JVM 详解与调优

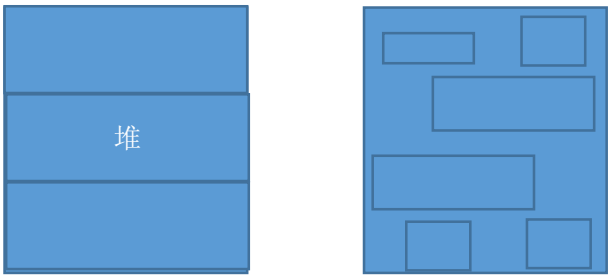
- **一 概念**

1. **数据类型**

基本类型：保存数据本身 **byte short long char float double Boolean**

引用类型：保存引用值 **类类型,接口类型 数组**

2. **堆与栈**



栈是运行时单位 堆是存储的单位

栈决定了程序的运行问题，程序如何执行，怎么处理数据;堆解决存储问题，数据怎么放饭在哪里

在 JAVA 中一个线程就有响应的一个线程栈与之对应,因为不同的程序执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所以线程共享的。栈因为是运行单位,因此里面存储的信息都是跟当前线程(程序)相关信息。包括局部变量.程序运行状态,返回值等等; 而堆只负责存储对象。

程序运行----------→线程切换[保存状态]

堆栈分离好处

1 设计角度,栈代表处理逻辑，堆代表数据。而堆代表了数据,分开逻辑清晰.。分而治之的思想。这种隔离，模块的思想在软件方面方方面面都有体现

2 堆与栈分离，使得堆中的内容可以被多个栈共享。a 高效的数据交互方式(内存共享),另一方面,堆中的共享常量和缓存可以被所以栈访问,节省空间

3 栈因为运行时的需要,保存系统运行的上下文，地址段的划分。由于栈只能向上增长,因此就会限制住栈的存储能力。而堆的对象是可以根据需要动态增长,因此栈和堆的拆分,使得动态增长成为可能,同时栈中只需要存储堆中的一个地址就可以了。

4 面对对象就是栈和堆的完美结合。其实面对对象和其他结构化的程序在执行上没有区别。但是，面对对象的引入，让我们的思考方式更接近自然的思考方式。当我们把对象拆开，对象的属性就是数据，存在栈中;对象的行为就是运行逻辑。

编写面对对象程序的时候，同时也编写了处理逻辑的数据。面对对象的设计确实很美

堆存储？栈存储？

堆中存储的对象。栈中存储的是基本类型，对象的引用。一个对象的大小是不可以估计的，动态变化的，但是栈中一个只对于了一个 4byte 的应用

基本类型不放在堆中的原因?因为它占用的空间一般是 1-8 个字节—需要空间比较少，而且是基本类型，不会出现自增长—长度固定,因此栈中的存储就够了。

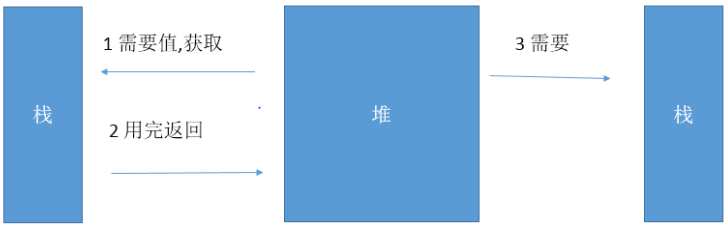
重点 Java 的参数传递是传值不是传引用

1 不能和 C 语言类别,java 没有指针概念

2 程序永远在栈中进行,因此传值的时候，只存在传递基本类型和对象的引用问题。不会直接传递对象本身

所以，Java 里面都是传值调用，简化了 C 中的复杂性。

在运行栈中，基本类型和引用处理是一样的,都是传值,所以,如果是传引用的方法调用,也同时理解”传引用值”的传递调用

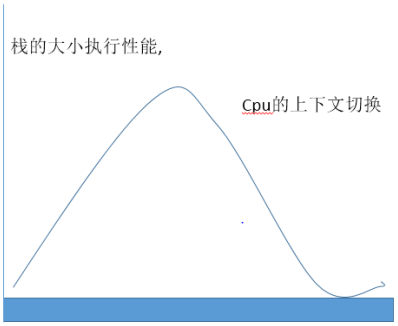


问题: 不控制可能读取脏数据

解决:串行化..

栈和堆中,栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为了栈进行数据存储服务，堆就是一块共享内存。正是因为堆和栈的分离,才使得 java 的垃圾回收成为可能

Java 中，**栈的大小-Xss** 来设置，当栈中的存储数据比较多时候，适当调大这个值,否则出现 java.lang.StackOverflowError 异常。常见的这个异常时无法返回的递归,因为此时栈中保存的信息都是方法



3 引用类型

分为 **强引用[可以自己控制] ,软引用[java 虚拟机控制]，弱引用**

强引用:就是我们一般对象是时虚拟机生成的应用,强引用的环境下,垃圾回收需要严格判断当前对象是否被强引用,如果被强引用化,则不能垃圾回收

软引用:软引用一般当做缓存来做。与强引用的区别，弱应用在垃圾回收时候，虚拟机会根据当前的系统剩余内存来决定是否对软引用进行回收。如果剩余内存紧张，则虚拟机会回收软应用剩余空间;如果剩余空间比较富裕,则不会进行回收。换句话说，虚拟机在发生 OutOfMemory 时候，肯定没有软引用。

强引用不用说,我们系统一般在使用的时候都是强引用,而”软引用“和”弱引用”比较少见。他们一般被当做缓存来使用，而且一般是在内存比较受限制的情况下，如果内存足够大，直接使用强引用设置缓存可以。同时可控性更高。因此，他们常见的是被当做桌面系统应用系统缓存。

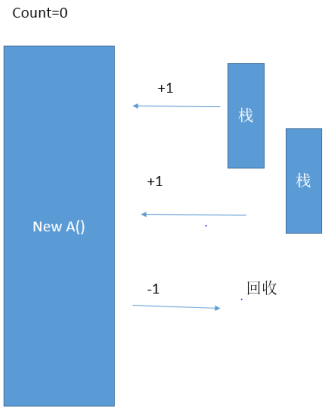
Android2.3 在我的记忆中用的弱引用作为缓存

- ## 二 基本垃圾回收算法

1 不同角度划分垃圾回收算法

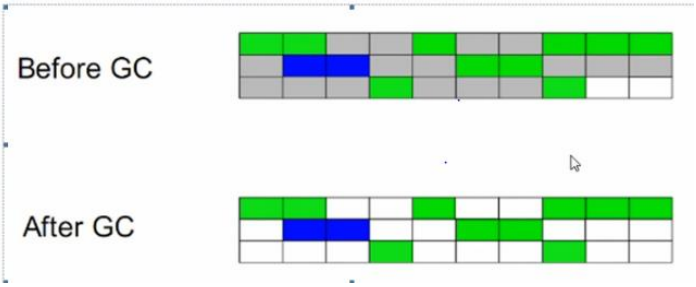
引用计数 (Reference Counting):

比较古老的算法。原理是此对象有一个引用,即增加一个计数,删除一个引用就减少一个计数。垃圾回收时,只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题



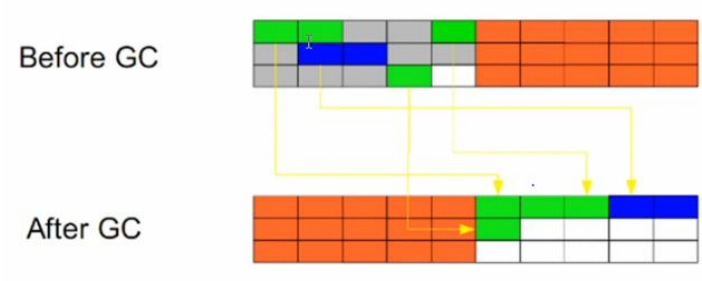
标记-清除 (Mark-Sweep): || 要求的是响应时间,或者吞吐量不同情况

