

## 常见面试题

问题答案在文中都有提到

- 如何判断对象是否死亡（两种方法）。
- 简单的介绍一下强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）。
- 如何判断一个常量是废弃常量
- 如何判断一个类是无用的类
- 垃圾收集有哪些算法，各自的特点？
- HotSpot 为什么要分为新生代和老年代？
- 常见的垃圾回收器有哪些？
- 介绍一下 CMS，G1 收集器。
- Minor Gc 和 Full GC 有什么不同呢？

## 本文导线



当需要排查各种内存溢出问题、当垃圾收集成为系统达到更高并发的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

# 1 揭开JVM内存分配与回收的神秘面纱

Java 的自动内存管理主要是针对对象内存的回收和对象内存的分配。同时，Java 自动内存管理最核心的功能是 **堆** 内存中对象的分配与回收。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆**（Garbage Collected Heap）。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

**堆空间的基本结构：**



上图所示的 eden 区、s0("From") 区、s1("To") 区都属于**新生代**，tentired 区属于**老年代**。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s1("To")，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

经过这次GC后，Eden区和"From"区已经被清空。这个时候，"From"和"To"会交换他们的角色，也就是新的"To"就是上次GC前的 "From"，新的"From"就是上次GC前的"To"。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到 "To" 区被填满，"To"区被填满之后，会将所有对象移动到老年代中。



## 1.1 对象优先在 eden 区分配

目前主流的垃圾收集器都会采用分代回收算法，因此需要将堆内存分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

大多数情况下，对象在新生代中 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 Minor GC。下面我们来进行实际测试一下。

在测试之前我们先来看看 **Minor GC 和 Full GC 有什么不同呢？**

- 新生代 GC (Minor GC)：指发生新生代的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- 老年代 GC (Major GC/Full GC)：指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

针对HotSpot VM的实现，它里面的GC其实准确分类只有两大种：

Partial GC: 并不收集整个GC堆的模式

Young GC: 只收集young gen的GC

Old GC: 只收集old gen的GC。只有CMS的concurrent collection是这个模式

Mixed GC: 收集整个young gen以及部分old gen的GC。只有G1有这个模式

Full GC: 收集整个堆, 包括young gen、old gen、perm gen (如果存在的话) 等所有部分的模式。

Major GC通常是跟full GC是等价的, 收集整个GC堆。但因为HotSpot VM发展了这么多年, 外界对各种名词的解读已经完全混乱了, 当有人说“major GC”的时候一定要问清楚他想要指的是上面的full GC还是old GC。

最简单的分代式GC策略, 按HotSpot VM的serial GC的实现来看, 触发条件是:

young GC: 当young gen中的eden区分配满的时候触发。注意young GC中有部分存活对象会晋升到old gen, 所以young GC后old gen的占用量通常会有所升高。

full GC: 当准备要触发一次young GC时, 如果发现统计数据说之前young GC的平均晋升大小比目前old gen剩余的空间大, 则不会触发young GC而是转为触发full GC (因为HotSpot VM的GC里, 除了CMS的concurrent collection之外, 其它能收集old gen的GC都会同时收集整个GC堆, 包括young gen, 所以不需要事先触发一次单独的young GC); 或者, 如果有perm gen的话, 要在perm gen分配空间但已经没有足够空间时, 也要触发一次full GC; 或者System.gc()、heap dump带GC, 默认也是触发full GC。

HotSpot VM里其它非并发GC的触发条件复杂一些, 不过大致的原理与上面说的其实一样。

当然也总有例外。Parallel Scavenge (-XX:+UseParallelGC) 框架下, 默认是在要触发full GC前先执行一次young GC, 并且两次GC之间能让应用程序稍微运行一小下, 以期降低full GC的暂停时间 (因为young GC会尽量清理了young gen的死对象, 减少了full GC的工作量)。控制这个行为的VM参数是-XX:+ScavengeBeforeFullGC。这是HotSpot VM里的奇葩。可跳传送门围观:

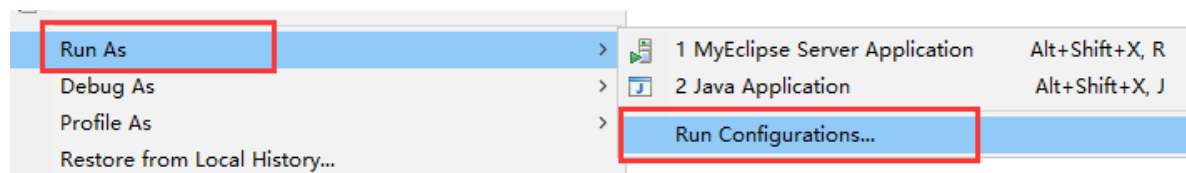
[JVM full GC的奇怪现象, 求解惑? - RednaxelaFX 的回答](#)

并发GC的触发条件就不太一样。以CMS GC为例, 它主要是定时去检查old gen的使用量, 当使用量超过了触发比例就会启动一次CMS GC, 对old gen做并发收集。

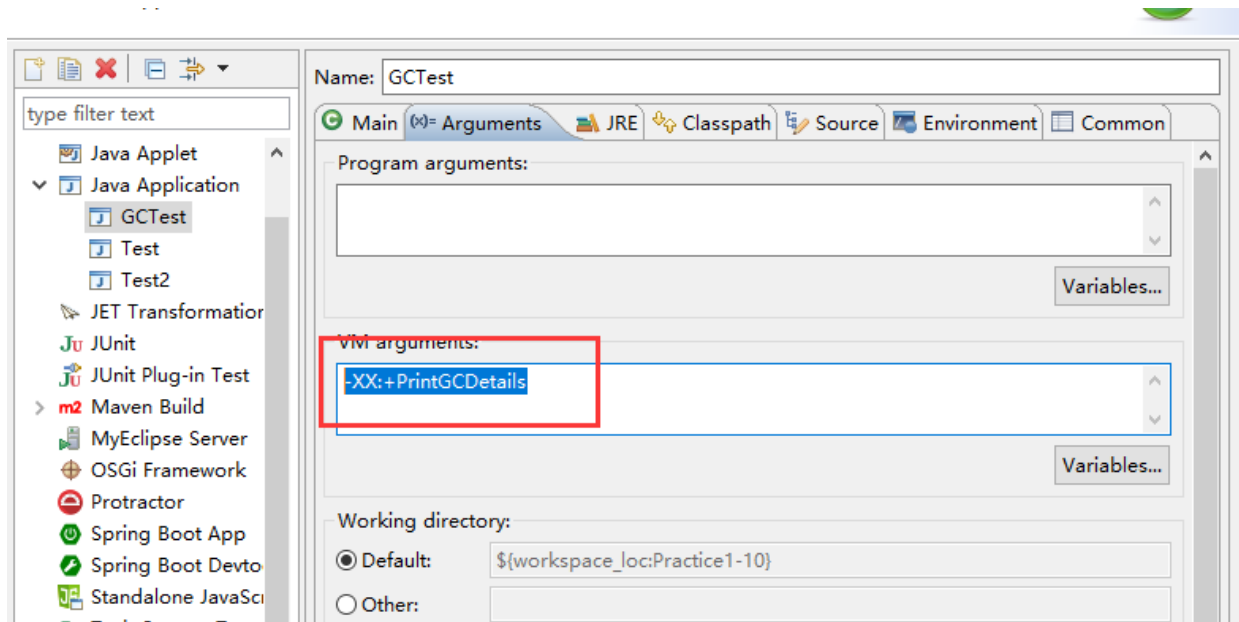
测试:

```
1 public class GCTest {
2     public static void main(String[] args) {
3         byte[] allocation1, allocation2;
4         allocation1 = new byte[30900*1024];
5         //allocation2 = new byte[900*1024];
6     }
7 }
```

通过以下方式运行:



添加的参数: -XX:+PrintGCDetails



运行结果（元空间描述有误，应该是对应于 JDK1.7 的永久代）：

```

Heap
PSYoungGen      total 38400K, used 33280K [0x00000000d5d00000, 0x00000000d8780000, 0x0000000100000000)
eden space 33280K, 100% used [0x00000000d5d00000, 0x00000000d7d80000, 0x00000000d7d80000)
from space 5120K, 0% used [0x00000000d8280000, 0x00000000d8280000, 0x00000000d8780000)
to space 5120K, 0% used [0x00000000d7d80000, 0x00000000d7d80000, 0x00000000d8280000)
ParOldGen       total 87552K, used 0K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 0% used [0x0000000081600000, 0x0000000081600000, 0x0000000086b80000)
Metaspace       used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space     used 283K, capacity 386K, committed 512K, reserved 1048576K
  
