JVM原理&线上排查问题经验交流

Java 内存管理回收机制、虚拟机配置、常见泄露和GC 问题。

本次交流分为以下四个部分

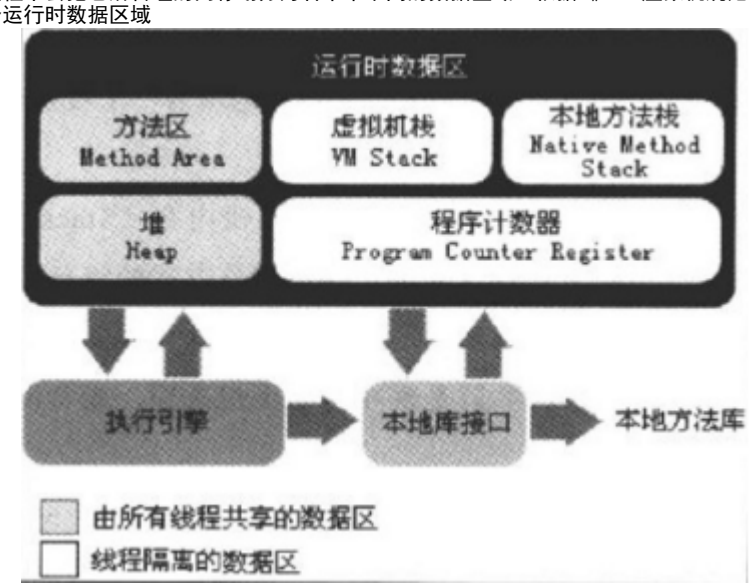
1、虚拟机内存划分
2、内存分配与回收机制&虚拟机配置

> *2.1）堆分为几个部分*
> *2.2）每部分的作用*
> *2.3）什么情况下进行垃圾回收*
> *2.4）使用什么方法进行垃圾回收*
> *2.5）分析线上scf虚拟机参数设置*

3、使用什么工具分析JVM
4、线上问题解决方案分享

虚拟机内存的划分

如果不了解虚拟机内存是如何划分的则很难排查线上的因虚拟机导致的问题，更不用提虚拟机调优了，所以我们先来说虚拟机内存是如何划分的
运行时数据区
java虚拟机在执行java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。根据《Java虚拟机规范（Java SE 7版）》的规定，Java虚拟机所管理的内存包括以下几个运行时数据区域



程序计数器：线程所执行字节码的行号指示器，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、调整、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成，每条线程都有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，这个区域为“线程私有”内存
java虚拟机栈：栈也是线程私有的，它的生命周期与线程相同，每个方法在执行的同时都会创建一个栈帧用于存储方法的一些信息，栈帧的大小在**编译期**就已经决定了，每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机中入栈到出栈的过程
本地方法栈：本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行java服务，而本地方法栈则为虚拟机使用到的Native方法服务
native关键字说明其修饰的方法

```
/* register the natives via the static initializer.
 *
 * VM will invoke the initializeSystemClass method to complete
 * the initialization for this class separated from clinit.
 * Note that to use properties set by the VM, see the constraints
 * described in the initializeSystemClass method.
 */
private static native void registerNatives();
static {
    registerNatives();
}
```

问：到这为止。有人知道为什么在tomcat中System.out和System.err会把日志输出到catalina.out中吗？

上面注释翻译过来的意思是：通过静态初始化注册native方法，该方法会令jvm通过调用initializeSystemClass方法来完成初始化工作

```
1.  private static void initializeSystemClass() {
2.   // props
3.  props = new Properties();
4.  initProperties(props);
5.  sun.misc.VM.saveAndRemoveProperties(props);
6.   //
7.  lineSeparator = props.getProperty("line.separator");
8.  sun.misc.Version.init();
9.   //inouterrsetXX0()setXX0()
       1.  //native准流管理到类内的对象
10.  FileInputStream fdIn = new FileInputStream(FileDescriptor.in);
11.  FileOutputStream fdOut = new FileOutputStream(FileDescriptor.out);
12.  FileOutputStream fdErr = new FileOutputStream(FileDescriptor.err);
13. setIn0(new BufferedInputStream(fdIn));
14. setOut0(new PrintStream(new BufferedOutputStream(fdOut, 128), true));
15. setErr0(new PrintStream(new BufferedOutputStream(fdErr, 128), true));
16.  //zipjava.util.zip.ZipFile
17. loadLibrary("zip");
18.  //
19.  Terminator.setup();
20.  // sun.misc
21. sun.misc.VM.initializeOSEnvironment();
22.  //
       1.  //......
23.  Thread current = Thread.currentThread();
24. current.getThreadGroup().add(current);
25.  // JavaLangAccess
26. setJavaLangAccess();
27.  // sun.misc.VM.isBooted()application
       1.  //booted(),isBooted()true
28. sun.misc.VM.booted();
29. }
```

