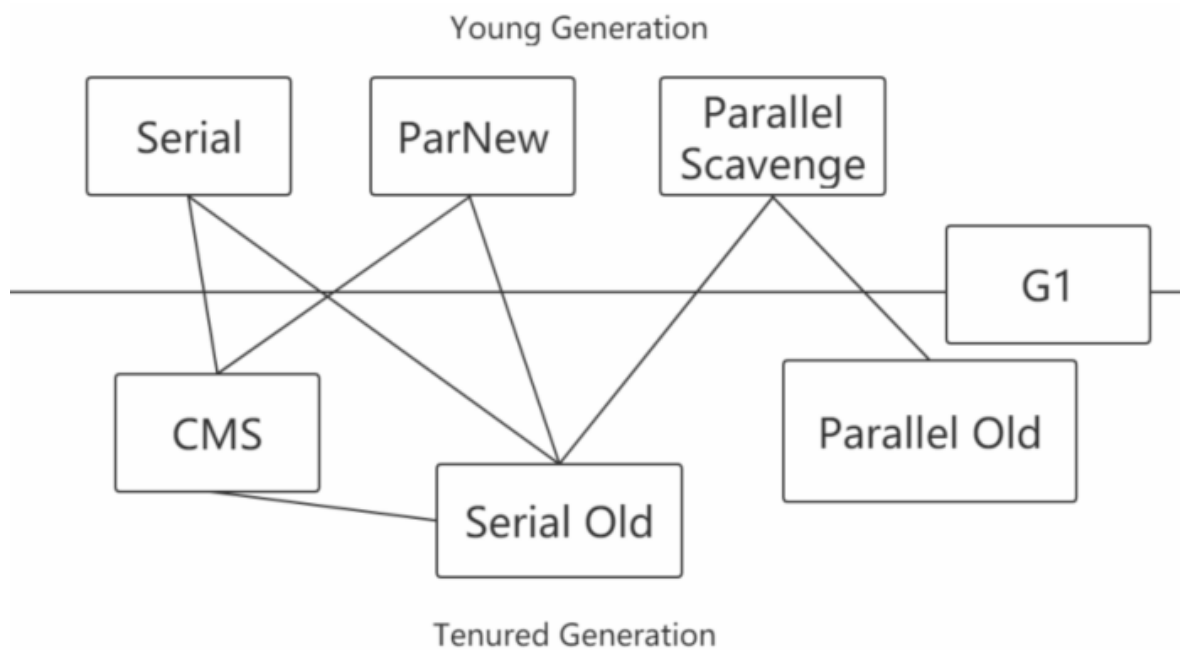


你接触过哪些垃圾收集器，聊一下

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。



- Serial

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK1.3.1之前）是虚拟机新生代收集的唯一选择。

它是一种单线程收集器，不仅仅意味着它只会使用一个CPU或者一条收集线程去完成垃圾收集工作，更重要的是其在进行垃圾收集的时候需要暂停其他线程。

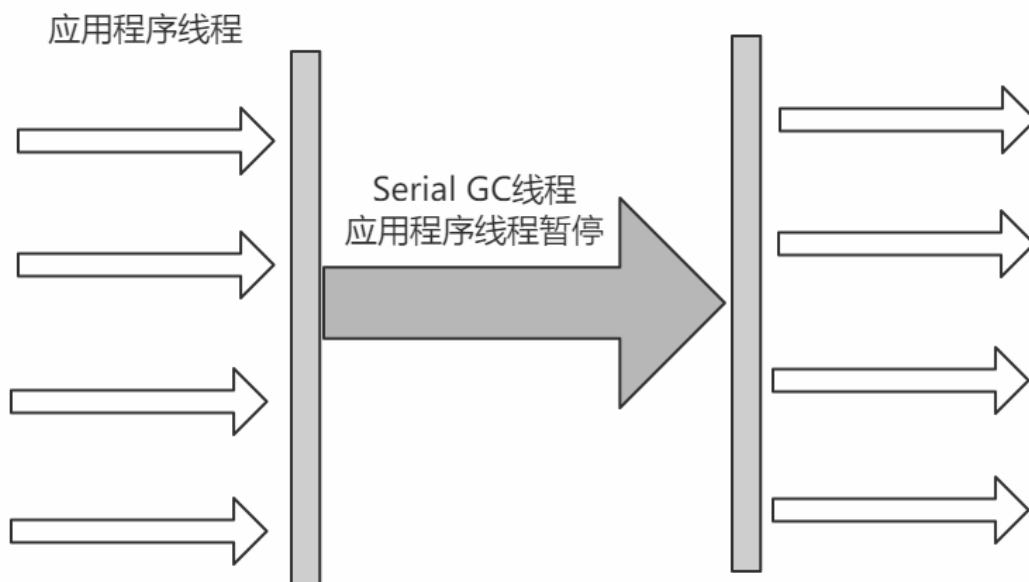
优点：简单高效，拥有很高的单线程收集效率

缺点：收集过程需要暂停所有线程

算法：复制算法

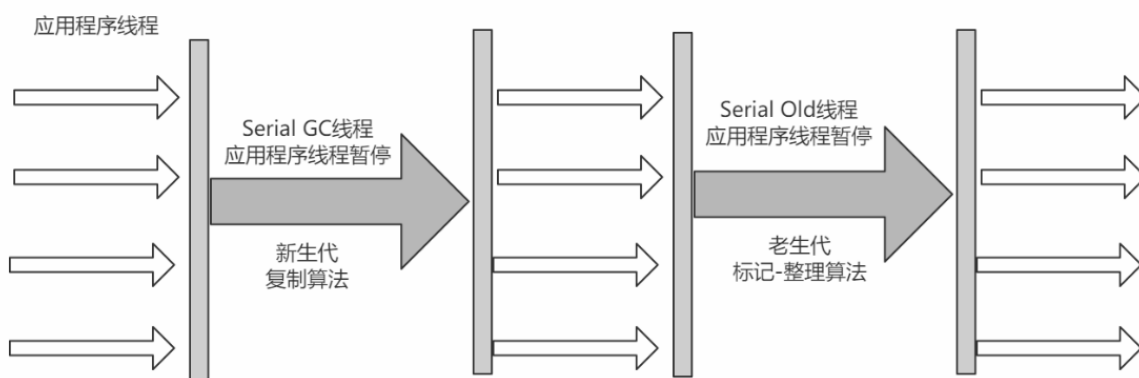
适用范围：新生代

应用：client模式下的默认新生代收集器



- Serial Old

Serial Old收集器是Serial收集器的老年代版本，也是一个单线程收集器，不同的是采用"标记-整理算法"，运行过程和Serial收集器一样。

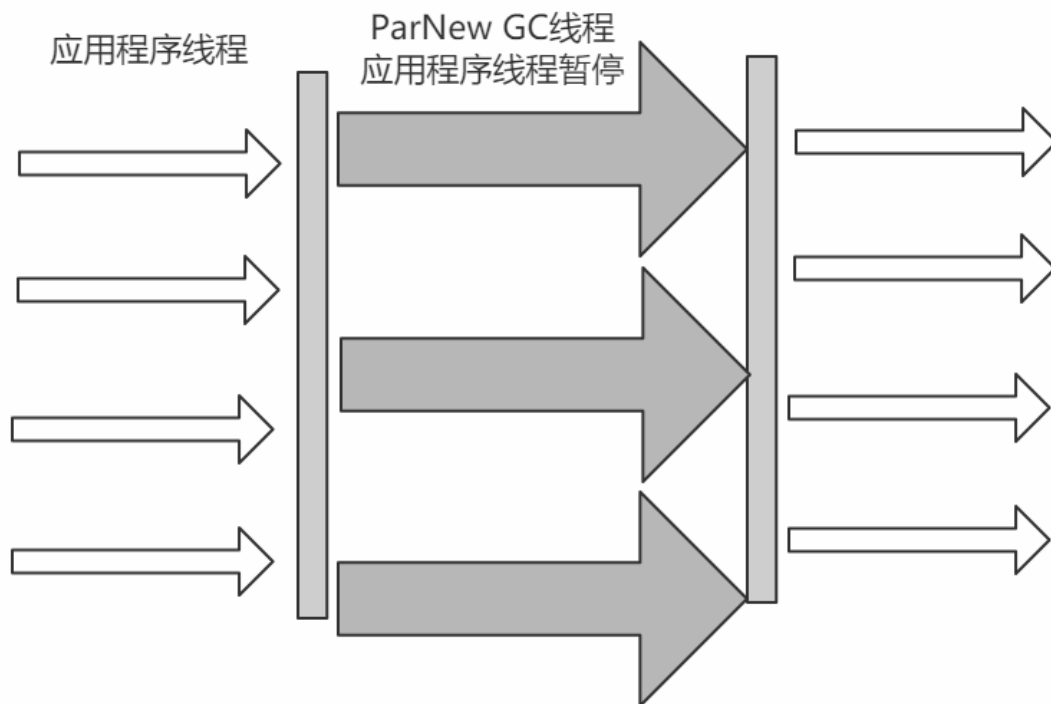


早期的垃圾收集器为什么要设计成单线程？单核CPU 嵌入式 STW

- ParNew

可以把这个收集器理解为Serial收集器的多线程版本。

优点：在多CPU时，比Serial效率高。
 缺点：收集过程暂停所有应用程序线程，单CPU时比Serial效率差。
 算法：复制算法
 适用范围：新生代
 应用：运行在Server模式下的虚拟机中首选的新生代收集器



- Parallel Scavenge STW 单位时间内接受请求并响应的数量 能不能控制

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去和ParNew一样，但是Parallel Scavenge更关注系统的吞吐量。

吞吐量=运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)

比如虚拟机总共运行了100分钟，垃圾收集时间用了1分钟，吞吐量=(100-1)/100=99%。

若吞吐量越大，意味着垃圾收集的时间越短，则用户代码可以充分利用CPU资源，尽快完成程序的运算任务。