```

新生代      使用完全      老年代      元空间对应于 JDK1.8 的永久代

从上图我们可以看出 eden 区内存几乎已经被分配完全（即使程序什么也不做，新生代也会使用 2000 多 k 内存）。假如我们再为 allocation2 分配内存会出现什么情况呢？

```
1 allocation2 = new byte[900*1024];
```

```

[GC (Allocation Failure) [PSYoungGen: 32897K->768K(38400K)] 32897K->31676K(125952K), 0.0229658 secs] [Times: us
Heap
PSYoungGen      total 38400K, used 2001K [0x00000000d5d00000, 0x00000000da800000, 0x0000000100000000)
eden space 33280K, 3% used [0x00000000d5d00000, 0x00000000d5e344b8, 0x00000000d7d80000)
from space 5120K, 15% used [0x00000000d7d80000, 0x00000000d7e40030, 0x00000000d8280000)
to space 5120K, 0% used [0x00000000da300000, 0x00000000da300000, 0x00000000da800000)
ParOldGen       total 87552K, used 30908K [0x0000000081600000, 0x0000000086b80000, 0x00000000d5d00000)
object space 87552K, 35% used [0x0000000081600000, 0x000000008342f010, 0x0000000086b80000)
Metaspace       used 2621K, capacity 4486K, committed 4864K, reserved 1056768K
class space     used 283K, capacity 386K, committed 512K, reserved 1048576K
  
```

**简单解释一下为什么会出现这种情况：** 因为给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，我们刚刚讲了当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。GC 期间虚拟机又发现 allocation1 无法存入 Survivor 空间，所以只好通过 **分配担保机制** 把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存。可以执行如下代码验证：

```

1 public class GCTest {
2     public static void main(String[] args) {
3         byte[] allocation1, allocation2, allocation3, allocation4, allocation5;
  
```

```
4  allocation1 = new byte[32000*1024];
5  allocation2 = new byte[1000*1024];
6  allocation3 = new byte[1000*1024];
7  allocation4 = new byte[1000*1024];
8  allocation5 = new byte[1000*1024];
9  }
10 }
```

## 1.2 大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。

为什么要这样呢？

为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

## 1.3 长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 中每熬过一次 MinorGC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

## 1.4 动态对象年龄判定

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

“Hotspot遍历所有对象时，按照年龄从小到大对其所占用大小进行累积，当累积的某个年龄大小超过了survivor区的一半时，取这个年龄和MaxTenuringThreshold中更小的一个值，作为新的晋升年龄阈值”。

动态年龄计算的代码如下：

```
uint ageTable::compute_tenuring_threshold(size_t survivor_capacity) {
    //survivor_capacity是survivor空间的大小
    size_t desired_survivor_size = (size_t)((double)
    survivor_capacity)*TargetSurvivorRatio/100);
    size_t total = 0;
    uint age = 1;
    while (age < table_size) {
        total += sizes[age]; //sizes数组是每个年龄段对象大小
        if (total > desired_survivor_size) break;
    }
}
```

```
age++;  
}  
uint result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;  
...  
}
```

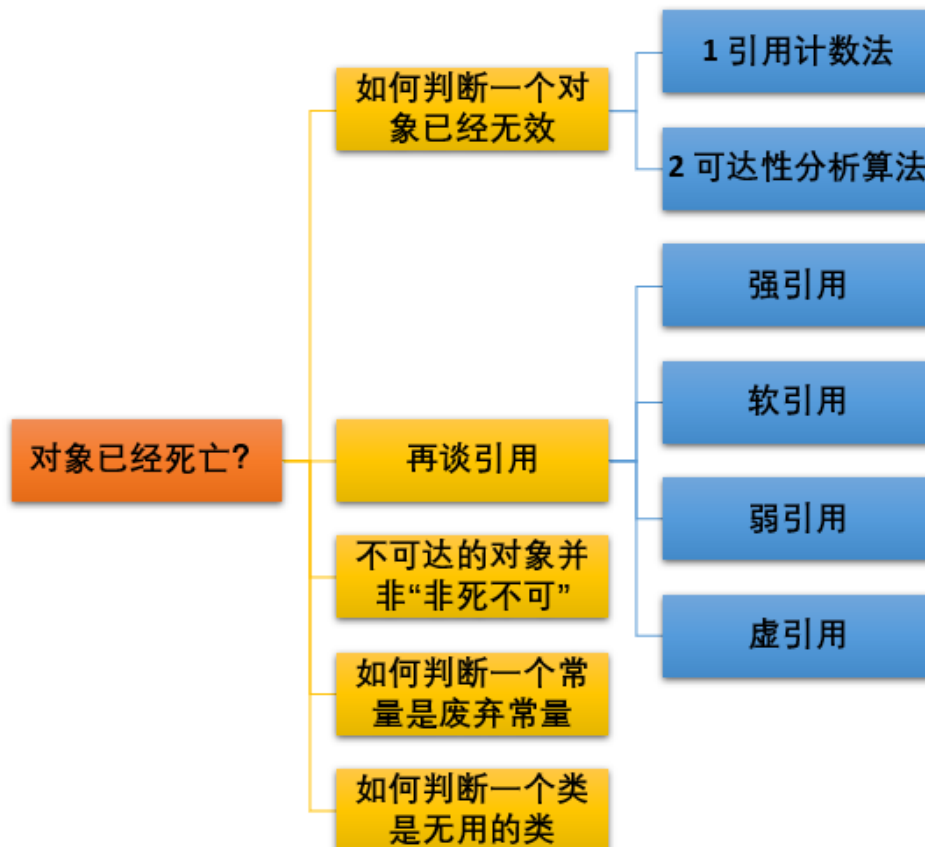
补充说明：关于默认的晋升年龄是15，这个说法的来源大部分都是《深入理解Java虚拟机》这本书。如果你去Oracle的官网阅读相关的虚拟机参数，你会发现-

XX:MaxTenuringThreshold=threshold这里有个说明

Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector.默认晋升年龄并不都是15，这个是要区分垃圾收集器的，CMS就是6。

## 2 对象已经死亡？

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。



### 2.1 引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。所谓对象之间的相互引用问

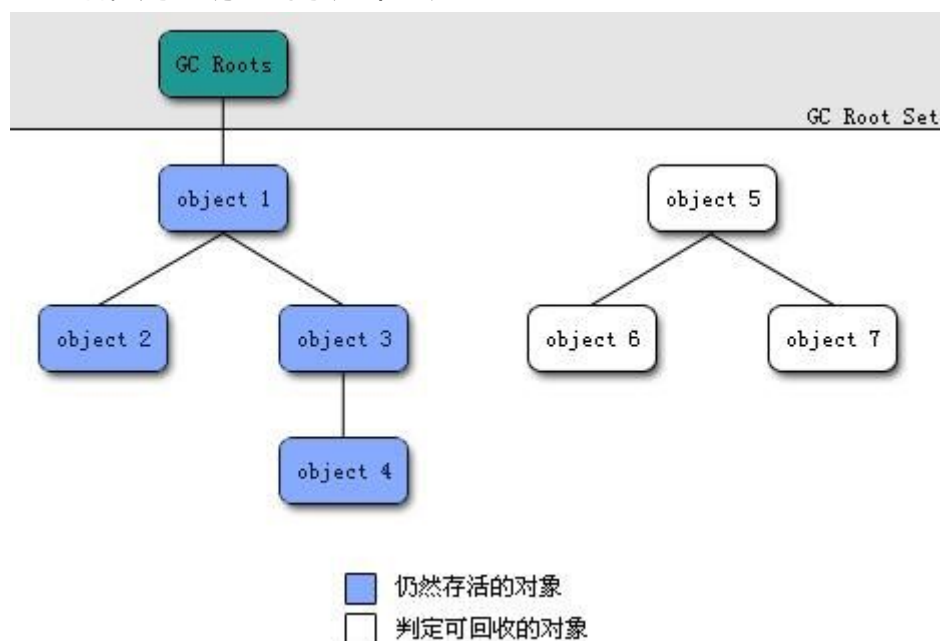


题，如下面代码所示：除了对象 objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为 0，于是引用计数算法无法通知 GC 回收器回收他们。

```
1 public class ReferenceCountingGc {
2     Object instance = null;
3     public static void main(String[] args) {
4         ReferenceCountingGc objA = new ReferenceCountingGc();
5         ReferenceCountingGc objB = new ReferenceCountingGc();
6         objA.instance = objB;
7         objB.instance = objA;
8         objA = null;
9         objB = null;
10    }
11 }
```