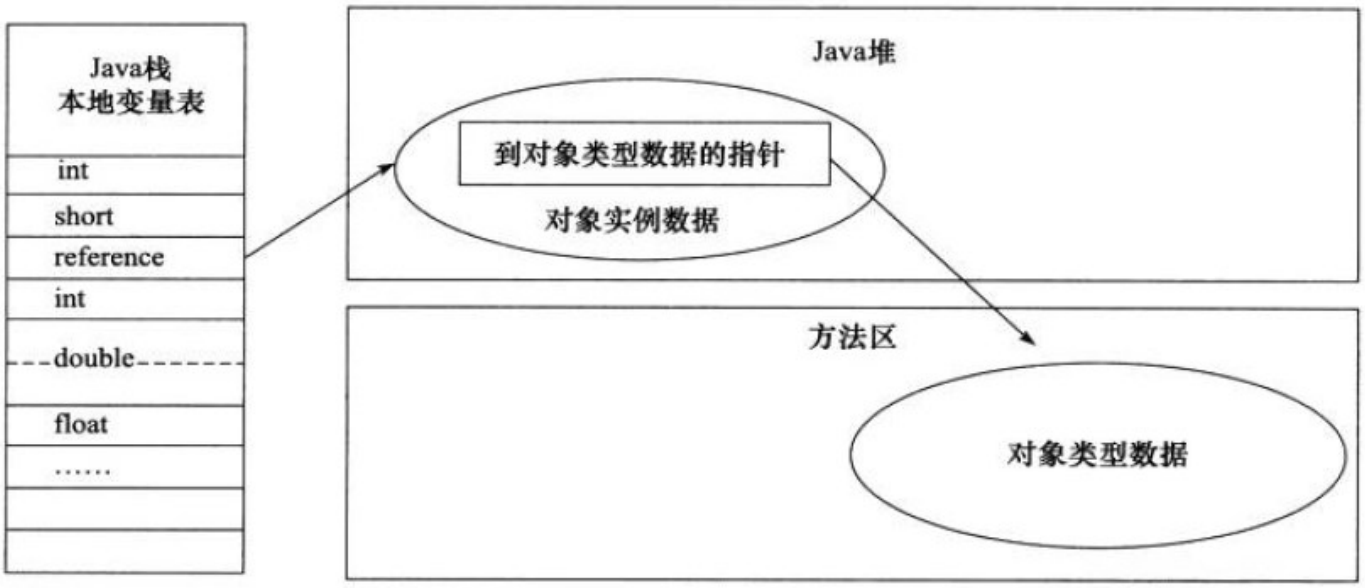
```
eval $_NOHUP "\"$_RUNJAVA\"" "\"$LOGGING_CONFIG\"" $LOGGING_MANAGER $JAVA_OPTS $CATALINA_OPTS \
  -Djava.endorsed.dirs="\"$JAVA_ENDORSED_DIRS\"" -classpath "\"$CLASSPATH\"" \
  -Dcatalina.base="\"$CATALINA_BASE\"" \
  -Dcatalina.home="\"$CATALINA_HOME\"" \
  -Djava.io.tmpdir="\"$CATALINA_TMPDIR\"" \
  org.apache.catalina.startup.Bootstrap "$@" start \
  >> "$CATALINA_OUT" 2>&1 "&"
```

java堆： java堆是java虚拟机所管理内存中最大的一块。java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有对象实例都在这里分配内存（有一些优化技术，例如栈上分配、标量替换等等，导致所有对象都分配在堆上渐渐变得不是那么绝对了），**java堆是垃圾收集管理器管理的主要区域，还会对那部分进行垃圾收集？**

问：垃圾收集器还能管理哪些区域？能管理java虚拟机栈吗？

方法区：方法区与java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。永久代！=方法区，仅仅是因为HotSpot虚拟机的设计团队选择把GC分代收集扩展至方法区，或者说使用永久代来实现方法区而已，对于其他虚拟机来说是不存在永久代的概念的
运行时常量池（属于方法区）：Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用。这部分内容在类加载后进入方法区的运行时常量池中存放

对象的内存布局：对象在内存中存储的布局可以分为3块区域：对象头、实例数据和对齐填充；
对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID等等（这部分官方称它为Mark Word）对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定java对象的大小，但是从数据的元数据中却无法确定数组的大小



HotSpot中的内存布局

以下面代码为例描述虚拟机从生到死的过程

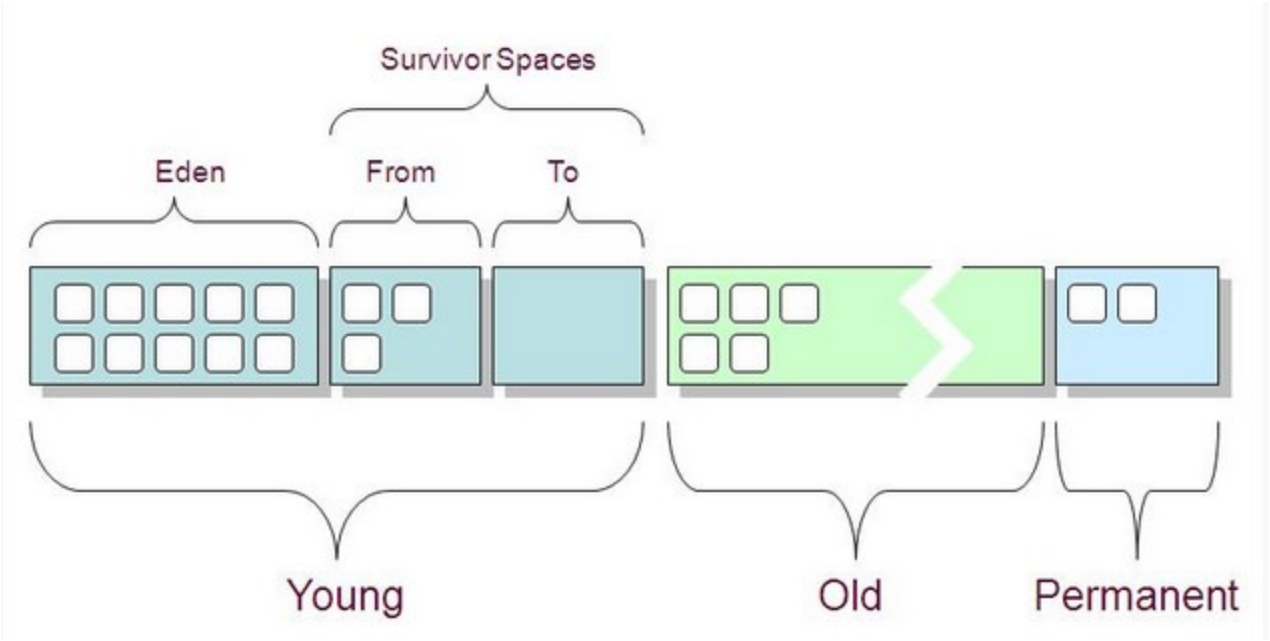
```
1.  public class Hello {
2.   public static void main() {
3.   Hello hello = new Hello();
4.  hello.speak();
5.    }
6.   public void speak(){
7.   System.out.println("Hello World");
8.    }
9.  }
```