-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。

- Parallel Old

Parallel Old收集器是**Parallel Scavenge**收集器的老年代版本，使用多线程和标记-整理算法进行垃圾回收，也是更加关注系统的吞吐量。

吞吐量 停顿时间

95%-96%以上 young GC full GC 不能小于一天/次

业务代码 + 垃圾收集线程 并发类垃圾收集器

- CMS

官网：

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#concurrent_mark_sweep cms collector

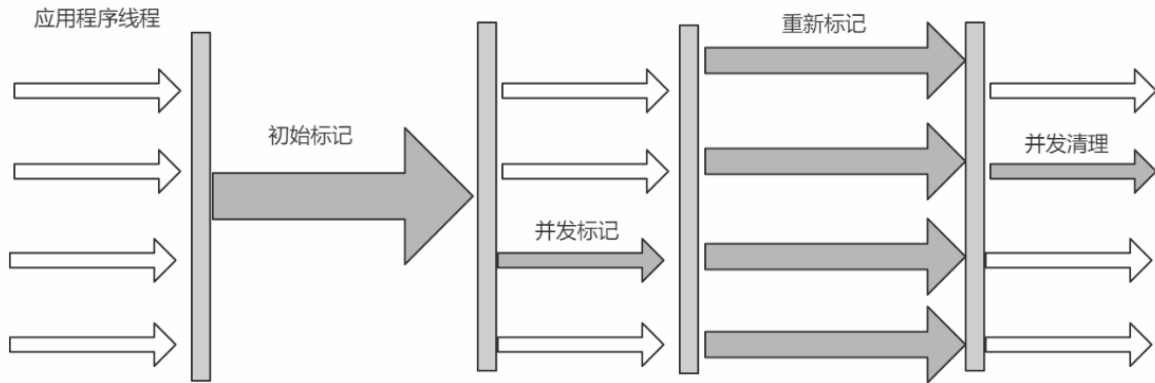
CMS(Concurrent Mark Sweep)收集器是一种以获取 最短回收停顿时间 为目标的收集器。

采用的是"标记-清除算法",整个过程分为4步

我们的CMS用什么方式尽可能的节省垃圾收集的时间

(1)初始标记 CMS initial mark	标记GC Roots直接关联对象，不用Tracing，速度很快	不耗时	STW
(2)并发标记 CMS concurrent mark	进行GC Roots Tracing	耗时	并发
(3)重新标记 CMS remark	修改并发标记因用户程序变动的内容	不耗时	STW
(4)并发清除 CMS concurrent sweep	清除不可达对象回收空间，同时有新垃圾产生，留着下次清理称为浮动垃圾		

由于整个过程中，并发标记和并发清除，收集器线程可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的。



优点：并发收集、低停顿

缺点：产生大量空间碎片、并发阶段会降低吞吐量

什么是记忆集？

当我们进行young gc时，我们的gc roots除了常见的栈引用、静态变量、常量、锁对象、class对象这些常见的之外，如果老年代有对象引用了我们的新生代对象，那么老年代的对象也应该加入gc roots的范围中，但是如果每次进行young gc我们都需要扫描一次老年代的话，那我们进行垃圾回收的代价实在是太大了，因此我们引入了一种叫做记忆集的抽象数据结构来记录这种引用关系。