## 2.2 可达性分析算法

这个算法的基本思想就是通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。



## 2.3 再谈引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2 之前，Java 中引用的定义很传统：如果 reference 类型的数据存储的数值代表的是另一块内存的起始地址，就称这块内存代表一个引用。

JDK1.2 以后，Java 对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（引用强度逐渐减弱）

### 1. 强引用 (StrongReference)

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于**必不可少**的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

### 2. 软引用 (SoftReference)

如果一个对象只具有软引用，那就类似于**可有可无**的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

### 3. 弱引用 (WeakReference)

如果一个对象只具有弱引用，那就类似于**可有可无**的生活用品。弱引用与软引用的区别在于：**只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。**不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

### 4. 虚引用 (PhantomReference)

"虚引用"顾名思义，就是**形同虚设**，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

**虚引用主要用来跟踪对象被垃圾回收的活动。**

**虚引用与软引用和弱引用的一个区别在于：**虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为**软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。**



## 2.4 不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是非死不可的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程；可达性分析法中不可达的对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize 方法。当对象没有覆盖 finalize 方法，或 finalize 方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。

被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。

## 2.5 如何判断一个常量是废弃常量

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的話，"abc" 就会被系统清理出常量池。

注意：JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。

## 2.6 如何判断一个类是无用的类

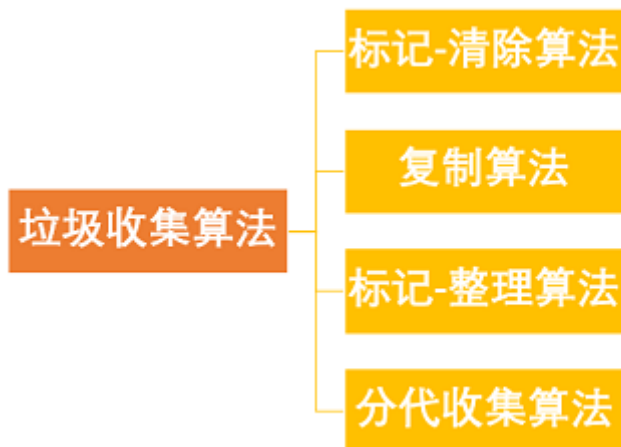
方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

- 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。
- 加载该类的 ClassLoader 已经被回收。
- 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。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

# 3 垃圾收集算法



### 3.1 标记-清除算法

该算法分为“标记”和“清除”阶段：首先标记出所有活动的对象，在标记完成后统一回收所有未被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）

内存整理前


内存整理后


可用内存	可回收内存	存活对象
------	-------	------

### 3.2 复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

内存整理前


内存整理后


可用内存	可回收内存	存活对象	保留内存
------	-------	------	------

3.3 标记-整理算法

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

回收前状态：


回收后状态：


存活对象	可回收	未使用
------	-----	-----

3.4 分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

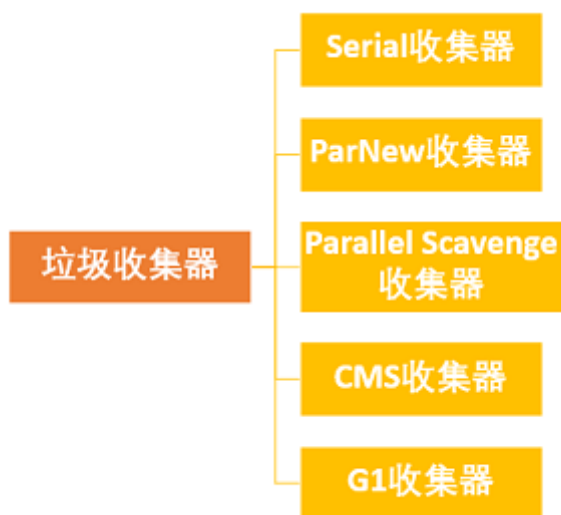
**比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。**

**而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。**

延伸面试问题：HotSpot 为什么要分为新生代和老年代？

根据上面的对分代收集算法的介绍回答。

## 4 垃圾收集器



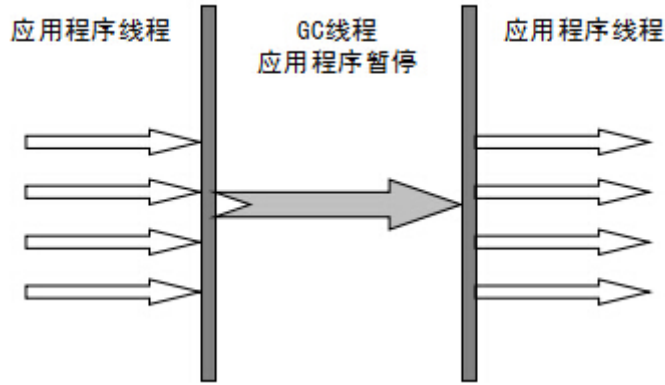
**如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。**

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，**我们能做的就是根据具体应用场景选择适合自己的垃圾收集器**。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 HotSpot 虚拟机就不会实现那么多不同的垃圾收集器了。

### 4.1 Serial 收集器

Serial（串行）收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的“**单线程**”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（"Stop The World"），直到它收集结束。

**新生代采用复制算法，老年代采用标记-整理算法。**



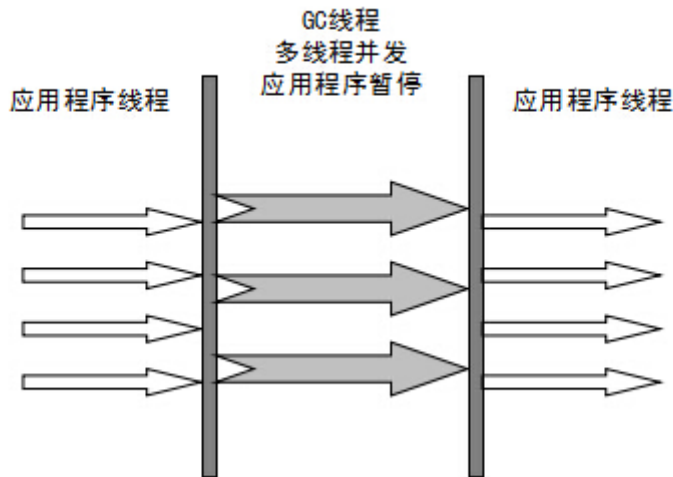
虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是 Serial 收集器有没有优于其他垃圾收集器的地方呢？当然有，它**简单而高效（与其他收集器的单线程相比）**。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错的选择。

## 4.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。

**新生代采用复制算法，老年代采用标记-整理算法。**



它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

**并行和并发概念补充：**

- 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

- 并发 (Concurrent)：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

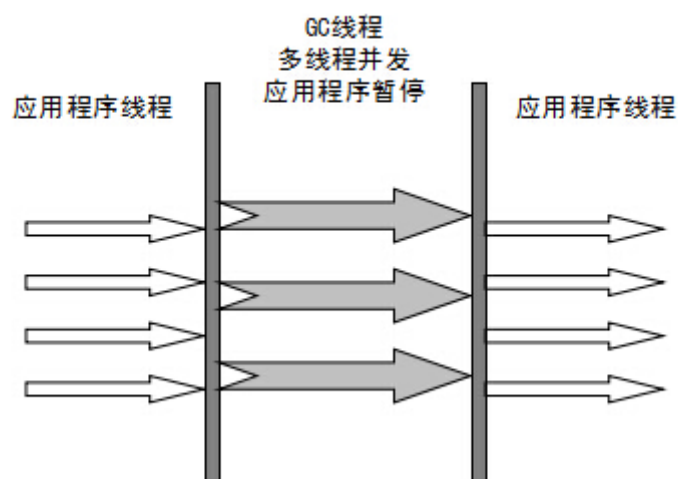
### 4.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是使用复制算法的多线程收集器，它看上去几乎和ParNew都一样。那么它有什么特别之处呢？

```
1  -XX:+UseParallelGC
2
3  使用 Parallel 收集器+ 老年代串行
4
5  -XX:+UseParallelOldGC
6
7  使用 Parallel 收集器+ 老年代并行
```

