选择收集器的起点：

  - 如果应用程序的数据集较小（最大约100 MB），则选择带有选项-XX：+ UseSerialGC的串行收集器。
  - 如果应用程序将在单个处理器上运行，并且没有暂停时间要求，则选择带有选项-XX：+ UseSerialGC的串行收集器。

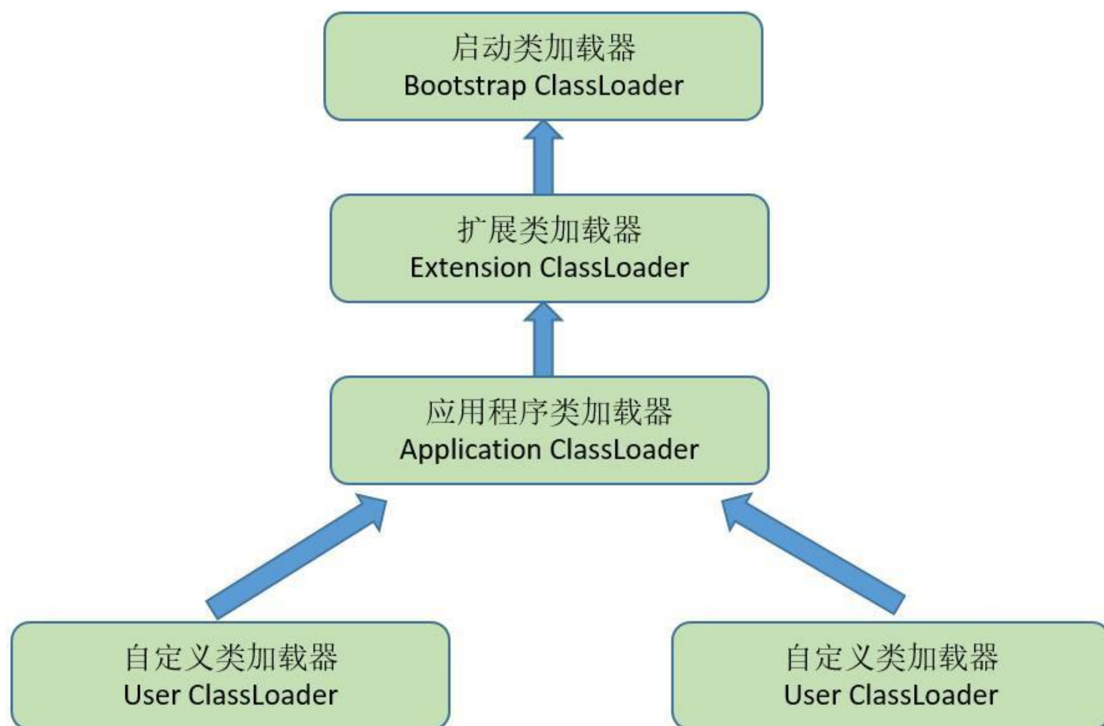
- 如果（a）峰值应用程序性能是第一要务，并且（b）没有暂停时间要求或可接受一秒或更长时间的暂停，则让VM选择收集器或使用-XX：+ UseParallelGC选择并行收集器 。
- 如果响应时间比整体吞吐量更重要，并且垃圾收集暂停时间必须保持在大约一秒钟以内，则选择具有-XX：+ UseG1GC。（值得注意的是JDK9中CMS已经被Deprecated，不可使用！移除该选项）
- 如果使用的是jdk8，并且堆内存达到了16G，那么推荐使用G1收集器，来控制每次垃圾收集的时间。
- 如果响应时间是高优先级，或使用的堆非常大，请使用-XX：UseZGC选择完全并发的收集器。（值得注意的是JDK11开始可以启动ZGC，但是此时ZGC具有实验性质，在JDK15中[202009发布]才取消实验性质的标签，可以直接显示启用，但是JDK15默认GC仍然是G1）

这些准则仅提供选择收集器的起点，因为性能取决于堆的大小，应用程序维护的实时数据量以及可用处理器的数量和速度。

如果推荐的收集器没有达到所需的性能，则首先尝试调整堆和新生代大小以达到所需的目标。 如果性能仍然不足，尝试使用其他收集器

总体原则：减少STOP THE WORD时间，使用并发收集器（比如CMS+ParNew，G1）来减少暂停时间，加快响应时间，并使用并行收集器来增加多处理器硬件上的总体吞吐量。

7. 简述一下Java类加载模型？



双亲委派模型
在某个类加载器加载class文件时，它首先委托父加载器去加载这个类，依次传递到顶层类加载器(Bootstrap)。如果顶层加载不了（它的搜索范围中找不到此类），子加载器才会尝试加载这个类。
双亲委派的好处

- 每一个类都只会被加载一次，避免了重复加载
- 每一个类都会被尽可能的加载（从引导类加载器往下，每个加载器都可能会根据优先次序尝试加载它）

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/966200025030010135