记忆集是一种用于记录从非收集区域指向收集区域的指针集合的数据结构。

如果我们不考虑效率和成本问题，我们可以用一个数组存储所有有指针指向新生代的老年代对象。但是如果这样的话我们维护成本就很好，打个比方，假如所有的老年代对象都有指针指向了新生代，那么我们需要维护整个老年代大小的记忆集，毫无疑问这种方法是不可取的。因此我们引入了卡表的数据结构

卡表

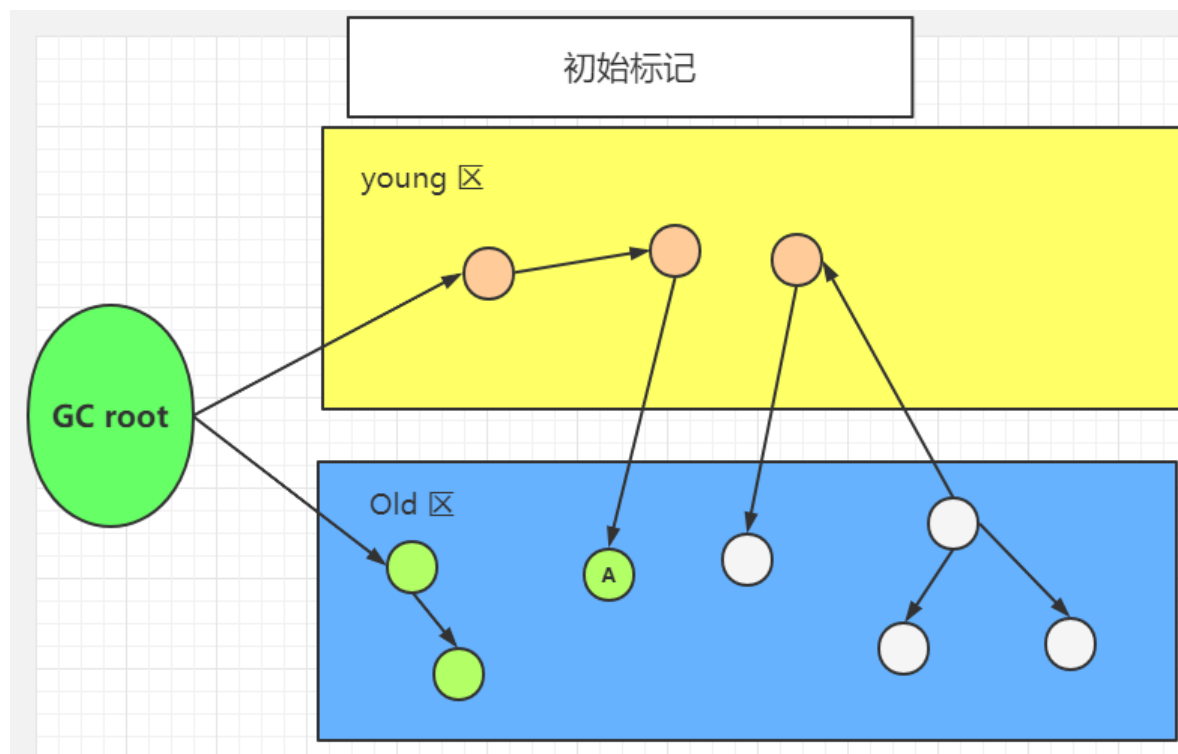
记忆集是我们针对于跨代引用问题提出的思想，而卡表则是针对于该种思想的具体实现。（可以理解为记忆集是结构，卡表是实现类）

[1字节，00001000，1字节，1字节]

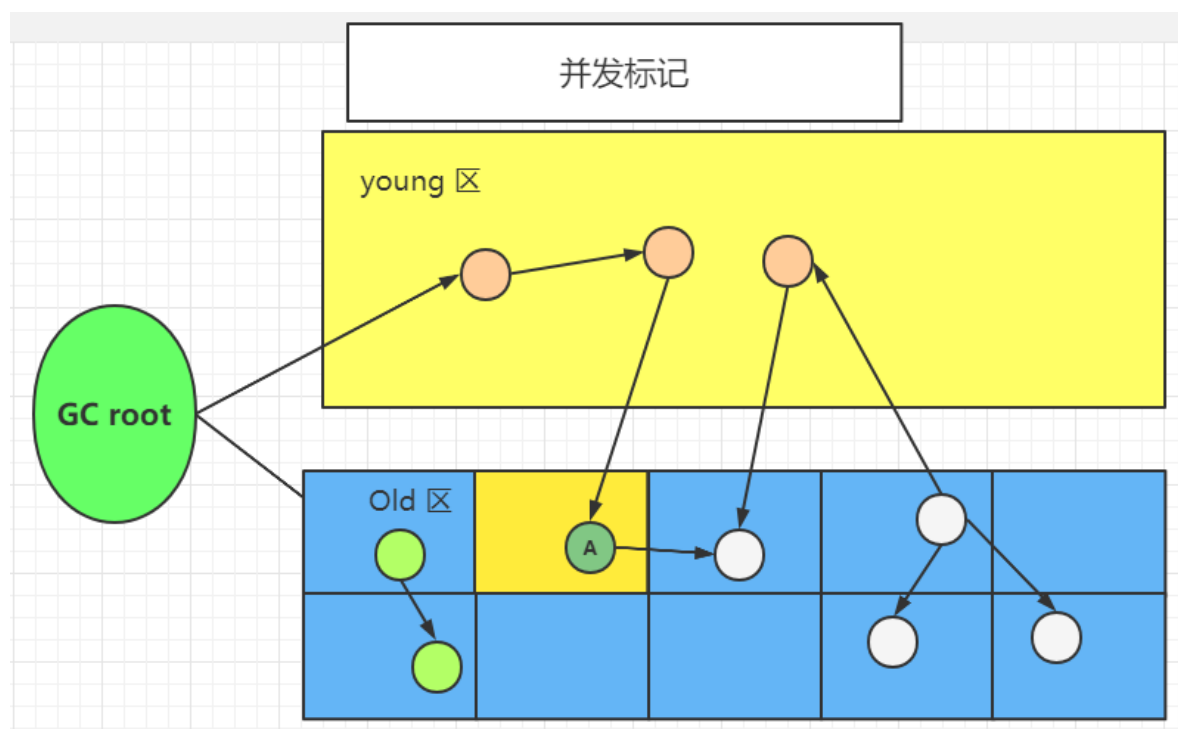
在hotspot虚拟机中，卡表是一个字节数组，数组的每一项对应着内存中的某一块连续地址的区域，如果该区域中有引用指向了待回收区域的对象，卡表数组对应的元素将被置为1，没有则置为0；

