1、关键字说明：

1.1、GC Roots

找到“GC Roots”也是要花很长的时间，然而这里又有新的解决方法，就是通过采用一个OopMap的数据结构来记录系统中存活的“GC Roots”，在类加载完成的时候，虚拟机就把对象内什么偏移量上是什么类型的数据计算出来保存在OopMap，通过解释OopMap就可以找到堆中的对象，这些对象就是GC Roots。而不需要一个一个的去判断某个内存位置的值是不是引用。这种方式也叫准确式GC。

1.2、Stop the World

GC停顿，将非GC线程相关的用户线程临时挂起，实现对象状态的一致性，不能存在GC过程中，对象的状态飘忽不定的状态，做到准确GC。

1.3、Safe Point

程序并非在所有地方都能停下来开始GC，只有到达某个特定的位置才可以暂停，这个特定的位置就称为Safe Point。抢断式中断，首先爸所有线程全部中断，如果有线程并不处于安全点，那就恢复这个线程，让它跑到安全点上---抢先式中断；不直接对线程操作，使用标志信息，各个线程论询这个标志，发现标志为真是就挂起。轮询标志的地方和安全点是重合的—主动式中断。

1.4、Safe Region

被扩展的Safe Point，在这个区域内被认为是引用关系是不改变的，可以随时进行GC。

2、常见的垃圾回收算法

2.1、标记-清除法

是最基础的算法，后续的算法都是在此基础上不断完善和改进完成的。这个算法主要有两个不足：1、效率问题，不管是标记还是清除，效率都不高；2、空间问题，标记清除之后，会留下很多不连续的内存碎片，而空间碎片的增多不利于大对象的分配，可能会提前触发一次GC。

2.2、复制算法

将内存空间划分为大小想相等的两块，每次仅对其中的一块进行操作，当其中的一块使用完了，就将存活的对象复制到另一块上，随后将使用过的那块一次性清除干净。

这样看，一次只操作一块内存，将近50%的空间，显得太浪费了。其实，研究表明，新生代中的对象98%的属于朝生夕死的对象，所以并不需要按照1:1的形式来划分，而是按照这个比例进行划分：8:1:1的形式；

两个问题：1、为什么两块survivor区？2、survivor满了怎么办（分配担保）？

2.3、标记-整理法

标记整理算法的“标记”过程和标记-清除算法一致，只是后面并不是直接对可回收对象进行整理，而是让所有存活的对象都向一段移动，然后直接清理掉端边界意外的内存。

3、垃圾收集器

前面对几种常见的垃圾收集算法进行了简单的说明，下面所要讲到的垃圾收集器其实可看做是垃圾收集算法的具体实现。

3.1、Serial 收集器

一款采用复制算法思想实现的串行收集器，是一款发展较久的基本的收集器，一般用于新生代的垃圾收集；

3.2、parNew 收集器

是Serial收集器的多线程版本，二者在实现上共用了很多的代码；同样采用复制算法进行收集；server模式下的虚拟机首选的垃圾收集器；

3.3、**Parallel Scavenge收集器**

年轻代并行垃圾收集器，同样采用了复制算法进行实现；看起来和刚才介绍的parNew收集器是一致，其实Parallel Scavenge收集器更注重吞吐量。

先讲下什么是吞吐量。吞吐量的计算公式是：吞吐量=运行用户代码时间/(运行用户代码时间+GC消耗时间)；就是说，GC时间越短，吞吐量越大。调整吞吐量涉及两个参数，第一个参数，最大停顿时间，单位毫秒；每次发生GC时，会尽量在这个时间内进行。当然，这个值并不是越小越好，停顿时间缩短是以牺牲新生代空间和吞吐量来实现的，试想一下，一个更小的新生代虽然GC时间缩短了，但是却导致了更加频繁的GC；第二个参数：垃圾收集时间占总时间的比率，是一个0-100的整数，默认是99，计算公式为：

这个收集器和ParNew收集器还有一个显著的差异是，引入了自适应调节策略，这个参数的开关打开后，系统会自动收集性能监控信息，动态调整垃圾收集的合理停顿时间或吞吐量。我们只需要将基本的参数做好设置，比如堆内存大小，然后使用以上两个参数的其中一个进行设置，作为一个调优目标，具体细节交给jvm自动调节即可。