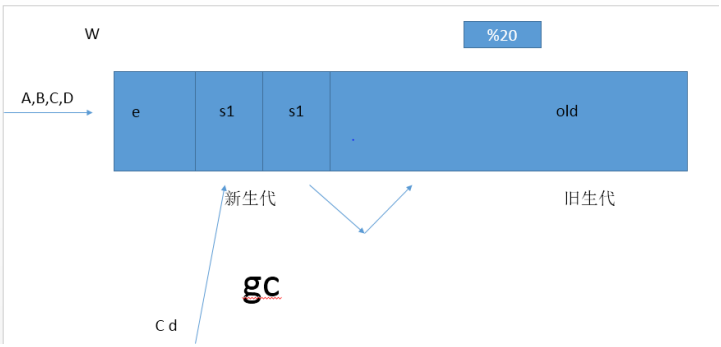
此算法执行分为两个阶段。第一阶段从引用跟节点开始标记所以被引用的对象,第二阶段遍历整个堆，把未标记的对象清除.此算法需要暂停整个应用，产生内存碎片



复制（Copying）||

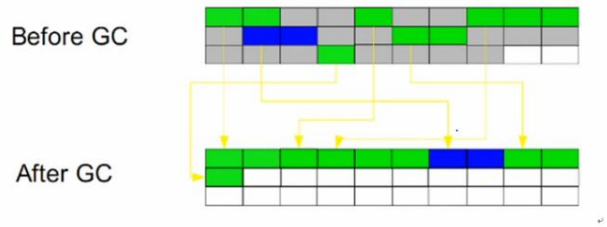
此算法把内存空间划分为两个相等的区域,每次只是用其中一个区域。垃圾回收时,遍历当前使用的区域，把正在使用的对象复制到另外一个区域中。此算法每次只处理正在处理中的对象，因此复制成本比较小，同时复制过去还可以进行碎片的整理，不会出现”碎片”问题，这个算法缺点很明显,需要两倍内存





标记--整理（Copying）

此算法结合了”标记-清楚”和复制两个算法的优点。分为两个部分,第一个阶段从根节点开始标记所有被引用对象,第二阶段遍历整个堆,把清楚未标记对象并且把或者的对象”压缩“到堆中的一块，顺序存放，避免了”标记-清楚”的碎片问题，同时也避免了”复制”算法的空间问题



按系统线程分

串行收集: [1 个核业务 3 个业务]

串行收集单线程处理所有垃圾回收工作,因为无需多线程交互,实现容易,而且效率比较高。局限性明显，无法使用多处理器的优势,所有此收集适合单处理器机器。当然,此收集器也可以用在小数据量(100M)情况下的多处理器机器上

并行收集:

并行收集使用多线程处理垃圾回收工作,因而速度快，效率高。而且理论上 CPU 数量越多，越能体现出并行收集器的优势

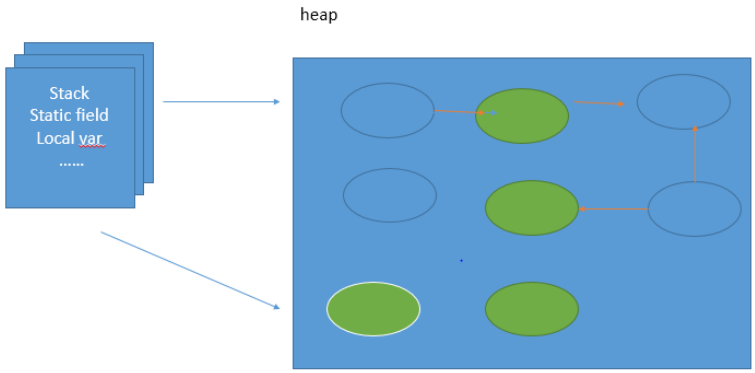
并发收集:

相对于串行收集和并行收集而言,前面两个在进行垃圾回收工作时，需要暂停整个运行环境，因此，系统在垃圾回收时候会有明显的暂停，而且对越大事件越长

三 垃圾回收面临的问题

如何区分垃圾

1 通过”引用计数法”，通过统计控制对象和删除对象时的引用数来做判断，垃圾回收程序收集器为 0 对象即可。但是这种方法无法解决循环引用。所以，2 后来实现的垃圾判断算法中，都是从程序运行的根阶段出发，遍历对象的引用，查找存活对象。那么在这种方式中实现中，垃圾回收从哪里开始？从哪里查找正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是程序执行的地方，所以要获取哪些对象正在使用，需要从 Java 栈开始。同时，一个栈对应一个线程的，因此，如果有多个线程的话，需要这些线程对应的所以栈进行检查。



同时，除了栈为，还有系统运行的寄存器等，也是存储程序运行数据的。这样，以栈或者寄存器为对象，可以最后找到堆中的对象，又从这些对象中对堆中的其他对象引用，这种引用逐步拓展，最终

以 Null 或者基本类型结束，这样就最终形成多颗对象树。在这些对象树上的引用，都是当前系统需要的对象，不能被垃圾回收，而其他的对象，则可以视为无法引用的对象，可以被垃圾回收。
所以，垃圾回收的起点是一些根对象(java 栈,静态变量，寄存器)，而最简单的 java 栈就是 main 函数。这种回收方式，也是""标记--清楚"机制

如何处理碎片

由于不同 java 对象的存货时间是不一定的，因此，在程序运行一段时间后，如果不进行内存整理，就会出现内存碎片。碎片最直接的导致的问题就是无法分配大的内存空间，以及程序运行效率降低。所以，在上面提到的垃圾回收算法中，"复制"和"标记-整理"，可以解决碎片化问题

如果解决同时存在的对象创建和对象回收问题

垃圾回收线程是回收内存的，而程序运行线程则是分配内存的，一个回收内存，一个分配内存，两者看起来是矛盾的，因此，现有的垃圾回收方式中，要进行垃圾回收前，一般需要展厅整个应用，然后进行垃圾回收，回收完后再继续应用，这是一种最直接，最有效的解决二则矛盾问题。

可是，这个有一个明显的弊端，堆空间增大，垃圾回收时间也会相应的持续增大，对应用暂停时间最大要求是几百毫秒，当堆空间大于几十个 G 时，就可能操作这个限制，这种情况下，垃圾回收将会成为系统运行的一个瓶颈。为了解决这种问题，有了并发垃圾回收算法，使用这种算法，垃圾回收线程和程序运行线程同时运行。解决了暂停问题，可是需要在新生对象的同时回收对象，算法复杂度增加，系统的处理能力相应降低，同时，"碎片化"问题比较难解决