java虚拟机根据虚拟机启动参数启动虚拟机-->加载用到的Hello.class文件将文件中的信息以一定的格式组织起来存放到方法区-->创建java虚拟机栈-->根据main方法信息创建栈帧并将栈帧压入栈顶-->创建Hello对象在堆中为其分配内存，栈帧中的hello引用指向该对象-->创建speak方法对应的栈帧-->创建本地方法栈初始化System类并输出Hello World-->speak栈帧弹出-->main栈帧弹出-->main函数所在虚拟机栈销毁，主线程退出-->java虚拟机退出

==========第一部分结束==========

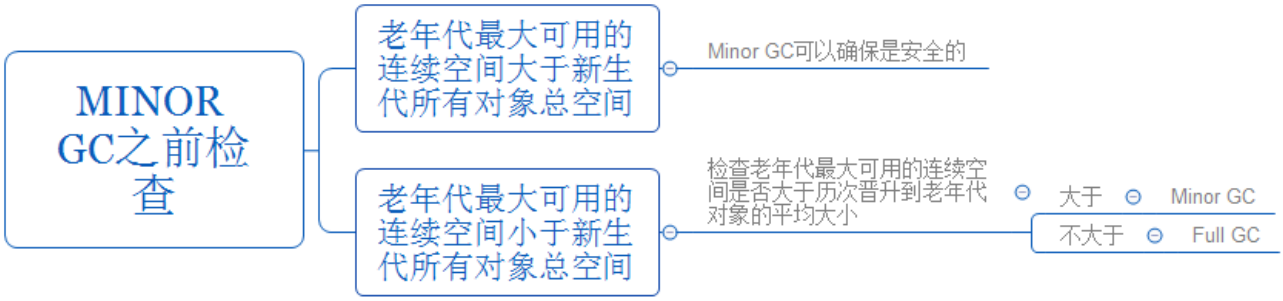
内存回收机制

对象的内存分配，往大方向上将就是在堆上分配，分配的规则并不是百分之百固定的，和当前使用哪一种垃圾收集器组合还有虚拟机中与内存相关的参数设置相关
下面说几条最普遍的内存分配规则，先来看下HotSpot虚拟机中Java堆中各代分布