(1) 卡表是使用一个字节数组实现: CARD_TABLE[], 每个元素对应着其标识的内存区域一块特定大小的内存块, 称为"卡页"。hotSpot使用的卡页是 2^9 大小, 即512字节

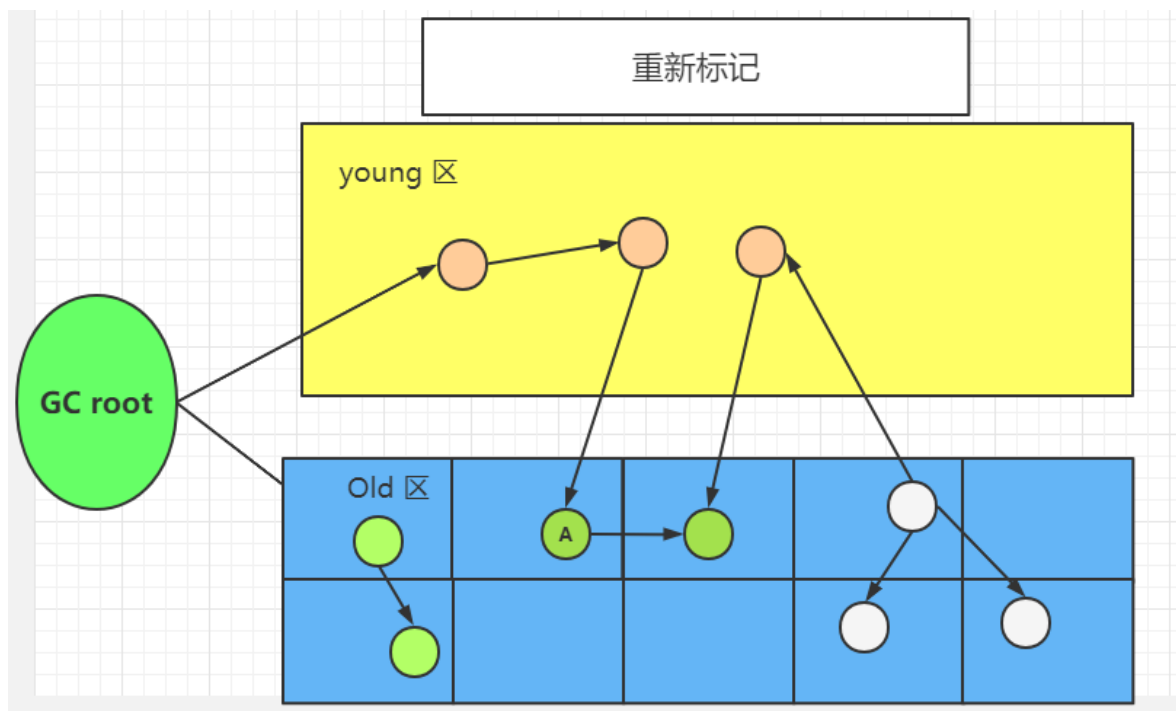
(2) 一个卡页中可包含多个对象, 只要有一个对象的字段存在跨代指针, 其对应的卡表的元素标识就变成1, 表示该元素变脏, 否则为0。GC时, 只要筛选本收集区的卡表中变脏的元素加入GC Roots里。



并发标记的时候，A对象发生了所在的引用发生了变化，所以A对象所在的块被标记为脏卡



继续往下到了重新标记阶段，修改对象的引用，同时清除脏卡标记。



卡表其他作用：

老年代识别新生代的时候

对应的card table被标识为相应的值（card table中是一个byte，有八位，约定好每一位的含义就可区分哪个是引用新生代，哪个是并发标记阶段修改过的）

- G1(Garbage-First) JDK8推荐使用的 JDK9默认的 JDK7的最后的维护版
要比CMS的停顿时间短 你想多短就多短 优先回收垃圾价值高的区域
它可以某种程度上解决空间碎片的问题

官网：

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html#garbage_first_garbage_collection

使用G1收集器时，Java堆的内存布局与就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region）2048个，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。

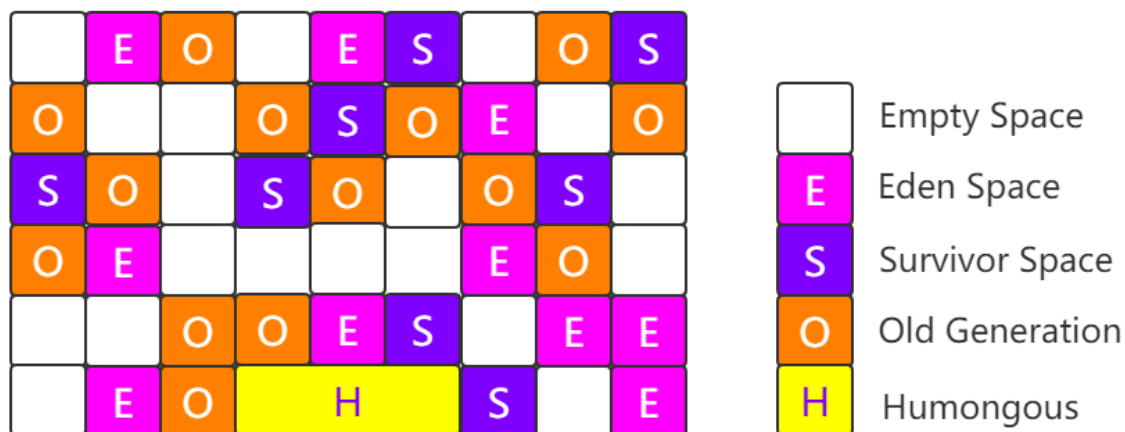
每个Region大小都是一样的，可以是1M到32M之间的数值，但是必须保证是2的n次幂