四，分带垃圾回收

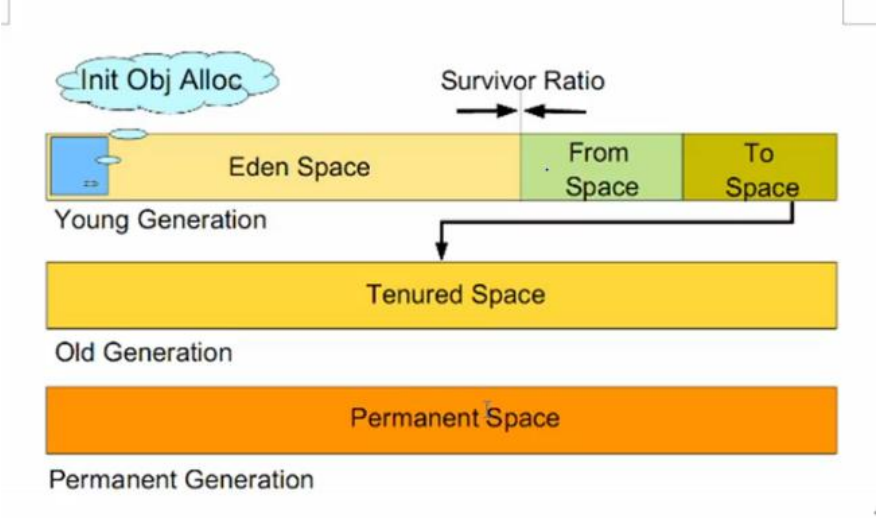
为何要分带

Java 程序中，会产生大量的对象，其中有些对象和业务信息相关

http 请求的 session,线程，Socket 连接，这类对象直接和业务挂钩，因此生命周期会比较长，但是一些对象，主要是程序的零临时变量，这些对象生命周期比较短，比如 String 对象，由于其不变类的特性，系统会产生大量的这些对象，因此有些使用一个就可以回收

如果，在不同对象存活时间区分的情况，每次回收都要对整个堆回收，花费时间相对较长，同时，因为每次垃圾回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为他们可能已经进行到了多次遍历，可是依然存在。因此分带垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上使用最适合他的垃圾回收算法方式进行回收

如何分带



年轻代.年老点，持久代(主要存放 java 类信息，与垃圾回收要收集的 java 对象关系不大)

年轻代:【主要是复制回收算法】

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制"年老区(Tenured)"。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来 对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代:

在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

持久代:

用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=<N>进行设置。

什么时候出发垃圾回收

由于对象进行了分代处理，因此垃圾回收区域，时间也不一样.GC 两种类型 GC 类型

GC 有两种类型：Scavenge GC 和 Full GC。

1. Scavenge GC

一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就好触发 Scavenge GC，堆 Eden 区域进行 GC，清除非存活对象，并且把尚且存活的对象移动到 Survivor 区。然后整理 Survivor 的两个区。

2. Full GC

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 比 Scavenge GC 要慢，因此应该尽可能减少 Full GC。有如下原因可能导致 Full GC：

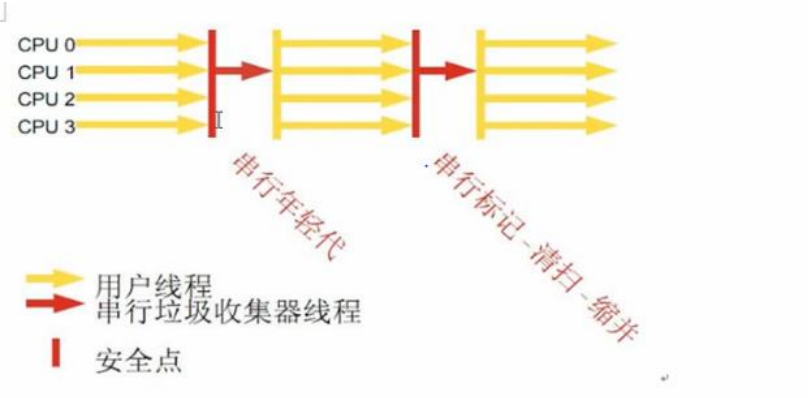
- Tenured(年老代)被写满
- Perm(持久带)域被写满
- System.gc()被显示调用
- 上一次 GC 之后 Heap 的各域分配策略动态变化

选择合适的收集器

目前的收集器主要有三种：串行收集器、并行收集器、并发收集器。

1. 串行收集器

使用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，也无法使用多处理器的优势，所以此收集器适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。可以使用-XX:+UseSerialGC 打开。



2. 并行收集器

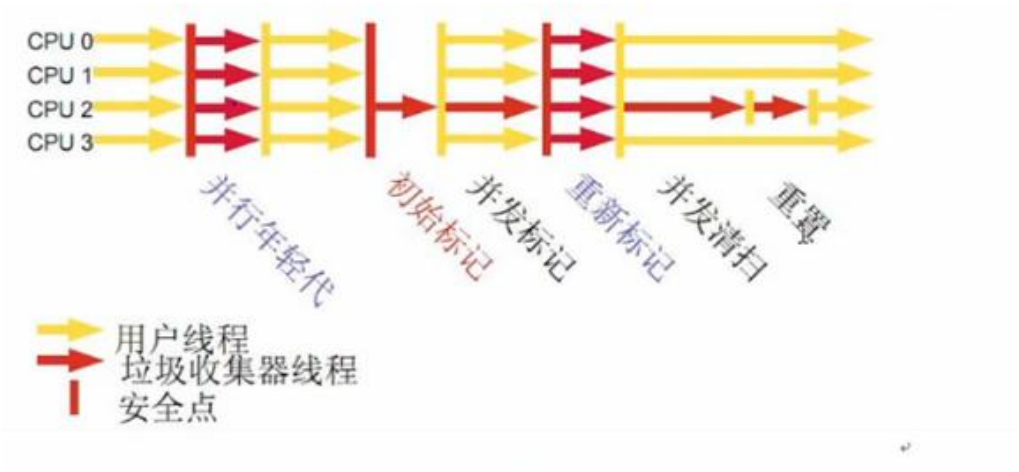
1. 对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。使用-XX:+UseParallelGC.打开。并行收集器在 J2SE5.0 第六 6 更新上引入，在 Java SE6.0 中进行了增强--可以堆年老代进行并行收集。如果年老代不使用并发收集的话，是使用单线程进行垃圾回收，因此会制约扩展能力。使用-XX:+UseParallelOldGC 打开。
2. 使用-XX:ParallelGCThreads=设置并行垃圾回收的线程数。
3. 此收集器可以进行如下配置：

§ 最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=指定。为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。

§ 吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=来设定，公式为 1/（1+N）。例如，-XX:GCTimeRatio=19 时，表示 5%的时间用于垃圾回收。默认情况为 99，即 1%的时间用于垃圾回收。

3. 并发收集器

可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。使用-XX:+UseConcMarkSweepGC 打开。



1. 并 发收集器主要减少年老代的暂停时间，他在应用不停止的情况下使用独立的垃圾回收线程，跟踪可达对象。在每个年老代垃圾回收周期中，在收集初期并发收集器会 对整个应用进行简短的暂停，在收集中还会再暂停一次。第二次暂停会比第一次稍长，在此过程中多个线程同时进行垃圾回收工作。

2. 并发收集器使用处理器换来短暂的停顿时间。在一个 N 个处理器的系统上，并发收集部分使用 K/N 个可用处理器进行回收，一般情况下 1<=K<=N/4。

3. 在只有一个处理器的主机上使用并发收集器，设置为 incremental mode 模式也可获得较短的停顿时间。

4. 浮动垃圾：由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要 20%的预留空间用于这些浮动垃圾。

5. Concurrent Mode Failure：并发收集器在应用运行时进行收集，所以需要保证堆在垃圾回收的这段时间有足够的空间供程序使用，否则，垃圾回收还未完成，堆空间先满了。这种情况下将会发生“并发模式失败”，此时整个应用将会暂停，进行垃圾回收。

6. 启动并发收集器：因为并发收集在应用运行时进行收集，所以必须保证收集完成之前有足够的内存空间供程序使用，否则会出现“Concurrent Mode Failure”。通过设置-XX:CMSInitiatingOccupancyFraction=指定还有多少剩余堆时开始执行并发收集