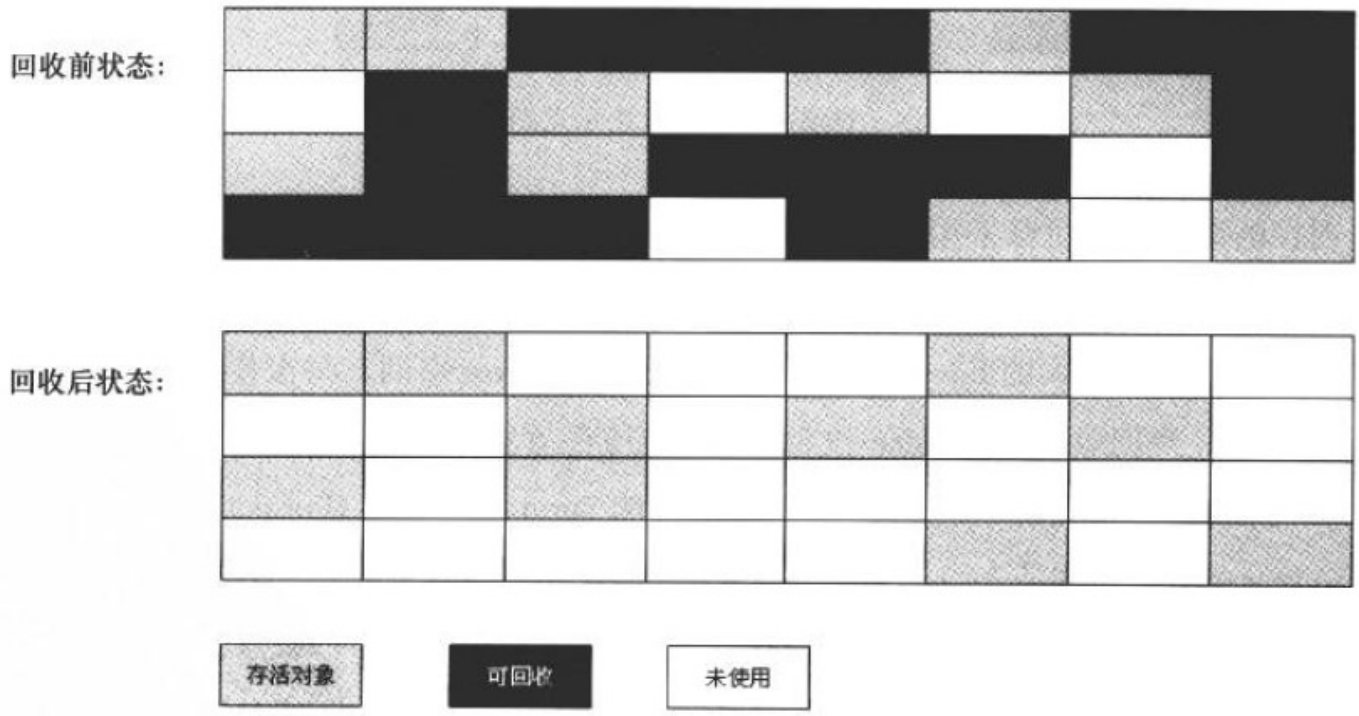


1、大多数情况下，对象在新生代Eden区分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次MinorGC
2、大对象直接进入老年代，所谓大对象是指，需要大量连续内存空间的java对象，虚拟机提供了一个-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在Old老年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制
3、长期存活的对象将进入老年代，虚拟机给每个对象定义了一个对象年龄计数器，如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1，对象在Survivor区中每熬过一次MinorGC，年龄就增加1岁，默认增加到15岁时改对象就会晋升到老年代，晋升到老年代的年龄阈值，可以通过-XX:MaxTenuringThreshold设置
虚拟机并不是永远要求对象年龄必须达到设置的值才能晋升到老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代
问：如果发生Minor GC时，Survivor区的空间容纳不下Eden区目前存活的对象怎么办？如果老年代也容纳不下怎么办？

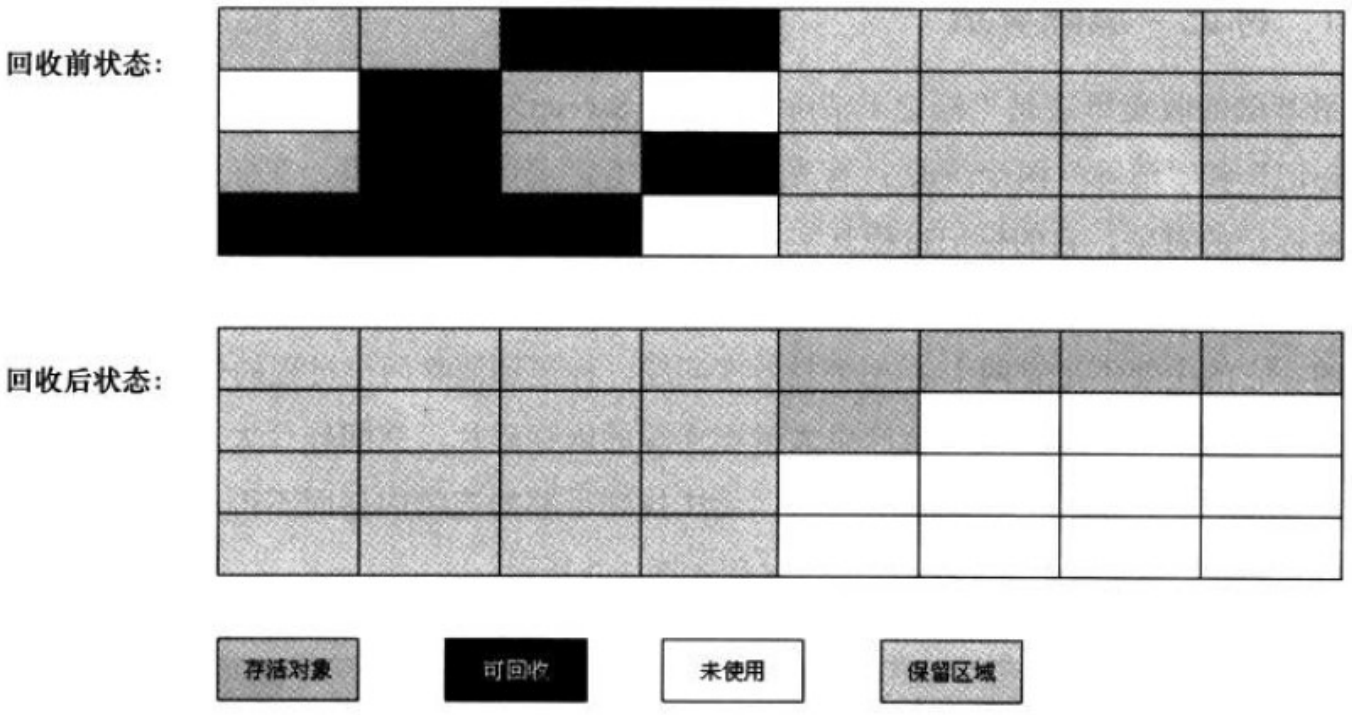
空间分配担保策略



java使用可达性分析算法而不是引用技术算法来判断对象是否存活
可达性分析算法基本思路是通过一系列的成为GC Roots的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的
可作为GC Roots的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧的本地变量表）中
类回收的条件：
1、该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例
2、该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法
3、加载该类的ClassLoader已经被回收
几种垃圾收集算法：
标记-清除算法：
缺点：效率问题，标记和清除两个过程效率都不高，标记清除之后产生大量不连续的内存碎片，空间碎片太多会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作

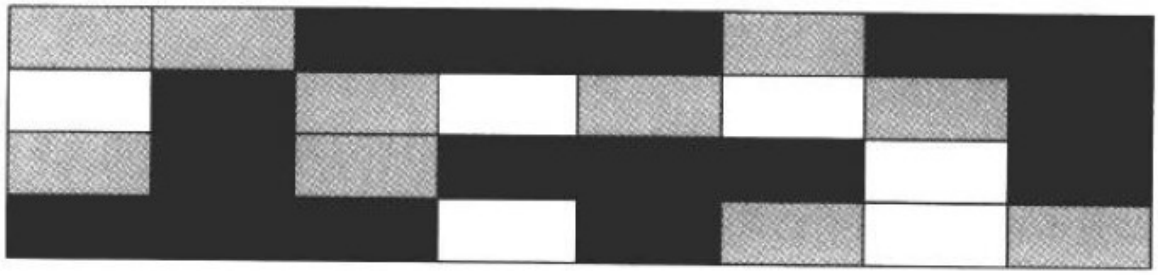


复制算法：
将可用内存按容量划分为大小相等的两块，每次只使用一块。当一块用完之后就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉
优点：内存分配不用考虑内存碎片，只要移动堆顶指针，按顺序分配内存，简单，高效
缺点：内存浪费严重
现代商业虚拟机都采用这种办法来回收新生代，只是改变了分配比例，HotSpot默认8:1:1