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。Parallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，手工优化存在困难的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

**新生代采用复制算法，老年代采用标记-整理算法。**



### 4.4.Serial Old 收集器

**Serial 收集器的老年代版本**，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

### 4.5 Parallel Old 收集器



**Parallel Scavenge 收集器的老年代版本。**使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

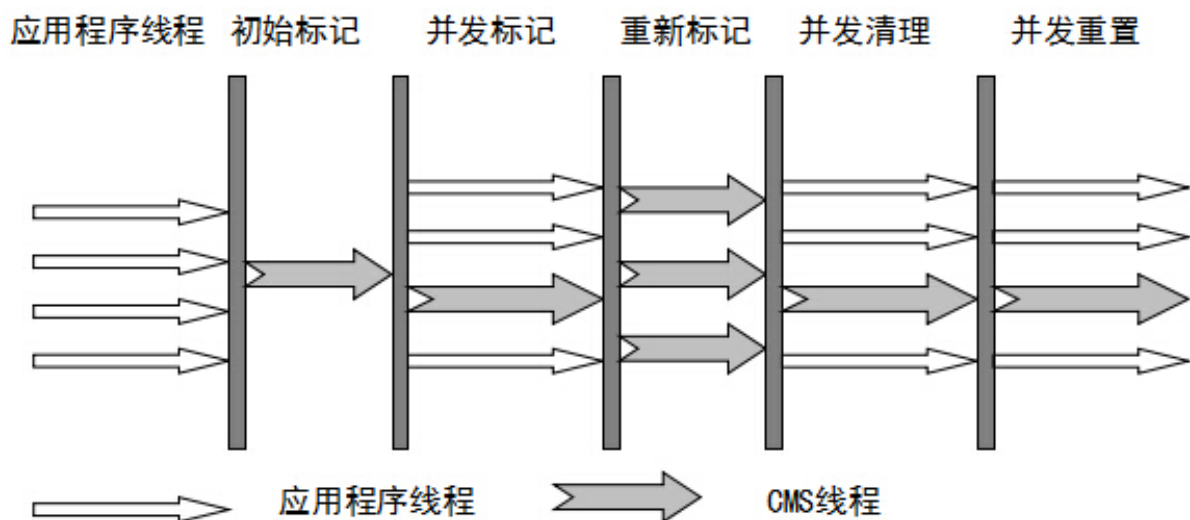
## 4.6 CMS 收集器

**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。它非常符合在注重用户体验的应用上使用。

**CMS (Concurrent Mark Sweep) 收集器**是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的Mark Sweep这两个词可以看出，CMS 收集器是一种“**标记-清除**”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记：** 暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快；
- **并发标记：** 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
- **重新标记：** 重新标记阶段就是为了修正并发标记期间因为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，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- **并发清除：** 开启用户线程，同时 GC 线程开始对未标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

- **对 CPU 资源敏感；**
- **无法处理浮动垃圾；**
- **它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。**

## 4.7 G1 收集器

**G1 (Garbage-First)** 是一款面向服务器的垃圾收集器，主要针对配备多颗处理器及大容量内存的机器。以极高概率满足 GC 停顿时间要求的同时，还具备高吞吐量性能特征。

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

- **并行与并发：**G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿时间。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：**虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- **空间整合：**与 CMS 的“标记--清理”算法不同，G1 从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿：**这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 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 毫秒的时间片段内。
- 

G1 收集器的运作大致分为以下几个步骤：

- **初始标记**
- **并发标记**
- **最终标记**
- **筛选回收**

**G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)。**这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

# JVM的监控与优化

监控过程是优化的前提，优化是监控之后采取的措施。JVM的监控主要介绍相关监控工具、定义监控内容；JVM的优化主要包括内存分配和垃圾回收机制设置。

## 一、JVM的监控

俗话说，“工欲善其事必先利其器”。要对JVM进行监控，首先要选择一款得心应手的监控工具。下面分别介绍JDK自带的监控工具、系统监控工具和商业化的监控工具，相信总有一款是你喜欢的！

### JDK 命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

- **jps** (JVM Process Status)：类似 UNIX 的 ps 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息；
- **jstat** (JVM Statistics Monitoring Tool)：用于收集 HotSpot 虚拟机各方面的运行数据；
- **jinfo** (Configuration Info for Java)：Configuration Info for Java，显示虚拟机配置信息；
- **jmap** (Memory Map for Java)：生成堆转储快照；
- **jhat** (JVM Heap Dump Browser)：用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果；
- **jstack** (Stack Trace for Java)：生成虚拟机当前时刻的线程快照，线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

### jps:查看所有 Java 进程

jps(JVM Process Status) 命令类似 UNIX 的 ps 命令。

jps：显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID (Local Virtual Machine Identifier,LVMID)。jps -q：只输出进程的本地虚拟机唯一 ID。

```
1 C:\Users\SnailClimb>jps
2 7360 NettyClient2
3 17396
4 7972 Launcher
5 16504 Jps
6 17340 NettyServer
```

jps -l:输出主类的全名，如果进程执行的是 Jar 包，输出 Jar 路径。

```
1 C:\Users\SnailClimb>jps -l
```

```
2 7360 firstNettyDemo.NettyClient2
3 17396
4 7972 org.jetbrains.jps.cmdline.Launcher
5 16492 sun.tools.jps.Jps
6 17340 firstNettyDemo.NettyServer
```

**jps -v**: 输出虚拟机进程启动时 JVM 参数。

**jps -m**: 输出传递给 Java 进程 main() 函数的参数。

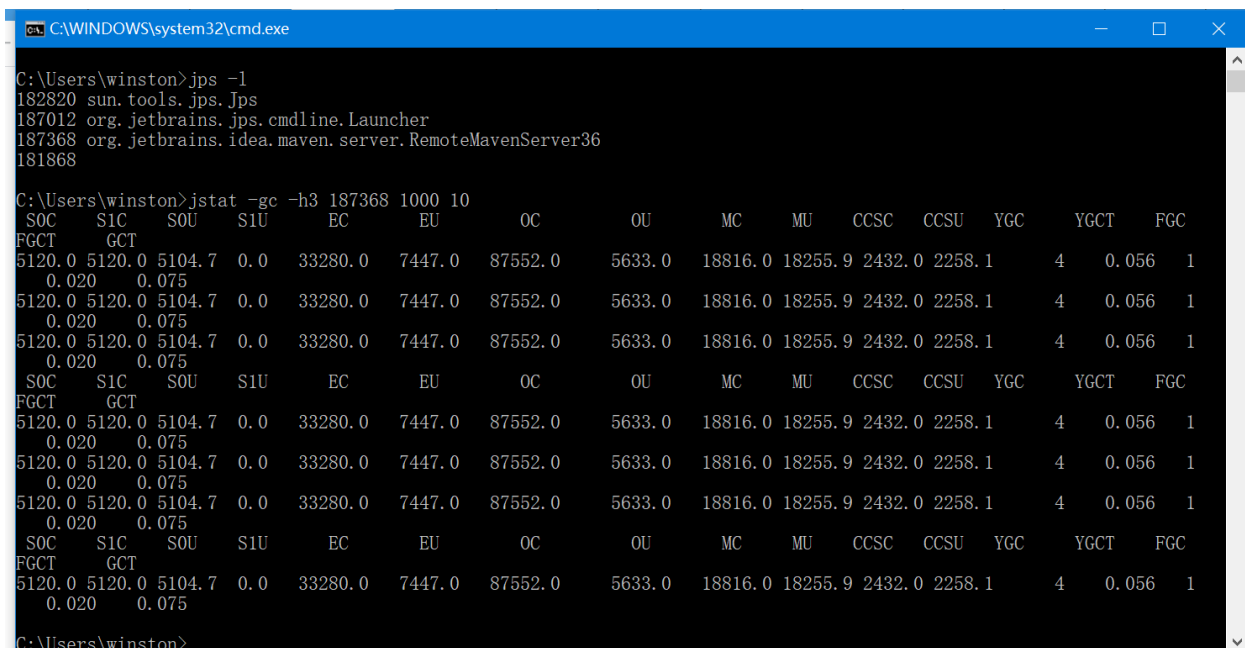
## jstat: 监视虚拟机各种运行状态信息

jstat (JVM Statistics Monitoring Tool) 使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程（需要远程主机提供 RMI 支持）虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。