- 串行处理器：

--适用情况：数据量比较小（100M 左右）；单处理器下并且对响应时间无要求的应用。

--缺点：只能用于小型应用

- 并行处理器：

--适用情况：“对吞吐量有高要求”，多 CPU、对应用响应时间无要求的中、大型应用。举例：后台处理、科学计算。

--缺点：应用响应时间可能较长

- 并发处理器：

--适用情况：“对响应时间有高要求”，多 CPU、对应用响应时间有较高要求的中、大型应用。举例：Web 服务器/应用服务器、电信交换、集成开发环境。

六 常见配置汇总

1. 堆大小设置

-Xmx3550m：设置 JVM 最大可用内存为 3550M。 <4G

-Xms3550m：设置 JVM 促使内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。 <4G -XX:MaxPermSize=16m

-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m:设置持久代大小为 16m。

2 回收器设置

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:+UseSerialGC:设置串行收集器

-XX:UseParalledOldGC:设置并行年老代收集器

-XX:+xUseConcMarkSweepGC:设置并发收集器

3 垃圾回收统计

- **-XX:+PrintGC**

输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs]

[Full GC 121376K->10414K(130112K), 0.0650971 secs]

- **-XX:+PrintGCDetails**

输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs]

118250K->113543K(130112K), 0.0124633 secs]

[GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs]

121376K->10414K(130112K), 0.0436268 secs]

- **-XX:+PrintGCTimeStamps -XX:+PrintGC**：PrintGCTimeStamps 可与上面两个混合使用

输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

- **-XX:+PrintGCApplicationConcurrentTime**:打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用

输出形式：Application time: 0.5291524 seconds

- **-XX:+PrintGCApplicationStoppedTime**：打印垃圾回收期间程序暂停的时间。可与上面混合使用

输出形式：Total time for which application threads were stopped: 0.0468229 seconds

- **-XX:PrintHeapAtGC**:打印 GC 前后的详细堆栈信息

4. 并行收集器设置

§ **-XX:ParallelGCThreads=n**:设置并行收集器收集时使用的 CPU 数。并行收集线程数。

§ **-XX:MaxGCPauseMillis=n**:设置并行收集最大暂停时间

§ **-XX:GCTimeRatio=n**:设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

§ **-XX:+CMSIncrementalMode**:设置为增量模式。适用于单 CPU 情况。

§ **-XX:ParallelGCThreads=n**:设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

五、常见配置举例

1. 堆大小设置

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt 还是 64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32 位系统 下，一般限制在 1.5G~2G；64 为操作系统对内存无限制。我在 Windows Server 2003 系统，3.5G 物理内存，JDK5.0 下测试，最大可设置为 1478m。

典型设置：

o java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置 JVM 最大可用内存为 3550M。 最大堆的大小

-Xms3550m：设置 JVM 促使内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

初始化对大小

-Xmn2g：设置年轻代大小为 2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss128k： 设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内 存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

o java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置 JVM 最大可用内存为 3550M。

-Xms3550m：设置 JVM 促使内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g：设置年轻代大小为 2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss128k： 设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为

1M，以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内 存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

o java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m:设置持久代大小为 16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

回收器选择

JVM 给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0 以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0 以后，JVM 会根据当前系统配置进行判断。

1. 吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

§ java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代

有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

§ java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0 支持对年老代并行收集。

§ java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值。

§ java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

2. 响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

§ java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4 的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn 设置。

-XX:+UseParNewGC:设置年轻代为并行收集。可与 CMS 收集同时使用。JDK5.0 以上，JVM 会根据系统配置自行设置，所以无需再设置此值。

§ java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性

能，但是可以消除碎片

六、调优总结

1. 年轻代大小选择

o 响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

o 吞吐量优先的应用：尽可能的设置大，可能到达 Gbit 的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合 8CPU 以上的应用。

2. 年老代大小选择

o 响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

§ 并发垃圾收集信息
§ 持久代并发收集次数
§ 传统 GC 信息
§ 花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

o 吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

3. 较小堆引起的碎片问题

因 为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间 较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出 现“碎片”，可能需要进行如下配置：

o -XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

o -XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次 Full GC 后，对年老代进行压缩

七 调优工具

Jconsole,jProfile,VisualVM

Jcomsole:jdk 自带，功能简单，可以在系统有一定符合的情况使用，对垃圾算法有详细的跟踪

Jprofile：商业化软件，需要付费，功能强大

VisualVM:jdk 自带，功能强大

八 常见异常

1 持久代堆占满

Spring 中，很容易出现持久代内存满的情况

异常:java.lang.OutOfMemoryError:PermGen space(持久代占满)

说明：Perm 空间被占满，无法为新的 class 分配存储空间而引发的异常。这个异常以前是没有的，现在 java 反射大量使用的今天这个异常比较常见了。主要原因是大量动态反射生成的类不断地被加载，最终导致 Perm 区被占满

更可怕的是，不同的 classloader 即便使用了相同的类，但都会对其加载，相遇于一个东西，被 N 个 classLoader 那么就会被加载 N 次。因此，某些情况下，这个问题被视为误解。当然，存在大量 classloader 和大量发射类的情况其实也不多

解决

1 –xx:MaxPermSize=16m

2 换用 JDK 比如 Jrocket

2 系统内存占满

异常:java.lang.OutOfMemoryError:unable to create new native thread

说明：这个异常由操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在 java 堆中分配内存在，操作系统本身也需要分配资源来创建线程。因此，当线程数量达到一定程序以后，堆中获取还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

解决

分配给 java 虚拟机的内存越多，系统剩余的资源就越少，因此，当系统内存固定时，分配给 java 虚拟机的内存越多，那么，系统总共产生的线程就越少，两者成反比关系。同样可以修改-Xss 来减少分配给单个线程的空间，也可以增加系统总共内生产的线程。

3 堆栈溢出

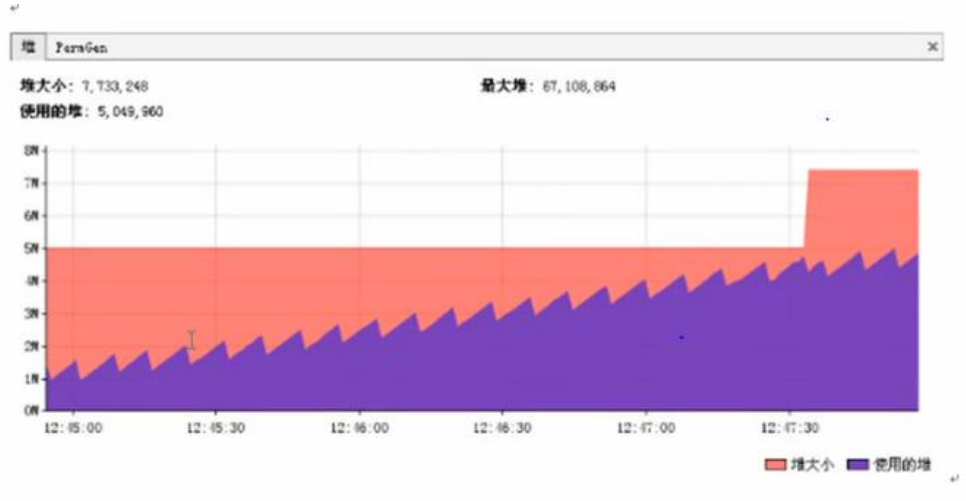
异常：java.lang.StackOverflowError

说明:一般是递归没返回，循环调用造成

4 年老代对空间被占满

异常：java.lang.OutOfMemoryError.Java heap space

说明:



这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再分配新的空间

上图，就是一种典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有低谷部分表示一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随着时间的推移，系统的堆空间不断暂满，最终导致暂满整个堆空间，因此可能初步可以认为是发生了内存泄漏。

解决：

这种方式解决起来比较容易，一般就是根据垃圾回收前后情况对比，根据对象引用情况(常见集合对象引用)分析，基本可以找到泄漏点

八 tomcat java JVM 配置例子【根据 [exception]

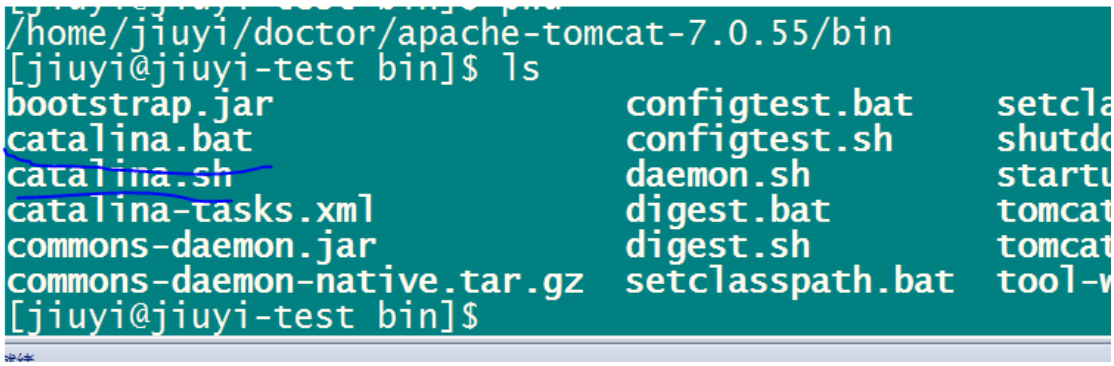
上面的情况举例】

位置:

windows %TOMCAT_HOME%\bin\catalina.bat

Linux %TOMCAT_HOME%\bin\catalina.sh

下图为 Linux 环境

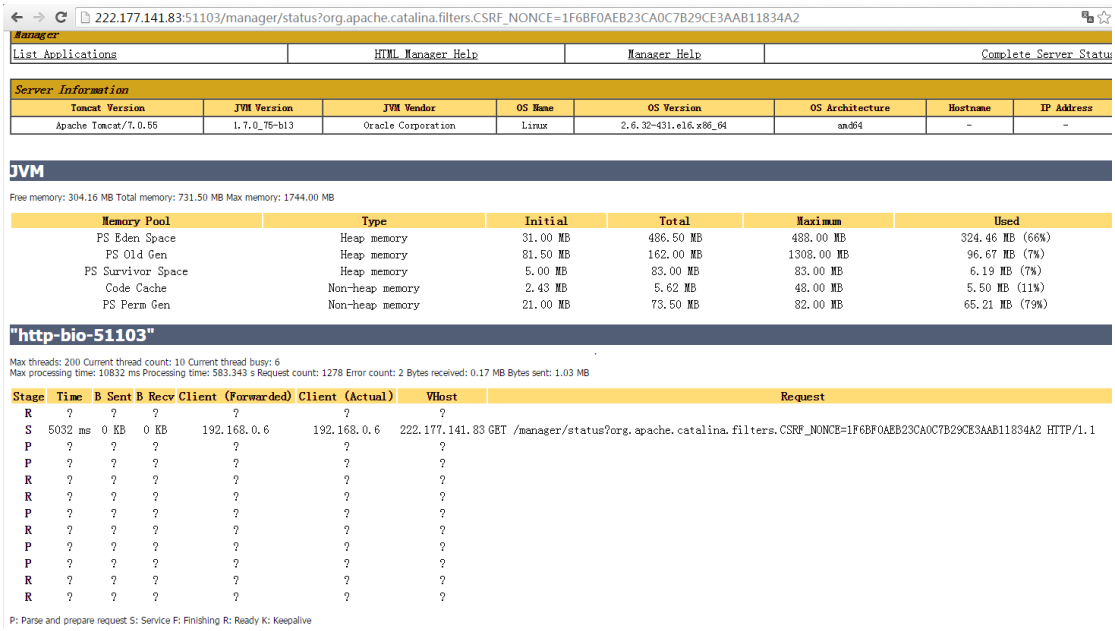


```
JAVA_OPTS       (Optional) Java runtime options used when any command
                is executed.
                Include here and not in CATALINA_OPTS all options, that
                should be used by Tomcat and also by the stop process,
                the version command etc.
                Most options should go into CATALINA_OPTS.
```

JAVA_OPTS 设置 jvm 参数

如果想看该 tomcat 下的 jvm 一种方法是通过访问管理页面 需要先在 tomcat 配置文件中加入用户

详细



Manager

List Applications	HTML Manager Help	Manager Help	Complete Server Status

Server Information

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture	Hostname	IP Address
Apache Tomcat/7.0.55	1.7.0_75-b13	Oracle Corporation	Linux	2.6.32-431.el6.x86_64	amd64	-	-

JVM

Free memory: 304.16 MB Total memory: 731.50 MB Max memory: 1744.00 MB

Memory Pool	Type	Initial	Total	Maximum	Used
PS Eden Space	Heap memory	31.00 MB	486.50 MB	488.00 MB	324.46 MB (66%)
PS Old Gen	Heap memory	81.50 MB	162.00 MB	1308.00 MB	96.67 MB (7%)
PS Survivor Space	Heap memory	5.00 MB	83.00 MB	83.00 MB	6.19 MB (7%)
Code Cache	Non-heap memory	2.43 MB	5.62 MB	48.00 MB	5.50 MB (11%)
PS Perm Gen	Non-heap memory	21.00 MB	73.50 MB	82.00 MB	65.21 MB (79%)

"http-bio-51103"

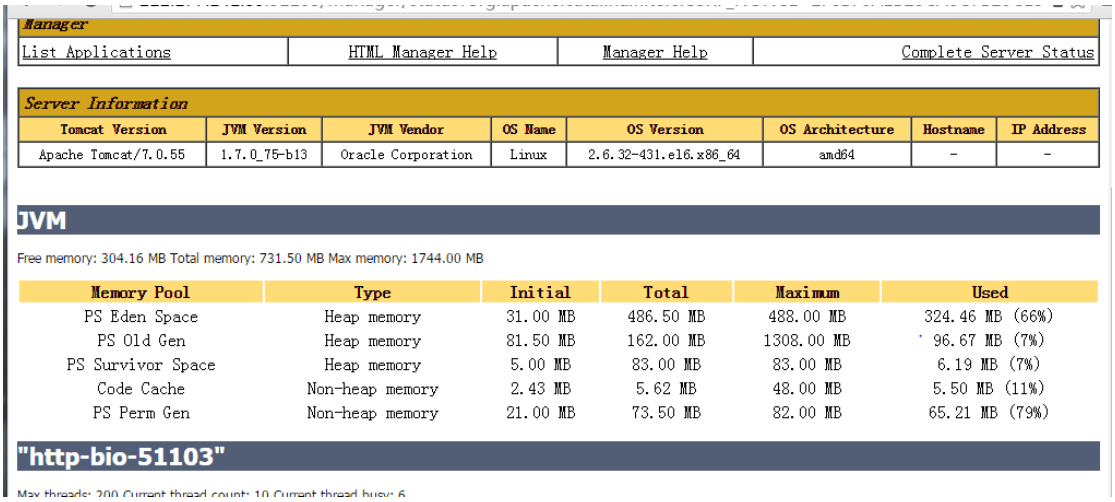
Max threads: 200 Current thread count: 10 Current thread busy: 6
Max processing time: 10832 ms Processing time: 583.343 s Request count: 1278 Error count: 2 Bytes received: 0.17 MB Bytes sent: 1.03 MB

Stage	Time	B Sent	B Recv	Client (Forwarded)	Client (Actual)	VHost	Request
R	?	?	?	?	?	?	
S	5032 ms	0 KB	0 KB	192.168.0.6	192.168.0.6	222.177.141.83	GET /manager/status?org.apache.catalina.filters.CSRF_NONCE=1F6BF0AEB23CA0C7B29CE3AAB11834A2 HTTP/1.1
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
R	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
R	?	?	?	?	?	?	