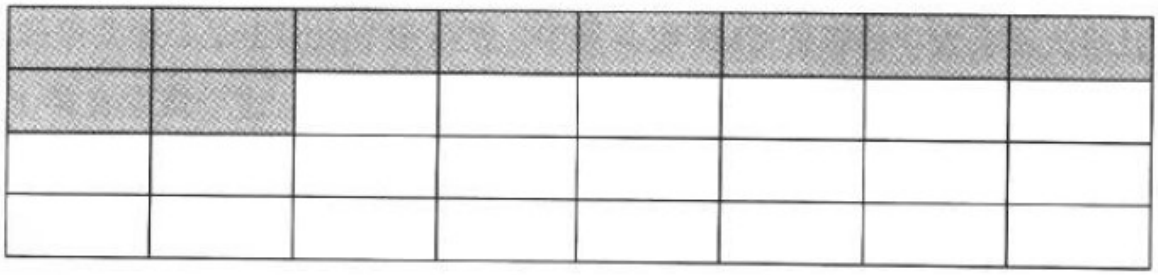


标记-整理算法：
标记清除算法改进版，一般老年代使用该算法

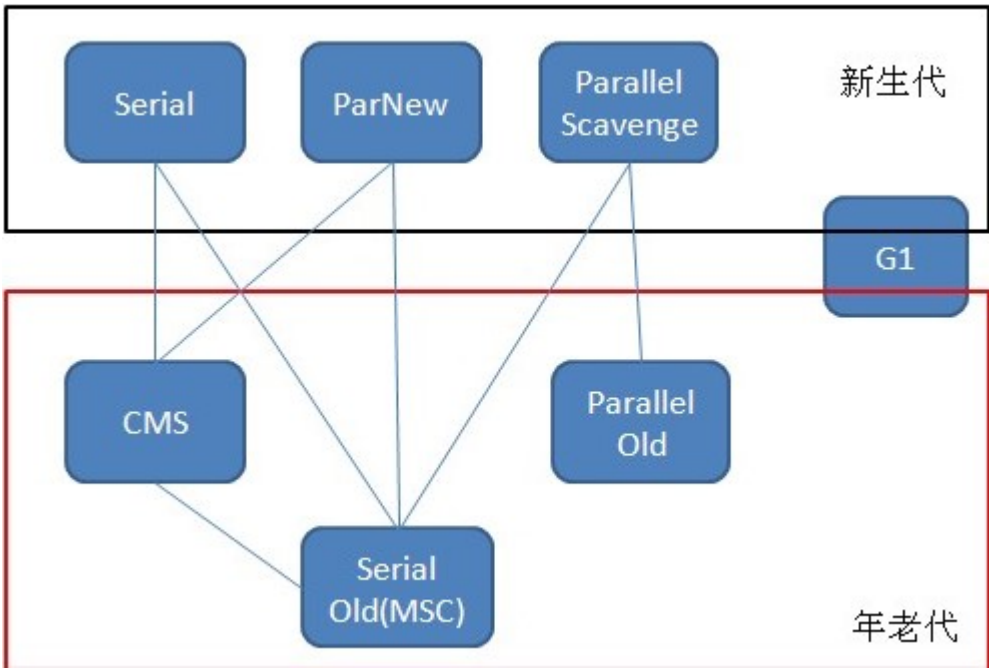
回收前状态：



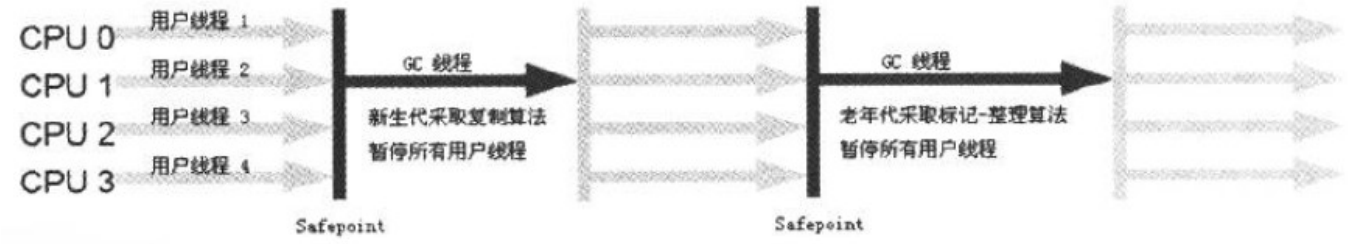
回收后状态：



几种垃圾收集器：



1、Serial收集器
单线程收集器，工作时，必须暂停其他所有的工作线程知道收集结束



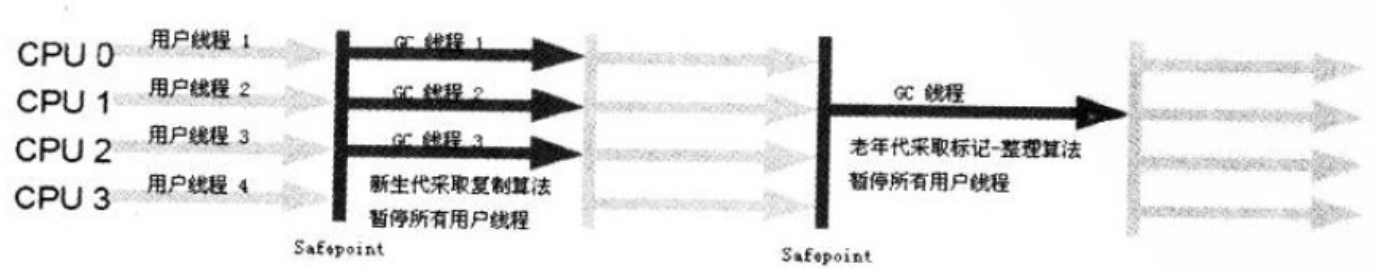
新生代Serial与年老代Serial Old搭配垃圾收集过程图

2、ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、回收策略等都和Serial收集器完全一样

3、Paraller Scavenge收集器

Paraller Scavenge收集器看上去和ParNew一样，其实关注点并不一样，ParNew更关注缩短垃圾收集时用户线程停顿时间，而Paraller Scavenge更关注吞吐量（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））提高响应速度