### jstat 命令使用格式:

```
1 jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

比如 **jstat -gc -h3 187368 1000 10** 表示分析进程 id 为 187368 的 gc 情况，每隔 1000ms 打印一次记录，打印 10 次停止，每 3 行后打印指标头部。



```
C:\WINDOWS\system32\cmd.exe
C:\Users\winston>jps -l
182820 sun.tools.jps.Jps
187012 org.jetbrains.jps.cmdline.Launcher
187368 org.jetbrains.idea.maven.server.RemoteMavenServer36
181868

C:\Users\winston>jstat -gc -h3 187368 1000 10
S0C   S1C   S0U   S1U   EC   EU   OC   OU   MC   MU   CCSC   CCSU   YGC   YGCT   FGC
FGCT   GCT
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
S0C   S1C   S0U   S1U   EC   EU   OC   OU   MC   MU   CCSC   CCSU   YGC   YGCT   FGC
FGCT   GCT
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
S0C   S1C   S0U   S1U   EC   EU   OC   OU   MC   MU   CCSC   CCSU   YGC   YGCT   FGC
FGCT   GCT
5120.0 5120.0 5104.7 0.0   33280.0 7447.0 87552.0 5633.0 18816.0 18255.9 2432.0 2258.1 4 0.056 1
0.020 0.075
```

常见的 option 如下:

- **jstat -class vmid** : 显示 ClassLoader 的相关信息;
- **jstat -compiler vmid** : 显示 JIT 编译的相关信息;
- **jstat -gc vmid** : 显示与 GC 相关的堆信息;
- **jstat -gccapacity vmid** : 显示各个代的容量及使用情况;
- **jstat -gcnew vmid** : 显示新生代信息;
- **jstat -gcnewcapacity vmid** : 显示新生代大小与使用情况;

- `jstat -gcold vmid` : 显示老年代和永久代的信息;
- `jstat -gcoldcapacity vmid` : 显示老年代的大小;
- `jstat -gcpermcapacity vmid` : 显示永久代大小;
- `jstat -gcutil vmid` : 显示垃圾收集信息;

另外, 加上 `-t` 参数可以在输出信息上加一个 Timestamp 列, 显示程序的运行时间。

### **jinfo: 实时地查看和调整虚拟机各项参数**

`jinfo vmid` : 输出当前 jvm 进程的全部参数和系统属性 (第一部分是系统的属性, 第二部分是 JVM 的参数)。

`jinfo -flag name vmid` : 输出对应名称的参数的具体值。比如输出 `MaxHeapSize`、查看当前 jvm 进程是否开启打印 GC 日志 ( `-XX:PrintGCDetails` : 详细 GC 日志模式, 这两个都是默认关闭的)。

```
1 C:\Users\SnailClimb>jinfo -flag MaxHeapSize 17340
2 -XX:MaxHeapSize=2124414976
3 C:\Users\SnailClimb>jinfo -flag PrintGC 17340
4 -XX:-PrintGC
```

使用 `jinfo` 可以在不重启虚拟机的情况下, 可以动态的修改 jvm 的参数。尤其在线上的环境特别有用, 请看下面的例子:

`jinfo -flag [+|-]name vmid` 开启或者关闭对应名称的参数。

```
1 C:\Users\SnailClimb>jinfo -flag PrintGC 17340
2 -XX:-PrintGC
3
4 C:\Users\SnailClimb>jinfo -flag +PrintGC 17340
5
6 C:\Users\SnailClimb>jinfo -flag PrintGC 17340
7 -XX:+PrintGC
```

### **jmap:生成堆转储快照**

`jmap` (Memory Map for Java) 命令用于生成堆转储快照。如果不使用 `jmap` 命令, 要想获取 Java 堆转储, 可以使用 `"-XX:+HeapDumpOnOutOfMemoryError"` 参数, 可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件, Linux 命令下可以通过 `kill -3` 发送进程退出信号也能拿到 dump 文件。

`jmap` 的作用并不仅仅是为了获取 dump 文件, 它还可以查询 finalizer 执行队列、Java 堆和永久代的详细信息, 如空间使用率、当前使用的是哪种收集器等。和 `jinfo` 一样, `jmap` 有不少功能在 Windows 平台下也是受限制的。

示例：将指定应用程序的堆快照输出到桌面。后面，可以通过 jhat、Visual VM 等工具分析该堆文件。

```
1 C:\Users\SnailClimb>jmap -dump:format=b,file=C:\Users\SnailClimb\Desktop\heap.hprof 17340
2 Dumping heap to C:\Users\SnailClimb\Desktop\heap.hprof ...
3 Heap dump file created
```

### jhat: 分析 heapdump 文件

jhat 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果。

```
1 C:\Users\SnailClimb>jhat C:\Users\SnailClimb\Desktop\heap.hprof
2 Reading from C:\Users\SnailClimb\Desktop\heap.hprof...
3 Dump file created Sat May 04 12:30:31 CST 2019
4 Snapshot read, resolving...
5 Resolving 131419 objects...
6 Chasing references, expect 26 dots.....
7 Eliminating duplicate references.....
8 Snapshot resolved.
9 Started HTTP server on port 7000
10 Server is ready.
```

访问 <http://localhost:7000/>

### jstack :生成虚拟机当前时刻的线程快照

jstack (Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

生成线程快照的目的主要是定位线程长时间出现停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

下面是一个线程死锁的代码。我们下面会通过 jstack 命令进行死锁检查，输出死锁信息，找到发生死锁的线程。

```
1 public class DeadLockDemo {
2     private static Object resource1 = new Object();//资源 1
3     private static Object resource2 = new Object();//资源 2
4
5     public static void main(String[] args) {
```



```

6  new Thread(() -> {
7      synchronized (resource1) {
8          System.out.println(Thread.currentThread() + "get resource1");
9          try {
10             Thread.sleep(1000);
11         } catch (InterruptedException e) {
12             e.printStackTrace();
13         }
14         System.out.println(Thread.currentThread() + "waiting get resource2");
15         synchronized (resource2) {
16             System.out.println(Thread.currentThread() + "get resource2");
17         }
18     }
19 }, "线程 1").start();
20
21 new Thread(() -> {
22     synchronized (resource2) {
23         System.out.println(Thread.currentThread() + "get resource2");
24     } try {
25         Thread.sleep(1000);
26     } catch (InterruptedException e) {
27         e.printStackTrace();
28     }
29     System.out.println(Thread.currentThread() + "waiting get resource1");
30     synchronized (resource1) {
31         System.out.println(Thread.currentThread() + "get resource1");
32     }
33 }
34 }, "线程 2").start();
35 }
36 }

```

## Output

```

1 Thread[线程 1,5,main]get resource1
2 Thread[线程 2,5,main]get resource2
3 Thread[线程 1,5,main]waiting get resource2
4 Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 `synchronized (resource1)` 获得 `resource1` 的监视器锁，然后通过 `Thread.sleep(1000)` 让线程 A 休眠 1s 为的是让线程 B 得到执行，然后获取到

resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

通过 jstack 命令分析：

```
1 C:\Users\SnailClimb>jps
2 13792 KotlinCompileDaemon
3 7360 NettyClient2
4 17396
5 7972 Launcher
6 8932 Launcher
7 9256 DeadLockDemo
8 10764 Jps
9 17340 NettyServer
10
11 C:\Users\SnailClimb>jstack 9256
```