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

可以看到使用率

从图中可以看到因为我的项目中遇到的反射比较多，所以 PS Perm Gen 使用率比较高



Manager

List Applications	HTML Manager Help	Manager Help	Complete Server Status

Server Information

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture	Hostname	IP Address
Apache Tomcat/7.0.55	1.7.0_75-b13	Oracle Corporation	Linux	2.6.32-431.el6.x86_64	amd64	-	-

JVM

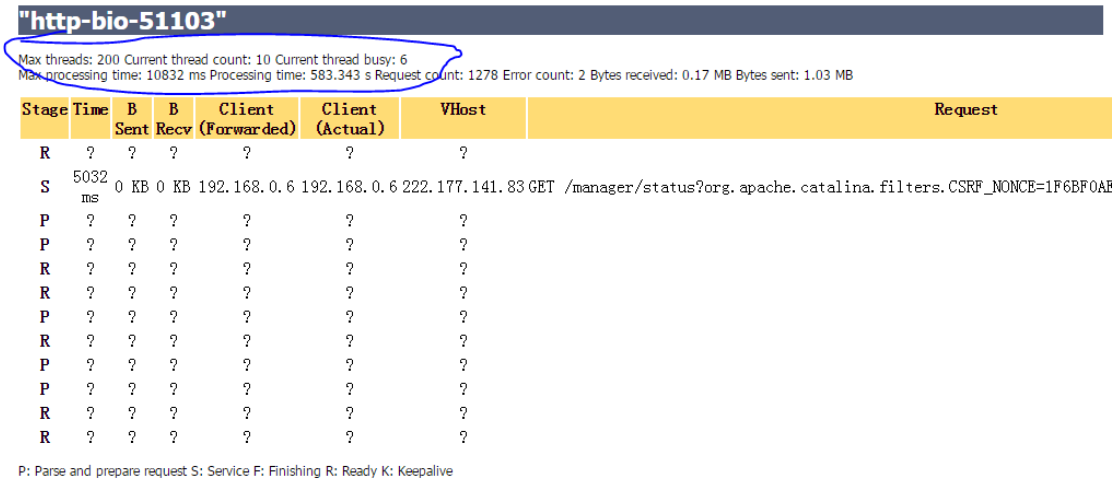
Free memory: 304.16 MB Total memory: 731.50 MB Max memory: 1744.00 MB

Memory Pool	Type	Initial	Total	Maximum	Used
PS Eden Space	Heap memory	31.00 MB	486.50 MB	488.00 MB	324.46 MB (66%)
PS Old Gen	Heap memory	81.50 MB	162.00 MB	1308.00 MB	96.67 MB (7%)
PS Survivor Space	Heap memory	5.00 MB	83.00 MB	83.00 MB	6.19 MB (7%)
Code Cache	Non-heap memory	2.43 MB	5.62 MB	48.00 MB	5.50 MB (11%)
PS Perm Gen	Non-heap memory	21.00 MB	73.50 MB	82.00 MB	65.21 MB (79%)

"http-bio-51103"

Max threads: 200 Current thread count: 10 Current thread busy: 6

线程处理，最大线程，活动线程 200 busy 6



"http-bio-51103"

Max threads: 200 Current thread count: 10 Current thread busy: 6
Max processing time: 10832 ms Processing time: 583.343 s Request count: 1278 Error count: 2 Bytes received: 0.17 MB Bytes sent: 1.03 MB

Stage	Time	B Sent	B Recv	Client (Forwarded)	Client (Actual)	VHost	Request
R	?	?	?	?	?	?	
S	5032 ms	0 KB	0 KB	192.168.0.6	192.168.0.6	222.177.141.83	GET /manager/status?org.apache.catalina.filters.CSRF_NONCE=1F6BF0AE
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
R	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	
R	?	?	?	?	?	?	
R	?	?	?	?	?	?	

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

在 tomcat 配置文件 server.xml 中的<Connector ... />配置中，和连接数相关的参数有：

minProcessors：最小空闲连接线程数，用于提高系统处理性能，默认值为 10

maxProcessors：最大连接线程数，即：并发处理的最大请求数，默认值为 75

acceptCount：允许的最大连接数，应大于等于 maxProcessors，默认值为 100

enableLookups：是否反查域名，取值为：true 或 false。为了提高处理能力，应设置为 false

connectionTimeout：网络连接超时，单位：毫秒。设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000 毫秒。

```
<Service name="Catalina">

  <!--The connectors can use a shared executor, you can define one or mor
  <!--
  <Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
      maxThreads="150" minSpareThreads="4"/>
  -->


  <!-- A "Connector" represents an endpoint by which requests are receive
       and responses are returned. Documentation at :
       Java HTTP Connector: /docs/config/http.html (blocking & non-blocki
       Java AJP  Connector: /docs/config/ajp.html
       APR (HTTP/AJP) Connector: /docs/apr.html
       Define a non-SSL HTTP/1.1 Connector on port 8080
  -->
  <Connector executor="tomcatThreadPool"
             port="51103" protocol="HTTP/1.1"
             connectionTimeout="20000"
             redirectPort="8443" />
  <!-- A "Connector" using the shared thread pool-->
  <!--
```

Web Server 最大的连接数收到 linux 内核限制

linux 最大进程数

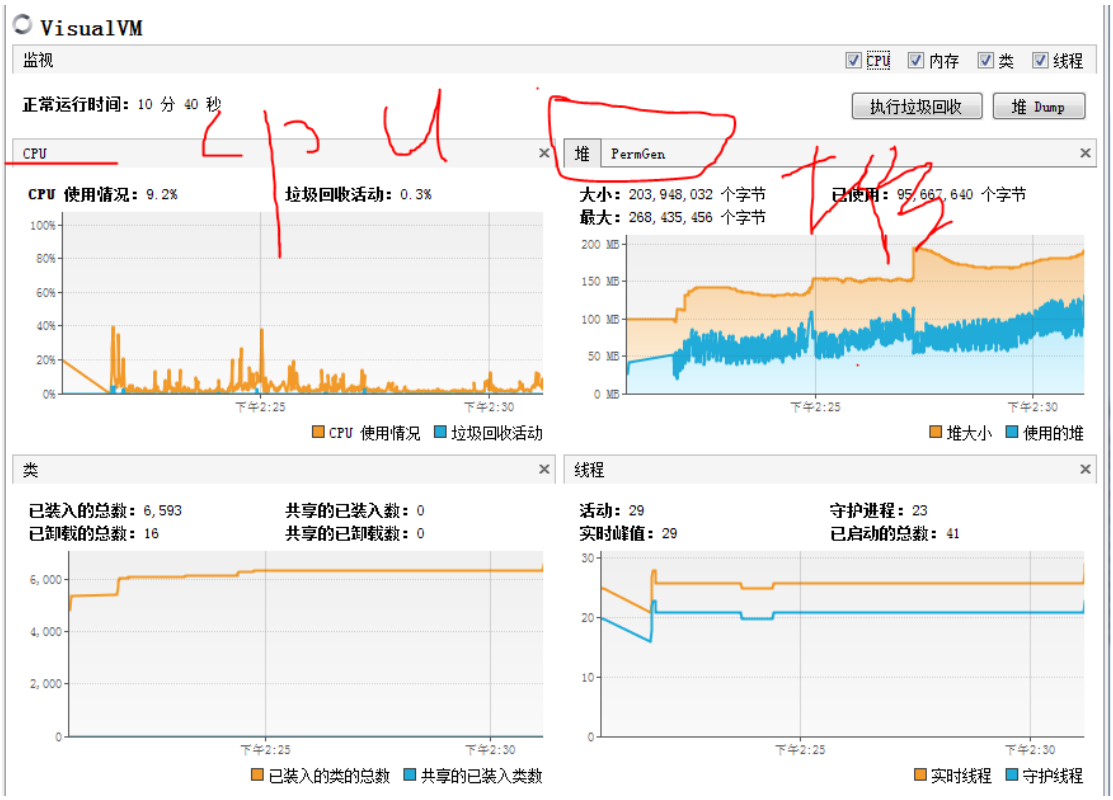
```
[jiuyi@jiuyi-test ~]$ cat /proc/sys/kernel/pid_max
32768
[jiuyi@jiuyi-test ~]$
```