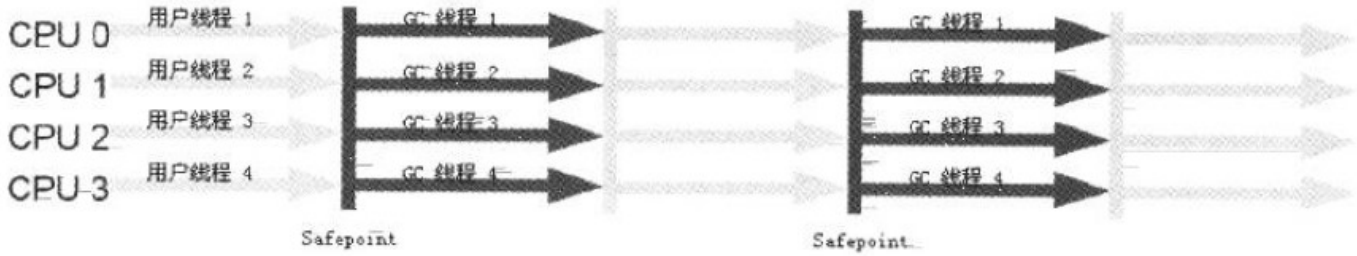
新生代Paraller Scavenge/ParNew与年老代Serial Old搭配垃圾收集过程图

4、Serial Old收集器

Serial Old是Serial收集器的老年代版本，同样是一个单线程收集器，使用标记-整理算法

5、Paraller Old收集器

Paraller Old是Paraller Scavenge收集器的老年代版本，使用多线程和标记-整理算法，JDK1.6开始提供，在注重吞吐量以及CPU资源敏感的场合，都可以有限考虑Paraller Old+Paraller Scavenge组合

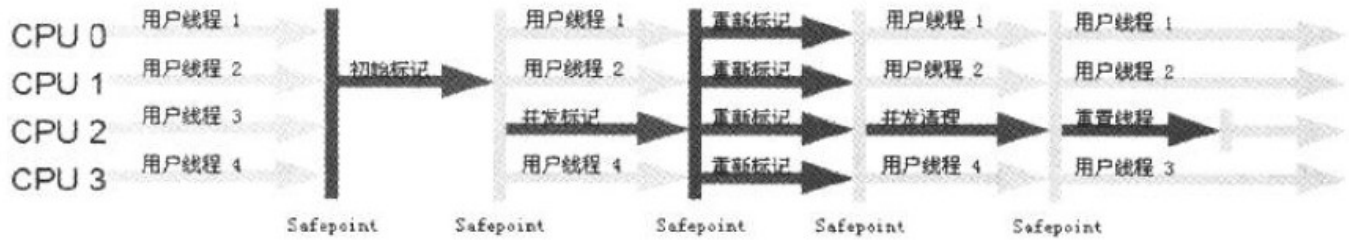


新生代Paraller Scavenge/ParNew与年老代Paraller Old搭配垃圾收集过程图

6、CMS收集器

CMS收集器是基于标记-清除算法实现的，整个过程分为4个步骤：

1）初始标记
2）并发标记
3）重新标记
4）并发清除



CMS收集器是一种以获取最短回收时间为目标的收集器。非常适合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的应用上述四个步骤中的初始标记和重新标记两个步骤仍然需要Stop The World。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段

则是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一般会比初始标记阶段稍长一些，但远比并发标记的时间短，由于整个过程中耗时最长的并发
标记和并发清除过程收集器线程都可以和用户线程一起工作，所以，从总体上说，CMS的内存回收过程是与用户线程一起并发执行的
缺点一：在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分CPU资源而导致应用程序变慢，总吞吐量会降低，CMS默认启动的回收线程数是（CPU数量+3）/4，**当CPU不足4个时**，CMS对用户程序
的影响会变得很大。
缺点二：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（Concurrent Mode Failure完全违背了CMS的初衷），由于CMS并发清理阶段用户线程还在运行着，伴随程序
运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉，这一部分垃圾就称为浮动垃圾。由于在垃圾收集阶段用户线程还需运行，
那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等待老年代几乎完全被填满了再进行收集，1.5默认当老年代使用了68%的空间后就被激活，1.6为92%，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**（CMS收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数，默认开启，用于在CMS收集器顶不住要进行Full GC时开启内存碎片的合并整理，内存整理的过程是无法并发的），临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿的时间就很长了，所以说参数-XX:
CMSInitiatingOccupancyFraction设置的太高很容易导致大量Concurrent Mode Failure失败，性能反而降低
7、G1收集器
目前为止最牛逼的收集器，但是因为比较新用的人还比较少，直到JDK7u4之后，Sun公司认为它达到了足够成熟的商用程度（HotSpot团队的目标是未来替换掉CMS收集器）
G1收集器与其他收集器相比的特点：
并行与并发：充分利用多CPU、多核环境下的硬件优势，使用多个CPU来说短Stop The World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。
分代收集：G1收集器不需要与其他收集器配合，能够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的对象以获取更好的收集效果
空间整合：运行期间不会产生内存空间碎片
可预测的停顿：G1除了追求低停顿，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
使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个堆划分为多个大小相等的独立区域，虽然还保留有新生代和老年代的概念，但新生代和老年代已经不再是物理隔离的了，它们都是一部分Region的集合
G1收集器几乎可以说还没有经过实际应用的考验，网络上关于G1收集器的性能测试也非常贫乏。如果你现在采用的收集器没有出现问题，那就没有任何理由现在去选择G1