如果对象太大，一个Region放不下[超过Region大小的50%]，那么就会直接放到H中

设置Region大小：-XX:G1HeapRegionSize=M

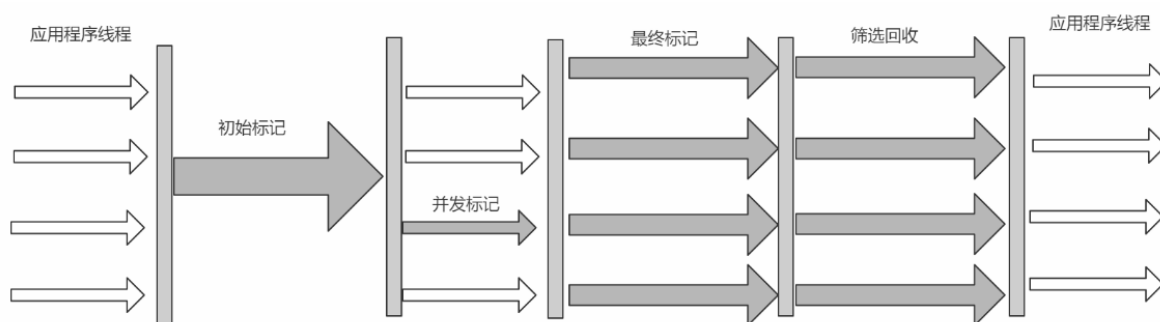
所谓Garbage-Frist，其实就是优先回收垃圾最多的Region区域

- （1）分代收集（仍然保留了分代的概念）
- （2）空间整合（整体上属于“标记-整理”算法，不会导致空间碎片）
- （3）可预测的停顿（比CMS更先进的地方在于能让使用者明确指定一个长度为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）



工作过程可以分为如下几步

初始标记 (Initial Marking)	标记以下GC Roots能够关联的对象，并且修改TAMS的值，需要暂停用户线程
并发标记 (Concurrent Marking)	从GC Roots进行可达性分析，找出存活的对象，与用户线程并发执行
最终标记 (Final Marking)	修正在并发标记阶段因为用户程序的并发执行导致变动的数据，需暂停用户线程
筛选回收 (Live Data Counting and Evacuation)	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停顿时间制定回收计划



• ZGC

官网: <https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html#GUID-A5A42691-095E-47BA-B6DC-FB4E5FAA43D0>

JDK11新引入的ZGC收集器，不管是物理上还是逻辑上，ZGC中已经不存在新老年代的概念了，会分为一个个page，当进行GC操作时会对page进行压缩，因此没有碎片问题，只能在64位的linux上使用，目前用得还比较少

- (1) 可以达到10ms以内的停顿时间要求
- (2) 支持TB级别的内存
- (3) 堆内存变大后停顿时间还是在10ms以内

JVM常用参数有哪些？

JVM参数

3.1.1 标准参数

```
-version
-help
-server
-cp
```

```
[root@localhost ~]# java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

3.1.2 -X参数

非标准参数，也就是在JDK各个版本中可能会变动

```
-Xint      解释执行
-Xcomp     第一次使用就编译成本地代码
-Xmixed    混合模式，JVM自己来决定
```

```
[root@localhost ~]# java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
[root@localhost ~]# java -Xint -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, interpreted mode)
[root@localhost ~]# java -Xcomp -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, compiled mode)
[root@localhost ~]# java -Xmixed -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

3.1.3 -XX参数

使用得最多的参数类型

非标准化参数，相对不稳定，主要用于JVM调优和Debug

a. Boolean类型

格式: `-XX:[+|-]<name>` +或-表示启用或者禁用`name`属性
比如: `-XX:+UseConcMarkSweepGC` 表示启用CMS类型的垃圾回收器
 `-XX:+UseG1GC` 表示启用G1类型的垃圾回收器

b. 非Boolean类型

格式: `-XX<name>=<value>`表示`name`属性的值是`value`
比如: `-XX:MaxGCPauseMillis=500`

3.1.4 其他参数

```
-Xms1000M等价于-XX:InitialHeapSize=1000M
-Xmx1000M等价于-XX:MaxHeapSize=1000M
-Xss100等价于-XX:ThreadStackSize=100
```


所以这块也相当于是-XX类型的参数

3.1.5 查看参数

```
java -XX:+PrintFlagsFinal -version > flags.txt
```

```
[root@localhost bin]# java -XX:+PrintFlagsFinal -version > flags.txt
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
[root@localhost bin]# sz flags.txt
```

```
[Global flags]
  intx ActiveProcessorCount           = -1           {product}
  uintx AdaptiveSizeDecrementScaleFactor = 4           {product}
  uintx AdaptiveSizeMajorGCDecayTimeScale = 10          {product}
  uintx AdaptiveSizePausePolicy         = 0           {product}
  uintx AdaptiveSizePolicyCollectionCostMargin = 50          {product}
  uintx AdaptiveSizePolicyInitializingSteps = 20          {product}
  uintx AdaptiveSizePolicyOutputInterval = 0           {product}
  uintx AdaptiveSizePolicyWeight         = 10          {product}
  uintx AdaptiveSizeThroughPutPolicy     = 0           {product}
  uintx AdaptiveTimeWeight              = 25          {product}
  bool  AdjustConcurrency                = false        {product}
  bool  AggressiveHeap                  = false        {product}
  bool  AggressiveOpts                  = false        {product}
```

值得注意的是"="表示默认值, ":="表示被用户或VM修改后的值
要想查看某个进程具体参数的值, 可以使用jinfo, 这块后面聊
一般要设置参数, 可以先查看一下当前参数是什么, 然后进行修改

3.1.6 设置参数的常见方式

- 开发工具中设置比如IDEA, eclipse
- 运行jar包的时候:java -XX:+UseG1GC xxx.jar
- web容器比如tomcat, 可以在脚本中的进行设置
- 通过jinfo实时调整某个java进程的参数(参数只有被标记为manageable的flags可以被实时修改)

3.1.7 实践和单位换算

```
1Byte(字节)=8bit(位)
1KB=1024Byte(字节)
1MB=1024KB
1GB=1024MB
1TB=1024GB
```

```
(1) 设置堆内存大小和参数打印
-Xmx100M -Xms100M -XX:+PrintFlagsFinal
(2) 查询+PrintFlagsFinal的值
:=true
(3) 查询堆内存大小MaxHeapSize
:= 104857600
(4) 换算
104857600(Byte)/1024=102400(KB)
102400(KB)/1024=100(MB)
(5) 结论
104857600是字节单位
```