3.4、**Serial Old收集器**

以上介绍的是新生代的垃圾收集器，下面开始介绍老年代的垃圾收集器。第一个，Serial Old 收集器。

3.5、**CMS收集器**

CMS收集器是一款老年代比较优秀和复杂的收集器，目标是实现垃圾收集停顿时间的尽可能缩短。采用的是标记-清理算法。CMS收集器垃圾收集过程分为4个过程，一个过程，初始标记，完成GC Roots可以直接关联到对象的标记，速度很快。第二个过程，并发标记，会从GC Roots 出发，标记出所有可达的对象，这个过程可能会花费相对比较长的时间，但是由于在这个阶段，GC线程和用户线程是可以一起运行的，所以即使标记过程比较耗时，也不会影响到系统的运行；第三个过程，重新标记，是对并发标记期间因用户程序运行而导致标记变动的那部分记录进行修正，重新标记阶段耗时一般比初始标记稍长，但是远小于并发标记阶段；第四个过程，并发清除，将标记的待回收的对象按照标记清除算法进行清除，与用户线程并发进行，即使相对耗时，也不会对系统运行造成大的影响。

不足之处：

首先，CMS收集器对CPU资源非常敏感；

其次，CMS收集器在处理垃圾收集的过程中，可能会产生浮动垃圾，由于它无法处理浮动垃圾，所以可能会出现Concurrent Mode Failure问题而导致触发一次Full GC。所谓的浮动垃圾，是由于CMS收集器的并发清理阶段，清理线程是和用户线程一起运行，如果在清理过程中，用户线程产生了垃圾对象，由于过了标记阶段，所以这些垃圾对象就成为了浮动垃圾，CMS无法在当前垃圾收集过程中集中处理这些垃圾对象。由于这个原因，CMS收集器不能像其他收集器那样等到完全填满了老年代以后才进行垃圾收集，需要预留一部分空间来保证当出现浮动垃圾的时候可以有空间存放这些垃圾对象。在JDK 1.6中，CMS收集器的激活阀值变成了92%。如果在CMS运行期间没有足够的内存来存放浮动垃圾，那么就会导致"Concurrent Mode Failure"失败，这个时候，虚拟机将启动后备预案，临时启动Serial Old收集器来对老年代重新进行垃圾收集，这样会导致垃圾收集的时间边长，特别是当老年代内存很大的时候。所以对参数"-XX:CMSInitiatingOccupancyFraction"的设置，过高，会导致发生Concurrent Mode Failure，过低，则浪费内存空间。

3.6、**G1收集器**

之前介绍的几组垃圾收集器都有以下几点共同特性：

1. 年轻代、老年代都是独立连续的内存块；
2. 年轻代收集使用单eden，双survivor的复制算法进行收集；
3. 老年代收集必须扫描整个老年代区域；
4. 都是尽可能少而快地执行GC为设计原则；

下面介绍的这一种收集器与之前介绍的收集器有着很多的不同，先从G1收集器的内存模型来说起，

G1采用了分区(Region)的思路，将整个堆空间分成若干个大小相等的内存区域，每次分配对象空间将逐段地使用内存。因此，在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可；每个分区也不会确定地为某个代服务，可以按需在年轻代和老年代之间切换。启动时可以通过参数-XX:G1HeapRegionSize=n可指定分区大小(1MB~32MB，且必须是2的幂)，默认将整堆划分为2048个分区。

在每个分区内部又被分成了若干个大小为512 Byte卡片(Card)，标识堆内存最小可用粒度所有分区的卡片将会记录在全局卡片表(Global Card Table)中，分配的对象会占用物理上连续的若干个卡片，当查找对分区内对象的引用时便可通过记录卡片来查找该引用对象(见RSet)。每次对内存的回收，都是对指定分区的卡片进行处理。

在分代方面，G1依然保留着分代的逻辑概念，但是各个分代并不是连续的存储单元；一个大小达到甚至超过分区大小一半的对象称为巨型对象，巨型对象会直接在老年代分配，所占用的连续空间称为巨型分区；

在串行和并行收集器中，GC通过整堆扫描，来确定对象是否处于可达路径中。然而G1为了避免STW式的整堆扫描，在每个分区记录了一个已记忆集合(RSet)，内部类似一个反向指针，记录引用分区内对象的卡片索引。当要回收该分区时，通过扫描分区的RSet，来确定引用本分区内的对象是否存活，进而确定本分区内的对象存活情况。

收集集合(CSet)代表每次GC暂停时回收的一系列目标分区。在任意一次收集暂停中，CSet所有分区都会被释放，内部存活的对象都会被转移到分配的空闲分区中。因此无论是年轻代收集，还是混合收集，工作的机制都是一致的。

* Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。
* Mixed GC：选定所有年轻代里的Region，通过内部启发式算法，在用户指定的开销目标范围内尽可能选择收益高的老年代Region。

G1收集器回收的过程大致为以下几个阶段，

G1收集器实现可预测的停顿，基本思想是通过给垃圾标记优先级，来进行最优化的收集，其中实现方式就是Cset的设计和使用