输出的部分内容如下：

```
1 Found one Java-level deadlock:
2 =====
3 "线程 2":
4   waiting to lock monitor 0x00000000333e668 (object 0x00000000d5efe1c0, a
   java.lang.Object),
5   which is held by "线程 1"
6 "线程 1":
7   waiting to lock monitor 0x00000000333be88 (object 0x00000000d5efe1d0, a
   java.lang.Object),
8   which is held by "线程 2"
9
10 Java stack information for the threads listed above:
11 =====
12 "线程 2":
13   at DeadLockDemo.lambda$main$1(DeadLockDemo.java:31)
14   - waiting to lock <0x00000000d5efe1c0> (a java.lang.Object)
15   - locked <0x00000000d5efe1d0> (a java.lang.Object)
16   at DeadLockDemo$$Lambda$2/1078694789.run(Unknown Source)
17   at java.lang.Thread.run(Thread.java:748)
18 "线程 1":
19   at DeadLockDemo.lambda$main$0(DeadLockDemo.java:16)
20   - waiting to lock <0x00000000d5efe1d0> (a java.lang.Object)
21   - locked <0x00000000d5efe1c0> (a java.lang.Object)
22   at DeadLockDemo$$Lambda$1/1324119927.run(Unknown Source)
```

```
23 at java.lang.Thread.run(Thread.java:748)
24
25 Found 1 deadlock.
```

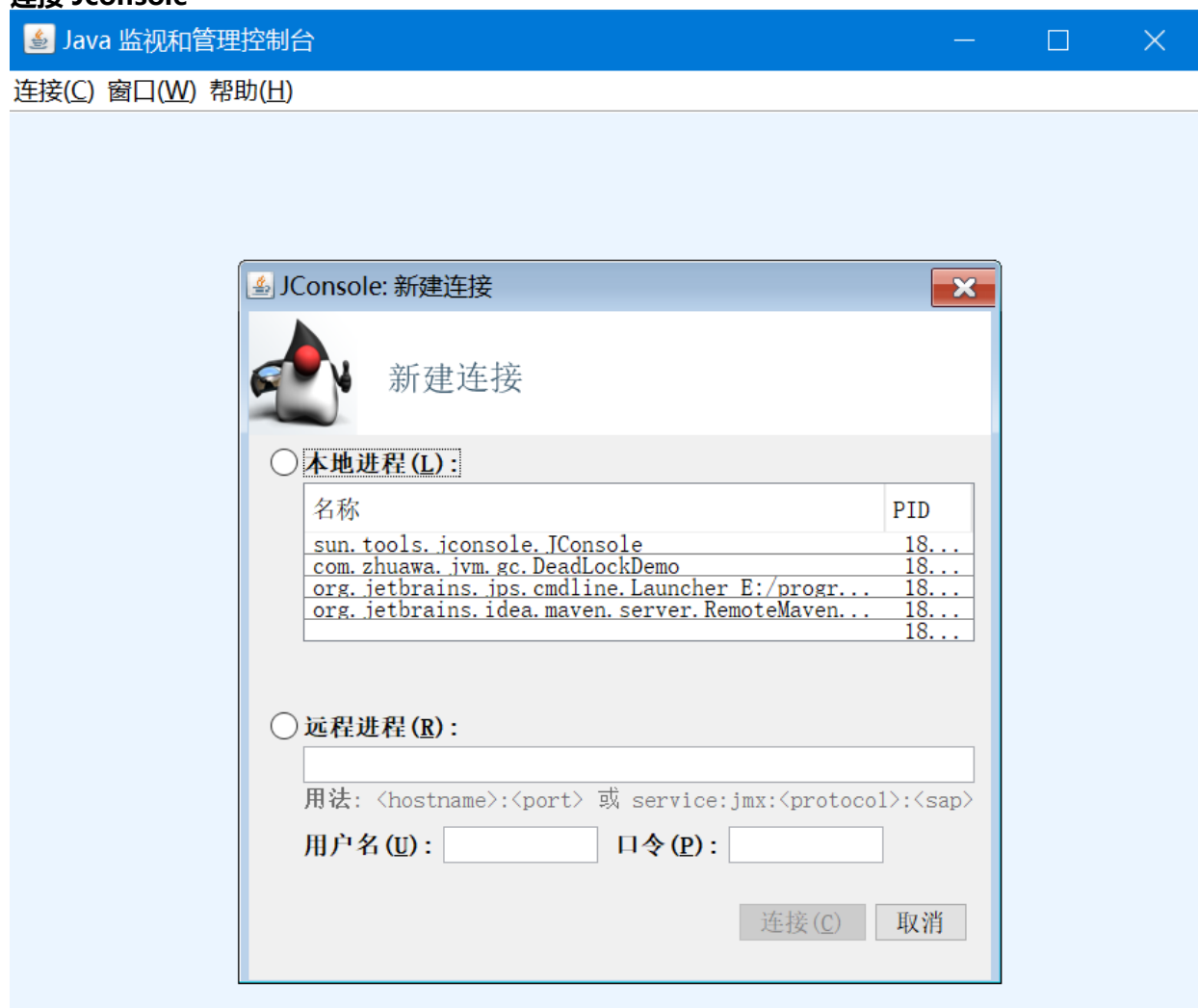
可以看到 jstack 命令已经帮我们找到发生死锁的线程的具体信息。

## JDK 可视化分析工具

### JConsole:Java 监视与管理控制台

JConsole 是基于 JMX 的可视化监视、管理工具。可以很方便的监视本地及远程服务器的 java 进程的内存使用情况。你可以在控制台输出 console 命令启动或者在 JDK 目录下的 bin 目录找到 jconsole.exe 然后双击启动。

### 连接 Jconsole



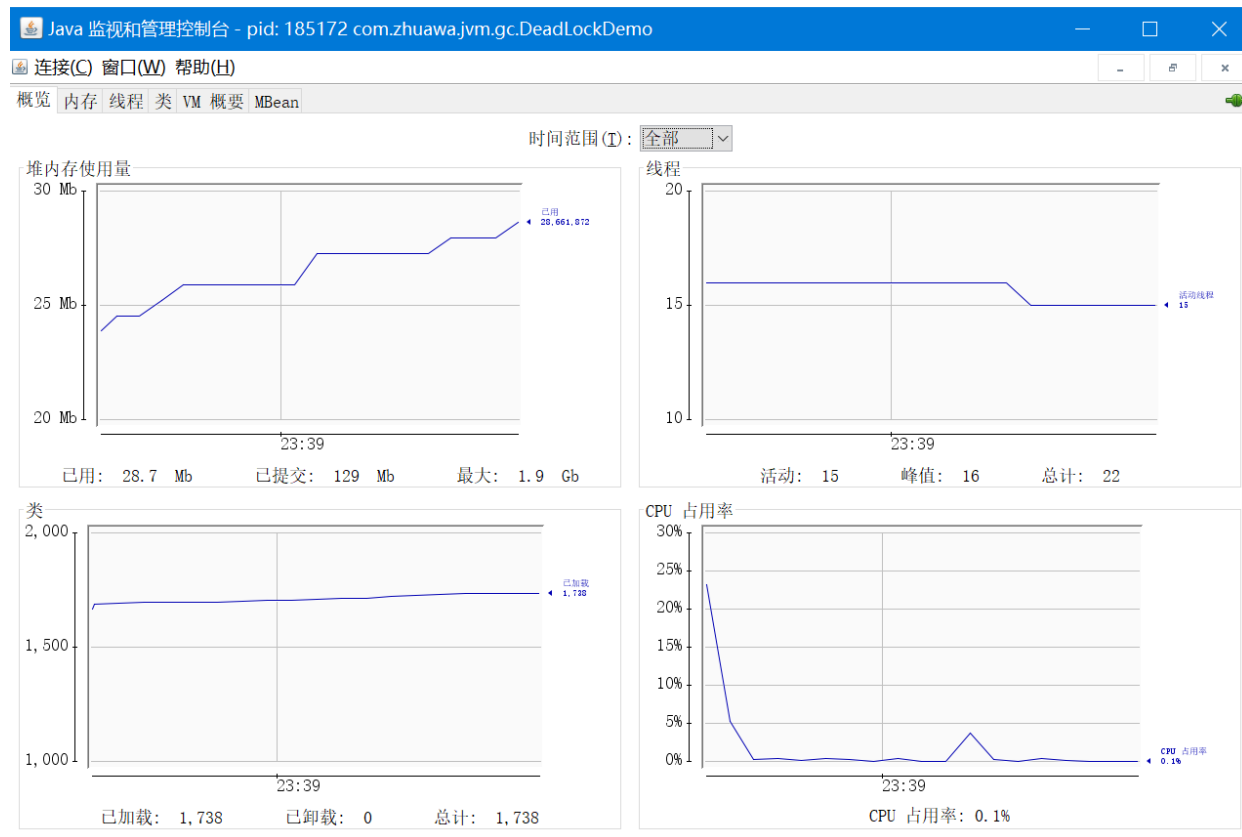
如果需要使用 JConsole 连接远程进程，可以在远程 Java 程序启动时加上下面这些参数：