3.1.8 常用参数含义

参数	含义	说明
-XX:CICompilerCount=3	最大并行编译数	如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加JVM崩溃的可能
-XX:InitialHeapSize=100M	初始化堆大小	简写-Xms100M
-XX:MaxHeapSize=100M	最大堆大小	简写-Xmx100M
-XX:NewSize=20M	设置年轻代的大小	
-XX:MaxNewSize=50M	年轻代最大大小	
-XX:OldSize=50M	设置老年代大小	
-XX:MetaspaceSize=50M	设置方法区大小	
-XX:MaxMetaspaceSize=50M	方法区最大大小	
-XX:+UseParallelGC	使用UseParallelGC	新生代，吞吐量优先
-XX:+UseParallelOldGC	使用UseParallelOldGC	老年代，吞吐量优先
-XX:+UseConcMarkSweepGC	使用CMS	老年代，停顿时间优先
-XX:+UseG1GC	使用G1GC	新生代，老年代，停顿时间优先
-XX:NewRatio	新老生代的比值	比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5
-XX:SurvivorRatio	两个S区和Eden区的比值	比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8，也就是一个S占整个新生代的1/10
-XX:+HeapDumpOnOutOfMemoryError	启动堆内存溢出打印	当JVM堆内存发生溢出时，也就是OOM，自动生成dump文件
-XX:HeapDumpPath=heap.hprof	指定堆内存溢出打印目录	表示在当前目录生成一个heap.hprof文件
-XX:+PrintGCDetails - XX:+PrintGCTimeStamps - XX:+PrintGCDateStamps -Xloggc:g1-gc.log	打印出GC日志	可以使用不同的垃圾收集器，对比查看GC情况
-Xss128k	设置每个线程的堆栈大小	经验值是3000-5000最佳
-XX:MaxTenuringThreshold=6	提升年老代的最大临界值	默认值为 15
-XX:InitiatingHeapOccupancyPercent	启动并发GC周期时堆内存使用占比	G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示“一直执行GC循环”. 默认值为 45.
-XX:G1HeapWastePercent	允许的浪费堆空间的占比	默认是10%，如果并发标记可回收的空间小于10%,则不会触发MixedGC。
-XX:MaxGCPauseMillis=200ms	G1最大停顿时间	暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。
-XX:ConcGCThreads=n	并发垃圾收集器使用的线程数量	默认值随JVM运行的平台不同而不同

参数	含义	说明
-XX:G1MixedGCLiveThresholdPercent=65	混合垃圾回收周期中要包括的旧区域设置占用率阈值	默认占用率为 65%
-XX:G1MixedGCCountTarget=8	设置标记周期完成后，对存活数据上限为 G1MixedGCLiveThresholdPercent的旧区域执行混合垃圾回收的目标次数	默认8次混合垃圾回收，混合回收的目标是要控制在此目标次数以内
-XX:G1OldCSetRegionThresholdPercent=1	描述Mixed GC时，Old Region被加入到CSet中	默认情况下，G1只把10%的Old Region加入到CSet中

JVM常用命令有哪些

jps

查看java进程

The jps command lists the instrumented Java HotSpot VMs on the target system. The command is limited to reporting information on JVMs for which it has the access permissions.

```
[root@localhost bin]# jps
3162 Jps
2908 Bootstrap
[root@localhost bin]# jps -l
3172 sun.tools.jps.Jps
2908 org.apache.catalina.startup.Bootstrap
[root@localhost bin]# jinfo -flag MaxHeapSize 2908
-XX:MaxHeapSize=257949696
```

jinfo

(1) 实时查看和调整JVM配置参数

The jinfo command prints Java configuration information for a specified Java process or core file or a remote debug server. The configuration information includes Java system properties and Java Virtual Machine (JVM) command-line flags.

(2) 查看用法

jinfo -flag name PID 查看某个java进程的name属性的值

```
jinfo -flag MaxHeapSize PID
jinfo -flag UseG1GC PID
```

```
[root@localhost bin]# jinfo -flag MaxHeapSize 2597
-XX:MaxHeapSize=257949696
[root@localhost bin]# jinfo -flag UseG1GC 2597
-XX: -UseG1GC
```

(3) 修改

参数只有被标记为manageable的flags可以被实时修改

```
jinfo -flag [+|-] PID
jinfo -flag <name>=<value> PID
```

(4) 查看曾经赋过值的一些参数

```
jinfo -flags PID
```

```
[root@localhost bin]# jinfo -flags 2908
Attaching to process ID 2908, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12
Non-default VM flags: -XX:CICompilerCount=2 -XX:InitialHeapSize=16777216 -XX:+ManagementServer -XX:MaxHeapSize=257949696
-XX:MaxNewSize=85983232 -XX:MinHeapDeltaBytes=196608 -XX:NewSize=5570560 -XX:OldSize=11206656 -XX:+UseCompressedClassPo
inters -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps
Command line: -Djava.util.logging.config.file=/usr/local/tomcat/apache-tomcat-8.5.37/conf/logging.properties -Djava.util
l.logging.manager=org.apache.juli.ClassLoaderLogManager -Djdk.tls.ephemeralDHKeySize=2048 -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8999 -Dcom.sun.management.jmxremote.authenticative=false -Dcom.sun.management.jmxremo
te.ssl=false -Djava.net.preferIPv4Stack=true -Djava.rmi.server.hostname=192.168.126.128 -Djava.protocol.handler.pkgs=org
.apache.catalina.webresources -Dorg.apache.catalina.security.SecurityListener.UMASK=0027 -Dignore.endorsed.dirs= -Dcatal
ina.base=/usr/local/tomcat/apache-tomcat-8.5.37 -Dcatalina.home=/usr/local/tomcat/apache-tomcat-8.5.37 -Djava.io.tmpdir=
/usr/local/tomcat/apache-tomcat-8.5.37/temp
```

jstat

(1) 查看虚拟机性能统计信息

The `jstat` command displays performance statistics for an instrumented Java HotSpot VM. The target JVM is identified by its virtual machine identifier, or `vmid` option.