看下scf容器的虚拟机参数

```
JAVA_OPTS="-Xms$VM_XMS -Xmx$VM_XMX -Xmn$VM_XMN -Xss1024K -XX:PermSize=256m -XX:MaxPermSize=512m -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:SurvivorRatio=65536 -XX:MaxTenuringThreshold=0 -XX:CMSInitiatingOccupancyFraction=80 -DSCF.uspcluster=$SERVICE_NAME -Dasyn.log.switch=$ASYN_LOG -Dasyn.log.queue.size=$ASYN_LOG_QUEUE_SIZE"
```

ServiceName	VM_XMS	VM_XMX	VM_XMN	ServiceID	Status
zzappmanager	2g	2g	1g	26816	RUNNING
zzfriendverify	1g	1g	512m	7235	RUNNING
zzinfologic	6g	6g	3g	7040	RUNNING
zzinfoshow	6g	6g	3g	3563	RUNNING
zzlogic	6g	6g	3g	17961	RUNNING
zzusercenter	6g	6g	3g	10230	RUNNING

```
VM_XMS=2G VM_XMX=2G VM_XMN=1G
上面3个参数是默认的，可配置
JAVA_OPTS="-Xms$VM_XMS -Xmx$VM_XMX -Xmn$VM_XMN -Xss1024K
-XX:PermSize=256m -XX:MaxPermSize=512m
-XX:ParallelGCThreads=20
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+UseCMSCompactAtFullCollection
-XX:SurvivorRatio=65536
-XX:MaxTenuringThreshold=0
-XX:CMSInitiatingOccupancyFraction=80
-DSCF.uspcluster=$SERVICE_NAME
-Dasyn.log.switch=$ASYN_LOG
-Dasyn.log.queue.size=$ASYN_LOG_QUEUE_SIZE"
```

```
-Xms-Xmx
-Xmn=1G1G
-Xss1024K1024K
-XX:PermSize=256m256m
-XX:MaxPermSize=512m512m
-XX:ParallelGCThreads=20GC20
-XX:+UseConcMarkSweepGC-XX:+UseParNewGCParNew
CMS
-XX:UseCMSCompactAtFullCollection)CMS
FullGC
-XX:SurvivorRatio=65535Survivor8:1
-XX:MaxTenuringThreshold=0
-XX:CMSInitiatingOccupancyFraction=80CMS
80%
2G(1G1GEden,
YGCS0S1,1G)256m
```

11个线程分配1024k的内存，年轻代使用ParNew进行并行垃圾收集策略，YGC的时候使用20条线程，年老代使用CMS进行4部收集策略，老年代达到80%要进行FullGC，进行FullGC时整理下内存

（1）-XX:+HeapDumpOnOutOfMemoryError参数表示当JVM发生OOM时，自动生成DUMP文件。

（2）-XX:HeapDumpPath=${目录}参数表示生成DUMP文件的路径，也可以指定文件名称，例如：-XX:HeapDumpPath=${目录}/java_heapdump.hprof。如果不指定文件名，默认为：java_<pid>_<date>_<time>_heapDump.hprof。

==========第二部分结束==========

排查问题工具

1、jstat工具：用于收集HotSpot虚拟机各方面的运行数据

关注百分比，关注动态

```
[work(konglingqiu@58OS)@bjdhj-183-79 ~]$ jstat -gcutil 12022 1000 5
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  37.59  63.74  18.96     42    2.488     2    0.018    2.506
  0.00   0.00  37.61  63.74  18.96     42    2.488     2    0.018    2.506
  0.00   0.00  37.93  63.74  18.96     42    2.488     2    0.018    2.506
  0.00   0.00  37.93  63.74  18.96     42    2.488     2    0.018    2.506
  0.00   0.00  37.96  63.74  18.96     42    2.488     2    0.018    2.506
```

关注容量，没有jmap直观

```
[work(konglingqiu@58OS)@bjdhj-183-79 ~]$ jstat -gc 12022 1000 5
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT
 64.0   64.0   0.0    0.0   262016.0 65167.2   262144.0   170783.4  262144.0 49710.9     43    2.559   2      0.018    2.577
 64.0   64.0   0.0    0.0   262016.0 65240.0   262144.0   170783.4  262144.0 49710.9     43    2.559   2      0.018    2.577
 64.0   64.0   0.0    0.0   262016.0 65344.9   262144.0   170783.4  262144.0 49710.9     43    2.559   2      0.018    2.577
 64.0   64.0   0.0    0.0   262016.0 65362.1   262144.0   170783.4  262144.0 49710.9     43    2.559   2      0.018    2.577
 64.0   64.0   0.0    0.0   262016.0 65520.4   262144.0   170783.4  262144.0 49710.9     43    2.559   2      0.018    2.577
```