- 1 -Djava.rmi.server.hostname=外网访问 ip 地址
- 2 -Dcom.sun.management.jmxremote.port=60001 //监控的端口号
- 3 -Dcom.sun.management.jmxremote.authenticate=false //关闭认证
- 4 -Dcom.sun.management.jmxremote.ssl=false

在使用 JConsole 连接时，远程进程地址如下：

1 外网访问 ip 地址:60001

## 查看 Java 程序概况

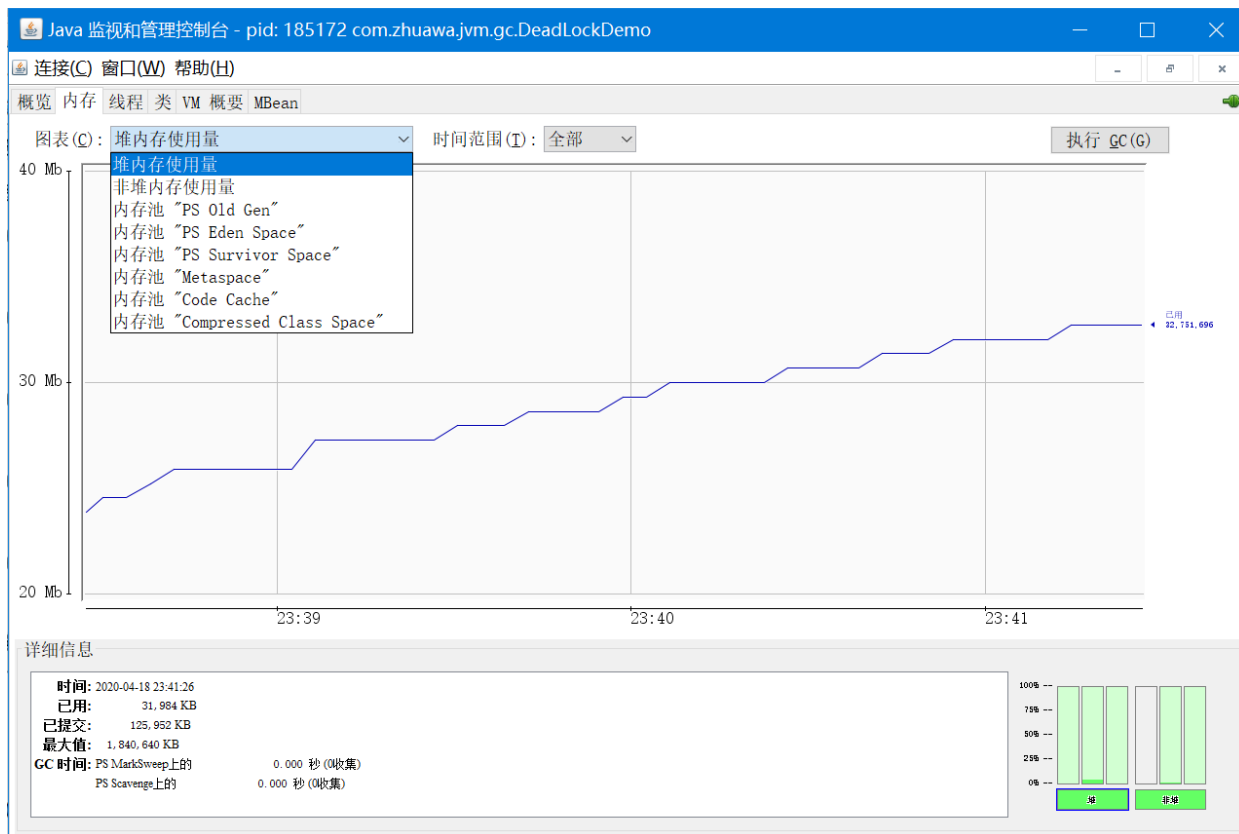


## 内存监控

JConsole 可以显示当前内存的详细信息。不仅包括堆内存/非堆内存的整体信息，还可以细化到 eden 区、survivor 区等的使用情况，如下图所示。

点击右边的“执行 GC(G)”按钮可以强制应用程序执行一个 Full GC。

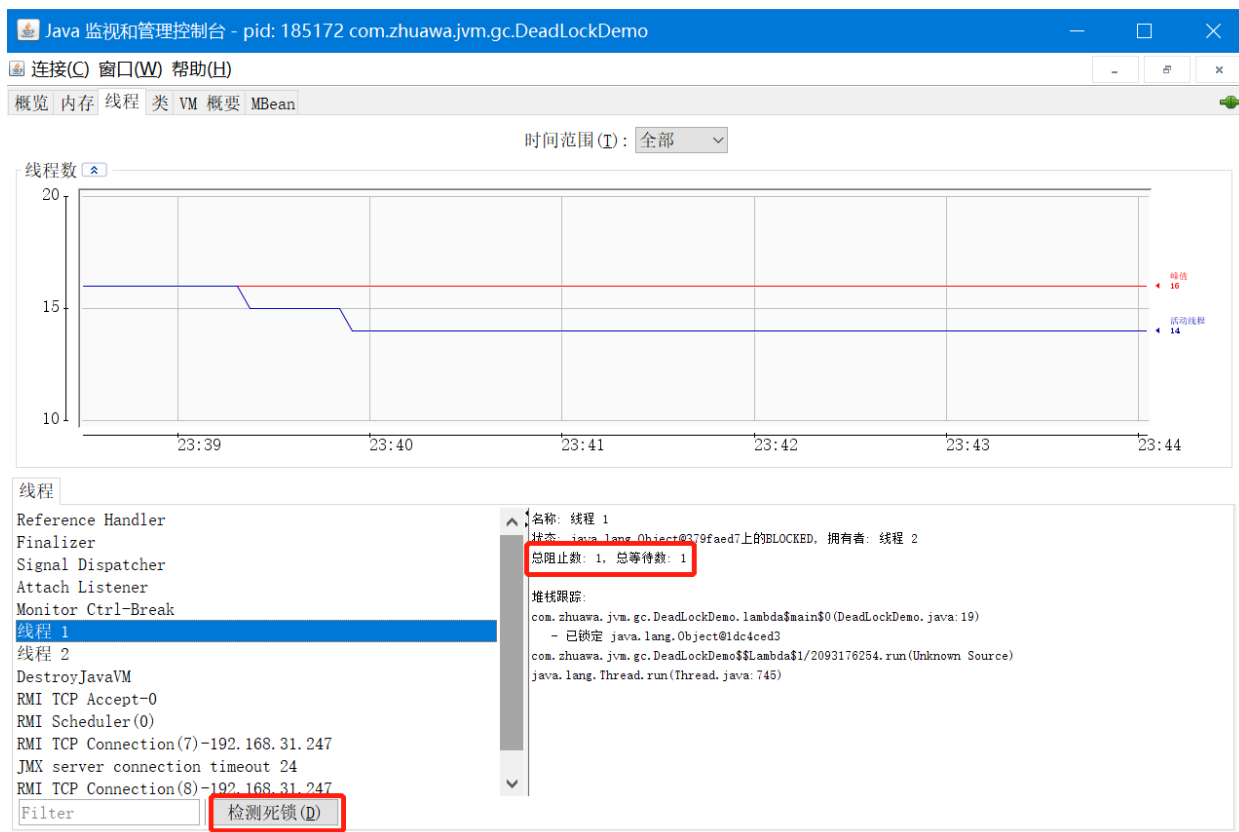
- 新生代 GC (Minor GC) :指发生新生代的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- 老年代 GC (Major GC/Full GC) :指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC (并非绝对)，Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。