(2) 查看类装载信息

`jstat -class PID 1000 10` 查看某个java进程的类装载信息，每1000毫秒输出一次，共输出10次

[illegible]

(3) 查看垃圾收集信息

```
jstat -gc PID 1000 10
```

```
[root@localhost bin]# jstat -gc 2597 1000 5
S0C   S1C   S0U   S1U   EC     EU      OC     OU      MC     MU      CCSC   CCSU   YGC     YGCT   FGC     FGCT     GCT
896.0 896.0 84.9  0.0   7488.0 794.3   18508.0 13830.4 18688.0 17984.0 2304.0 2039.3   14     0.118   1       0.023   0.141
896.0 896.0 84.9  0.0   7488.0 794.3   18508.0 13830.4 18688.0 17984.0 2304.0 2039.3   14     0.118   1       0.023   0.141
896.0 896.0 84.9  0.0   7488.0 794.3   18508.0 13830.4 18688.0 17984.0 2304.0 2039.3   14     0.118   1       0.023   0.141
896.0 896.0 84.9  0.0   7488.0 794.3   18508.0 13830.4 18688.0 17984.0 2304.0 2039.3   14     0.118   1       0.023   0.141
896.0 896.0 84.9  0.0   7488.0 882.8   18508.0 13830.4 18688.0 17984.0 2304.0 2039.3   14     0.118   1       0.023   0.141
```

jstack

(1) 查看线程堆栈信息

The jstack command prints Java stack traces of Java threads for a specified Java process, core file, or remote debug server.

(2) 用法

```
jstack PID
```

```
[root@localhost bin]# jstack 2597
2019-06-09 03:18:20
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.191-b12 mixed mode):

"Attach Listener" #47 daemon prio=9 os_prio=0 tid=0x00007ff81c002000 nid=0xacf waiting on condition [0x0000000000000000]
 java.lang.Thread.State: RUNNABLE

"ajp-nio-8009-AsyncTimeout" #45 daemon prio=5 os_prio=0 tid=0x00007ff844532800 nid=0xa5c waiting on condition [0x00007ff8134f3000]
 java.lang.Thread.State: TIMED_WAITING (sleeping)
   at java.lang.Thread.sleep(Native Method)
   at org.apache.coyote.AbstractProtocol$AsyncTimeout.run(AbstractProtocol.java:1149)
   at java.lang.Thread.run(Thread.java:748)

"ajp-nio-8009-Acceptor-0" #44 daemon prio=5 os_prio=0 tid=0x00007ff844530800 nid=0xa5b runnable [0x00007ff8135f4000]
 java.lang.Thread.State: RUNNABLE
   at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method)
   at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:422)
   at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:250)
   - locked <0x00000000f61143f0> (a java.lang.Object)
   at org.apache.tomcat.util.net.NioEndpoint$Acceptor.run(NioEndpoint.java:482)
   at java.lang.Thread.run(Thread.java:748)

"ajp-nio-8009-ClientPoller-0" #43 daemon prio=5 os_prio=0 tid=0x00007ff84452e800 nid=0xa5a runnable [0x00007ff8136f5000]
 java.lang.Thread.State: RUNNABLE
   at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
   at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
   at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:93)
   at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
   - locked <0x00000000f1230e40> (a sun.nio.ch.Util$3)
   - locked <0x00000000f1230e30> (a java.util.Collections$UnmodifiableSet)
   - locked <0x00000000f1230e50> (a sun.nio.ch.EPollSelectorImpl)
   at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
   at org.apache.tomcat.util.net.NioEndpoint$Poller.run(NioEndpoint.java:825)
   at java.lang.Thread.run(Thread.java:748)
```

(4) 排查死锁案例

- DeadLockDemo

```
//运行主类
public class DeadLockDemo
{
    public static void main(String[] args)
    {
        DeadLock d1=new DeadLock(true);
        DeadLock d2=new DeadLock(false);
        Thread t1=new Thread(d1);
        Thread t2=new Thread(d2);
        t1.start();
        t2.start();
    }
}
```

```

//定义锁对象
class MyLock{
    public static Object obj1=new Object();
    public static Object obj2=new Object();
}
//死锁代码
class DeadLock implements Runnable{
    private boolean flag;
    DeadLock(boolean flag){
        this.flag=flag;
    }
    public void run() {
        if(flag) {
            while(true) {
                synchronized(MyLock.obj1) {
                    System.out.println(Thread.currentThread().getName()+"----if
获得obj1锁");








                    synchronized(MyLock.obj2) {
                        System.out.println(Thread.currentThread().getName()+"---
-if获得obj2锁");
                    }
                }
            }
        }
        else {
            while(true){
                synchronized(MyLock.obj2) {
                    System.out.println(Thread.currentThread().getName()+"----否则
获得obj2锁");

                    synchronized(MyLock.obj1) {
                        System.out.println(Thread.currentThread().getName()+"---
-否则获得obj1锁");
                    }
                }
            }
        }
    }
}

```

- 运行结果

Run: DeadLockDemo x

```

D:\install\dev\java\jdk1.8\bin\java.exe ...
Thread-0----if获得obj1锁
Thread-1----否则获得obj2锁

```

- jstack分析

```

C:\Users\jack>jps -l
1152 org.jetbrains.jps.cmdline.Launcher
15760 org.jetbrains.kotlin.daemon.KotlinCompileDaemon
5140 com.gupao.gupaojvm.demo.DeadLockDemo
7940 org.jetbrains.idea.maven.server.RemoteMavenServer
8404 org.netbeans/Main
15192
15880 sun.tools.jps.Jps
7960 sun.tools.jconsole.JConsole

C:\Users\jack>jstack 5140
2019-06-10 17:17:41
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.191-b12 mixed mode):

"DestroyJavaVM" #13 prio=5 os_prio=0 tid=0x000000003634000 nid=0x37c4 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"Thread-1" #12 prio=5 os_prio=0 tid=0x000000001e45b800 nid=0x1060 waiting for monitor entry [0x000000001eefe000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.gupao.gupaojvm.demo.DeadLock.run(DeadLockDemo.java:45)
    - waiting to lock <0x0000000076b3eb910> (a java.lang.Object)
    - locked <0x0000000076b3eb920> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:748)

```

把打印信息拉到最后可以发现

```

Found one Java-level deadlock:
=====
"Thread-1":
  waiting to lock monitor 0x000000001cd9d9c8 (object 0x0000000076b3eb910, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x000000001cda0468 (object 0x0000000076b3eb920, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
=====
"Thread-1":
    at com.gupao.gupaojvm.demo.DeadLock.run(DeadLockDemo.java:45)
    - waiting to lock <0x0000000076b3eb910> (a java.lang.Object)
    - locked <0x0000000076b3eb920> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:748)
"Thread-0":
    at com.gupao.gupaojvm.demo.DeadLock.run(DeadLockDemo.java:35)
    - waiting to lock <0x0000000076b3eb920> (a java.lang.Object)
    - locked <0x0000000076b3eb910> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.

```

jmap

(1) 生成堆转储快照

The jmap command prints shared object memory maps or heap memory details of a specified process, core file, or remote debug server.