Linux 最大线程数

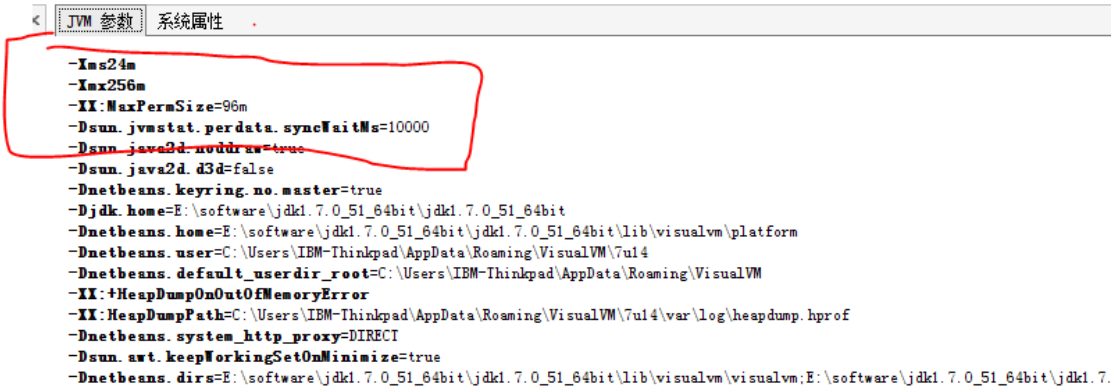
```
cat: /proc/sys/kernel/thread_max: æ²¡æ‰¾åˆ°æ–‡ä»¶æˆ–ç›®å½•
[jiuyi@jiuyi-test ~]$ cat /proc/sys/kernel/threads-max
125145
[jiuyi@jiuyi-test ~]$
```

九 VisualVm 工具分析使用

很方便查看堆栈信息,线程，内存，CPU



虚拟机参数



十 jMap 分析[linux]

Linux 中很多工具分析 jvm 在 bin 下

测试服务器在/usr/java/jdk/jdk1.7*/bin

/usr/java/jdk/jdk1.7.0_75/bin

```
[jiuyi@jiuyi-test bin]$ ls
appletviewer  javac            jconsole  jps              keytool       schemagen
apt           javadoc          jcontrol  jrunscript       native2ascii  serialver
ControlPanel  javafxpackager   jdb       jsadebugd        orbd          servertool
extcheck      javah            jhat      jstack           pack200       tnameserv
idlj          javap            jinfo     jstat            policytool    unpack200
jar           java-rmi.cgi     jmap      jstatd           rmic          wsgen
jarsigner     javaws           jmc       jstatd.all.policy rmid         wsimport
java          jcmd             jmc.ini   jvisualvm        rmiregistry   xjc
[jiuyi@jiuyi-test bin]$ jps
```

```
[root@taotaoPortal ~]# jmap
Usage:
    jmap [option] <pid>
        (to connect to running process)
    jmap [option] <executable <core>
        (to connect to a core file)
    jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
    <none>               to print same info as Solaris pmap
    -heap                to print java heap summary
    -histo[:live]        to print histogram of java object heap; if the "live"
                         suboption is specified, only count live objects
    -clstats             to print class loader statistics
    -finalizerinfo       to print information on objects awaiting finalization
    -dump:<dump-options> to dump java heap in hprof binary format
                         dump-options:
                           live         dump only live objects; if not specified,
                                        all objects in the heap are dumped.
                           format=b     binary format
                           file=<file>  dump heap to <file>
                         Example: jmap -dump:live,format=b,file=heap.bin <pid>
    -F                   force. Use with -dump:<dump-options> <pid> or -histo
                         to force a heap dump or histogram when <pid> does not
                         respond. The "live" suboption is not supported
                         in this mode.
    -h | -help           to print this help message
    -J<flag>             to pass <flag> directly to the runtime system
[root@taotaoPortal ~]#
```

查看聊天服务器 JVM

ps –ef | grep java

```
[jiuyi@jiuyi-test ~]$ ps -ef | grep java
jiuyi     1156     1  0 Aug09 ?        00:03:22 /usr/bin/java -Djava.util.loggi
ng.config.file=/home/jiuyi/weixin/apache-tomcat-7.0.69/conf/logging.properties
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djdk.tls.eph
emeralDHKeySize=2048 -Djava.endorsed.dirs=/home/jiuyi/weixin/apache-tomcat-7.0.
69/endorsed -classpath /home/jiuyi/weixin/apache-tomcat-7.0.69/bin/bootstrap.ja
r:/home/jiuyi/weixin/apache-tomcat-7.0.69/bin/tomcat-juli.jar -Dcatalina.base=/
home/jiuyi/weixin/apache-tomcat-7.0.69 -Dcatalina.home=/home/jiuyi/weixin/apach
e-tomcat-7.0.69 -Djava.io.tmpdir=/home/jiuyi/weixin/apache-tomcat-7.0.69/temp o
rg.apache.catalina.startup.Bootstrap start
root      2120     1  0 Jul29 ?        00:09:33 /usr/java/jdk/jdk1.7.0_75/bin/j
ava -Djava.util.logging.config.file=/home/jiuyi/vggle/apache-tomcat-7.0.55/conf
/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogM
anager -Djava.endorsed.dirs=/home/jiuyi/vggle/apache-tomcat-7.0.55/endorsed -cl
asspath /home/jiuyi/vggle/apache-tomcat-7.0.55/bin/bootstrap.jar:/home/jiuyi/vg
gle/apache-tomcat-7.0.55/bin/tomcat-juli.jar -Dcatalina.base=/home/jiuyi/vggle/
apache-tomcat-7.0.55 -Dcatalina.home=/home/jiuyi/vggle/apache-tomcat-7.0.55 -Dj
ava.io.tmpdir=/home/jiuyi/vggle/apache-tomcat-7.0.55/temp org.apache.catalina.s
tartup.Bootstrap start start
jiuyi     7508  7406  0 20:43 pts/3    00:00:00 grep java
jiuyi    12413     1  0 Aug04 ?        00:18:57 /usr/java/jdk/jdk1.7.0_75/bin/j
ava -Djava.util.logging.config.file=/home/jiuyi/register/apache-tomcat-7.0.55/c
onf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderL
ogManager -Djava.endorsed.dirs=/home/jiuyi/register/apache-tomcat-7.0.55/endors
ed -classpath /home/jiuyi/register/apache-tomcat-7.0.55/bin/bootstrap.jar:/home
```