2、jmap工具：功能和jstat类似生成内存快照，关注静态

```
[work(konglingqiu@58OS)@bjdhj-183-79 ~]$ jmap -heap 11111
using parallel threads in the new generation. #
using thread-local object allocation.
Concurrent Mark-Sweep GC ##
Heap Configuration: #
 MinHeapFreeRatio = 40 #
 MaxHeapFreeRatio = 70 #
 MaxHeapSize = 3221225472 (3072.0MB) #
 NewSize = 2147483648 (2048.0MB) #
 MaxNewSize = 2147483648 (2048.0MB) #
 OldSize = 5439488 (5.1875MB) #
 NewRatio = 2 #
 SurvivorRatio = 8 #suvivor
 PermSize = 21757952 (20.75MB) #perm
 MaxPermSize = 85983232 (82.0MB) #perm
Heap Usage: #
New Generation (Eden + 1 Survivor Space): #(+survior)
capacity = 1932787712 (1843.25MB) #
used = 127806624 (121.88589477539062MB) #
free = 1804981088 (1721.3641052246094MB) #
 6.612553629480028% used #
Eden Space: #
capacity = 1718091776 (1638.5MB) #
used = 125888720 (120.05683898925781MB) #
free = 1592203056 (1518.4431610107422MB) #
 7.327240707308991% used #
From Space: #survior1
capacity = 214695936 (204.75MB) #survior1
used = 1917904 (1.8290557861328125MB) #survior1
free = 212778032 (202.9209442138672MB) #survior1
 0.8933117392590049% used
To Space: #survior2
capacity = 214695936 (204.75MB) #survior2
used = 0 (0.0MB)
free = 214695936 (204.75MB)
 0.0% used
concurrent mark-sweep generation: #
capacity = 1073741824 (1024.0MB) #
used = 680358880 (648.8407897949219MB) #
free = 393382944 (375.1592102050781MB) #
 63.36335837841034% used
Perm Generation: #perm
```

```
capacity = 23220224 (22.14453125MB) #perm
used = 13999544 (13.351005554199219MB)
free = 9220680 (8.793525695800781MB)
 60.29030555433057% used
##############perm
```

```
[work(konglingqiu@580S)@bjdhj-183-79 ~]$ jmap -histo:live 12022 | more

 num     #instances         #bytes  class name
----------------------------------------------
   1:         31504       49133800  [B
   2:         81097       11064312  <constMethodKlass>
   3:         81097       11038024  <methodKlass>
   4:          7639        7717584  <constantPoolKlass>
   5:        122306        7205504  [C
   6:        106560        5932648  <symbolKlass>
   7:          7639        5752208  <instanceKlassKlass>
   8:        122718        3926976  java.lang.String
   9:          3839        2786808  <constantPoolCacheKlass>
  10:         16093        1931160  java.net.SocksSocketImpl
  11:         38769        1860912  com.google.gson.internal.LinkedTreeMap$Node
  12:         15627        1524000  [Ljava.util.HashMap$Entry;
  13:         45014        1440448  java.util.HashMap$Entry
  14:         19582        1177224  [Ljava.lang.Object;
  15:          2123        1068656  <methodDataKlass>
  16:         42524        1020576  java.util.LinkedList$Entry
  17:          6691        1018128  [I
  18:         23493         939720  java.util.LinkedHashMap$Entry
  19:          7953         827112  java.lang.Class
  20:         50341         805456  java.lang.Object
  21:         16098         772704  com.google.gson.internal.LinkedTreeMap
  22:         19028         761120  java.lang.ref.Finalizer
  23:          8548         752224  java.lang.reflect.Method
  24:         45613         729808  java.lang.Integer
  25:         14990         719520  com.bj58.zhuanzhuan.zzinfologic.entity.ZZDispLocal
  26:         11265         683880  [S
  27:         12727         674072  [[I
  28:         10397         665408  java.nio.DirectByteBuffer
  29:         16541         529312  java.util.concurrent.ConcurrentHashMap$HashEntry
  30:          9280         519680  java.util.LinkedHashMap
  31:         16091         514912  java.net.Socket
```

生成堆转储文件，文件大小基本等于当前使用内存大小，生成的文件使用Eclipse开发的工具Mat拿到线下来分析，不要使用jhat直接在线上分析，线上不到万不得已一定不要用，用上面的简化版替代

```
jmap -dump:format=b,file=tmp.dump 21711
```

3、jstack工具：显示虚拟机的线程快照

```
jstack 21711 > 21711.txt
```

可以打印出线程状态，根据线程名称，线程状态，线程数量，是否发生死锁等信息来定位问题

==========第三部分结束==========

线上问题解决方案分享

出问题基本都发生在CPU和内存上

1、CPU接近满负荷有很大几率是有死循环，比如某种条件下while循环不能退出、在并发环境下使用线程不安全的hashmap

```
topCPU
ps -mp pid -o THREAD,tid,time
printf "%x\n" tid ID16
jstack pid | grep threadId -A 100
```

2、某个进程响应极其缓慢

```
jstat -gcutiljstat -gcGC
FGCTGCTFGCGC
FGC
1scf

2GuavaCache
访问量的增大而增大
3jmap -histo
立刻改变思路（程序员找不到自己写的这么高深的
bug很正常，毕竟上我们内心都认为自己的代码是不可能存在这种漏洞的），应该先生成堆转
MatMat
了
```

3、整个机器的内存很快被吃没

```

线程池的地方，因为线程是分配在栈上的，在scf的
虚拟机参数中配置了-Xss=1024k，也就是说每创建1个线程就耗费掉1M的内存

1()
ps -xH |grep java|awk '{print $1}'|uniq -c|sort -nr|head -5
2、找出了对于的进程后用jstck查看是什么样的线程导致的(在这里建议大家无论是使用线程池
还是线程一定要起名字)
```

最后给大家推荐三本书
1、《Java并发编程的艺术》--方腾飞
2、《大型网站技术架构：核心原理与案例分析》--曾宪杰
3、《深入理解JAVA虚拟机----JVM高级特性与最佳实践》
==========THE END==========
http://blog.csdn.net/blade2001/article/details/9065985 socket问题导致while循环没有退出，导致cpu满载例子