(2) 打印出堆内存相关信息

```
jmap -heap PID
```

```
jinfo -flag UsePSAdaptiveSurvivorSizePolicy 35352
-XX:SurvivorRatio=8
```



```
[root@localhost bin]# jmap -heap 2597
Attaching to process ID 2597, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Mark Sweep Compact GC

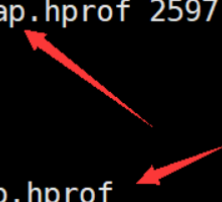
Heap Configuration:
  MinHeapFreeRatio      = 40
  MaxHeapFreeRatio      = 70
  MaxHeapSize           = 257949696 (246.0MB)
  NewSize               = 5570560 (5.3125MB)
  MaxNewSize            = 85983232 (82.0MB)
  OldSize               = 11206656 (10.6875MB)
  NewRatio              = 2
  SurvivorRatio         = 8
  MetaspaceSize         = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize      = 17592186044415 MB
  G1HeapRegionSize      = 0 (0.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
```

(3) dump出堆内存相关信息

```
jmap -dump:format=b,file=heap.hprof PID
```

```
[root@localhost tmp]# jmap -dump:format=b,file=heap.hprof 2597
Dumping heap to /tmp/heap.hprof ...
File exists
[root@localhost tmp]# ll
total 26300
-rw-----. 1 root root 26930127 Jun  9 03:26 heap.hprof
```



(4) 要是在发生堆内存溢出的时候，能自动dump出该文件就好了

一般在开发中，JVM参数可以加上下面两句，这样内存溢出时，会自动dump出该文件

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof
```

设置堆内存大小：-Xms20M -Xmx20M
启动，然后访问localhost:9090/heap，使得堆内存溢出

你会估算GC频率吗？

正常情况我们应该根据我们的系统来进行一个内存的估算，这个我们可以在测试环境进行测试，最开始可以将内存设置的大一些，比如4G这样，当然这也可以根据业务系统估算来的。

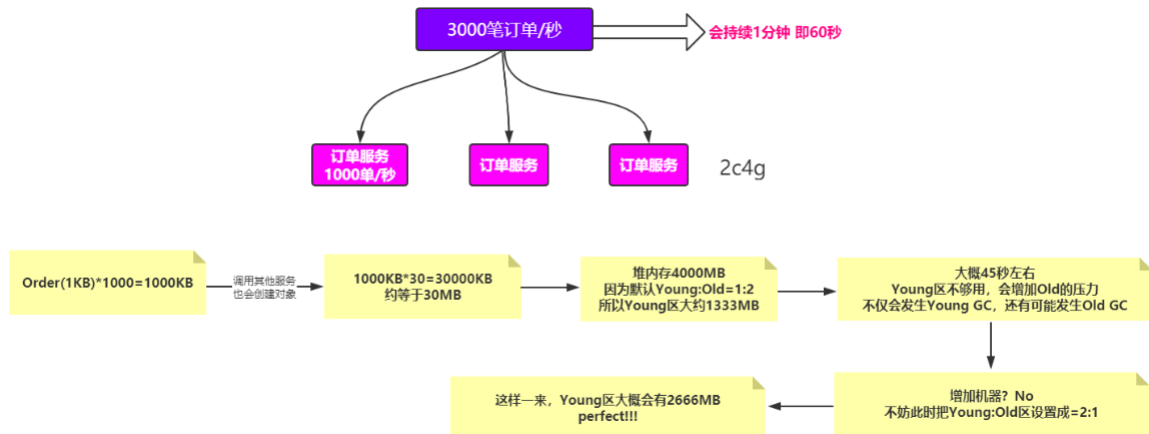
比如从数据库获取一条数据占用128个字节，需要获取1000条数据，那么一次读取到内存的大小就是 $(128 \text{ B}/1024 \text{ Kb}/1024\text{M}) * 1000 = 0.122\text{M}$ ，那么我们程序可能需要并发读取，比如每秒读取100次，那么内存占用就是 $0.122 * 100 = 12.2\text{M}$ ，如果堆内存设置1个G，那么年轻代大小大约就是333M，那么 $333\text{M} * 80\% / 12.2\text{M} = 21.84\text{s}$ ，也就是说我们的程序几乎每分钟进行两到三次youngGC。这样可以让我们对系统有一个大致的估算。

内存优化

4.1.1 内存分配

正常情况下不需要设置，那如果是促销或者秒杀的场景呢？

每台机器配置2c4g，以每秒3000笔订单为例，整个过程持续60秒



4.1.2 内存溢出(OOM)

一般会有两个原因：

- (1) 大并发情况下
- (2) 内存泄露导致内存溢出

4.1.2.1 大并发[秒杀]

浏览器缓存、本地缓存、验证码

CDN静态资源服务器

集群+负载均衡

动静态资源分离、限流[基于令牌桶、漏桶算法]

应用级别缓存、接口防刷限流、队列、Tomcat性能优化

异步消息中间件

Redis热点数据对象缓存

分布式锁、数据库锁

5分钟之内没有支付，取消订单、恢复库存等

4.1.2.2 内存泄露导致内存溢出

ThreadLocal引起的内存泄露，最终导致内存溢出

```
public class TLController {
    @RequestMapping(value = "/t1")
    public String t1(HttpServletRequest request) {
        ThreadLocal<Byte[]> t1 = new ThreadLocal<Byte[]>();
        // 1MB
        t1.set(new Byte[1024*1024]);
        return "ok";
    }
}
```

(1) 上传到阿里云服务器

jvm-case-0.0.1-SNAPSHOT.jar

(2) 启动

```
java -jar -Xms1000M -Xmx1000M -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=jvm.hprof jvm-case-0.0.1-SNAPSHOT.jar
```

(3) 使用jmeter模拟10000次并发

39.100.39.63:8080/t1

(4) top命令查看

```
top
top -Hp PID
```

(5) jstack查看线程情况，发现没有死锁或者IO阻塞的情况

```
jstack PID
java -jar arthas.jar ---> thread
```

(6) 查看堆内存的使用，发现堆内存的使用率已经高达88.95%

```
jmap -heap PID
java -jar arthas.jar ---> dashboard
```

(7) 此时可以大体判断出来，发生了内存泄露从而导致的内存溢出，那怎么排查呢？

```
jmap -histo:live PID | more
获取到jvm.hprof文件，上传到指定的工具分析，比如heaphero.io
```

GC的垃圾回收条件模式图：

<https://www.processon.com/view/link/62bc50e47d9c08073522779c>