堆配置

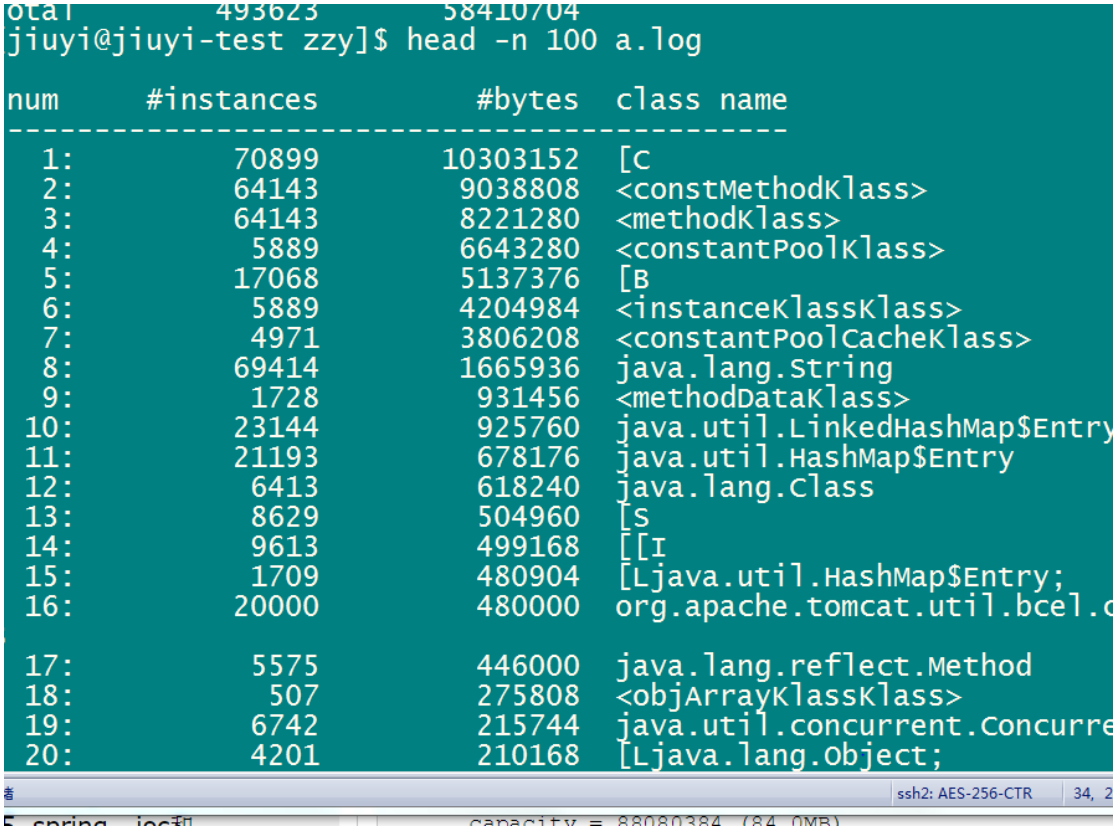
```
[jiuyi@jiuyi-test ~]$ jmap -heap 31556
Attaching to process ID 31556, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 24.75-b04

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio = 0
   MaxHeapFreeRatio = 100
   MaxHeapSize      = 2057306112 (1962.0MB)
   NewSize          = 1310720 (1.25MB)
   MaxNewSize       = 17592186044415 MB
   OldSize          = 5439488 (5.1875MB)
   NewRatio         = 2
   SurvivorRatio    = 8
   PermSize         = 21757952 (20.75MB)
   MaxPermSize      = 85983232 (82.0MB)
   G1HeapRegionSize = 0 (0.0MB)
```

堆使用情况

```
Heap Usage:
PS Young Generation
Eden Space:
   capacity = 297271296 (283.5MB)
   used     = 24994184 (23.83631134033203MB)
   free     = 272277112 (259.66368865966797MB)
   8.407869961316413% used
From Space:
   capacity = 37748736 (36.0MB)
   used     = 25423936 (24.24615478515625MB)
   free     = 12324800 (11.75384521484375MB)
   67.35042995876736% used
To Space:
   capacity = 38273024 (36.5MB)
   used     = 0 (0.0MB)
   free     = 38273024 (36.5MB)
   0.0% used
PS Old Generation
   capacity = 115343360 (110.0MB)
   used     = 38845912 (37.046348571777344MB)
   free     = 76497448 (72.95365142822266MB)
   33.678498701615766% used
PS Perm Generation
   capacity = 35651584 (34.0MB)
   used     = 35416160 (33.775482177734375MB)
   free     = 235424 (0.224517822265625MB)
   99.33965346392463% used
```

```
2313:             1             16  com.jiuyi.qujiuyi.service.syscfg.impl.SysCfgServi
ceImpl
2314:             1             16  java.util.Collections$EmptyIterator
2315:             1             16  java.security.ProtectionDomain$1
Total        493623       58410704
[jiuyi@jiuyi-test ~]$ jmap -histo:live 31556 > ~/zzy/a.log
[jiuyi@jiuyi-test ~]$ cd ~/zzy/
[jiuyi@jiuyi-test zzy]$
```

查看聊天服务器存在的实例



Resion jvm 和 tomcat 类似

<jvm-arg>-Xms 512 m</jvm-arg>

（ jvm 最小内存，也是启动 resin 后的默认内存分配值）

<jvm-arg>-Xmx 512 m</jvm-arg> <!-- make ms=mx to reduce GC times -->

（ jvm 最大内存，当内存使用超过 Xms 分配的值之后会自动向这个最大值提升，一般配置成最大最小值相等，理论上能够降低 GC 垃圾收集的时间，可按实际进行配置）

<jvm-arg>-Xmn 86 m</jvm-arg>

（内存分配增量，当内存需求超过 Xms 值之后进行第一次分配请求的内存值，一般为 Xmx 的 1/3-1/4 ；开始时候可以先屏蔽，当应用出现 OutOfMemory 的时候再打开也可以）

<jvm-arg>-XX:MaxNewSize= 256 m</jvm-arg>

<jvm-arg>-XX:PermSize= 128 m</jvm-arg>

<jvm-arg>-XX:MaxPermSize= 256 m</jvm-arg>

（以上三项是为了减少 OutOfMemory 而配置的，是每个 java 编译执行的时候最多能一次申请 jvm 内存空间的值，以上默认配置基本够用，但依然出 OutOfMemory 的时候可以适当调大，

但不能超越 Xmx 的值；开始时候可以先屏蔽，当应用出现 OutOfMemory 的时候再打开也可以）
<jvm-arg>-Xss256k</jvm-arg> <!-- jvm Stack config -->
<jvm-arg>-Djava.awt.headless=true</jvm-arg> < （允许使用验证码）
<jvm-arg>-Djava.net.preferIPv4Stack=true</jvm-arg> <!-- disable IPv6 -->
<jvm-arg>-Doracle.jdbc.V8Compatible=true</jvm-arg>（针对 oracle10 的兼容配置）
<watchdog-arg>-Dcom.sun.management.jmxremote</watchdog-arg>

<!-- 强制 resin 强制重起时的最小空闲内存 -->
<memory-free-min> 2 M</memory-free-min>

<!-- 最大线程数量 . -->
<thread-max>256</thread-max>

<!-- 套接字等待时间 -->
<socket-timeout>65s</socket-timeout>

<!-- 配置 keepalive -->
<keepalive-max>128</keepalive-max>
<keepalive-timeout>15s</keepalive-timeout>