## 线程监控

类似我们前面讲的 `jstack` 命令，不过这个是可视化的。

最下面有一个"检测死锁 (D)"按钮，点击这个按钮可以自动为你找到发生死锁的线程以及它们的详细信息。



## Visual VM:多合一故障处理工具

VisualVM 提供在 Java 虚拟机 (Java Virtual Machine, JVM) 上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中, 可以方便、快捷地查看多个 Java 应用程序的相关信息。Visual VM 官网: <https://visualvm.github.io/>。Visual VM 中文文档:<https://visualvm.github.io/documentation.html>。

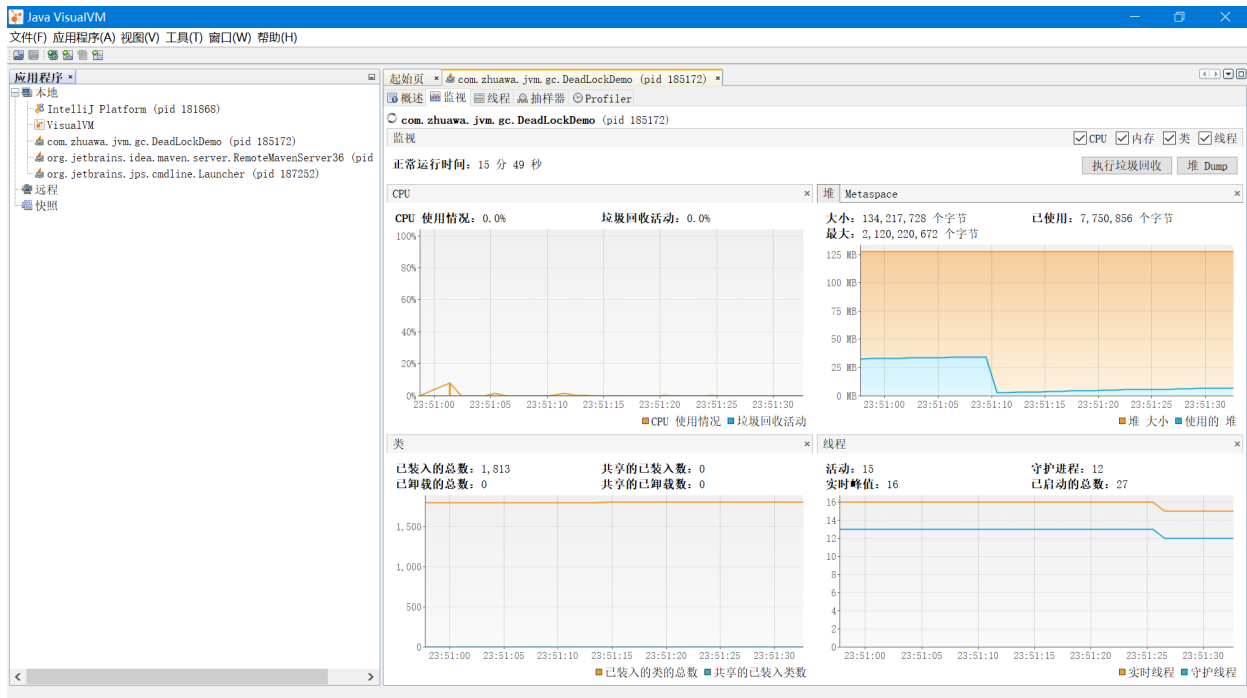
下面这段话摘自《深入理解 Java 虚拟机》。

VisualVM (All-in-One Java Troubleshooting Tool) 是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序, 官方在 VisualVM 的软件说明中写上了 “All-in-One” 的描述字样, 预示着他除了运行监视、故障处理外, 还提供了很多其他方面的功能, 如性能分析 (Profiling)。VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少, 而且 VisualVM 还有一个很大的优点: 不需要被监视的程序基于特殊 Agent 运行, 因此他对应用程序的实际性能的影响很小, 使得他可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

VisualVM 基于 NetBeans 平台开发, 因此它一开始就具备了插件扩展功能的特性, 通过插件扩展支持, VisualVM 可以做到:

- 显示虚拟机进程以及进程的配置、环境信息 (jps、jinfo)。
- 监视应用程序的 CPU、GC、堆、方法区以及线程的信息 (jstat、jstack)。
- dump 以及分析堆转储快照 (jmap、jhat)。
- 方法级的程序运行性能分析, 找到被调用最多、运行时间最长的方法。
- 离线程序快照: 收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照, 可以将快照发送开发者处进行 Bug 反馈。
- 其他 plugins 的无限的可能性.....





这里就不具体介绍 VisualVM 的使用，如果想了解的话可以看：

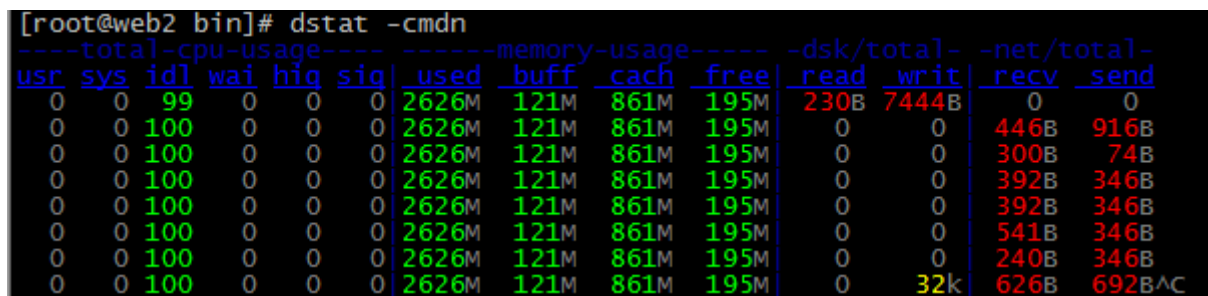
- <https://visualvm.github.io/documentation.html>
- <https://www.ibm.com/developerworks/cn/java/j-lo-visualvm/index.html>

## JVM监控内容

工具选定之后，就要确定监控对象。我们知道，性能无非就是这几种：CPU、内存、磁盘 IO、网络。如果程序性能异常，监控这几个参数一般都可以发现问题。

### CPU

CPU的监控可以采用top、dstat、jconsole或visualvm等工具。下面使用dstat工具监控：



上图第一部分total-cpu-usage是cpu的使用信息。usr表示用户程序占用cpu比例，sys表示系统用户占用cpu比例，idl表示cpu空闲比例。

### 内存

内存除了观察使用率之外，还要关注GC情况。这些信息可以使用jmap和jstat两个命令获得。如果发现内存使用量不断增加，而且gc频率很高，这时需要谨慎查找程序中的内存增长点，可能存在内存泄露问题，可以采用visualvm工具辅助查找无法回收的对象。

### 磁盘IO

磁盘使用情况可以使用dstat (<https://man.linuxde.net/dstat>) 命令获得。

## 网络

网络使用情况可以使用dstat命令获得。

## 二、JVM的优化

### 1、内存分配（可以通过JVM参数：-Xms 和 -Xmx 指定）

#### 堆的分配参数：

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-bit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位操作系统虽然寻址空间大小是4G（ $2^{32}$ ），但具体操作系统会给一个限制（一般Windows系统有2GB内核空间，故用户空间限制在1.5G~2G，Linux系统有1GB内核空间，故用户空间是2G~3G）；64为操作系统对内存无限制。**所有线程共享数据区大小=新生代大小 + 年老代大小 + 持久代大小**。持久代一般固定大小为64m。所以java堆中增大年轻代后，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**java堆的3/8**。

#### 1、-Xmx -Xms：指定java堆最大值（默认值是物理内存的1/4(<1GB)) 和初始java堆最小值（默认值是物理内存的1/64(<1GB))

默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制.，默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制。开发过程中，通常会将 -Xms 与 -Xmx两个参数的配置相同的值，其目的是为了能够在java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小而浪费资源。

注意：此处设置的是Java堆大小，也就是新生代大小 + 年老代大小

#### 2、-Xmn、-XX:NewRatio、-XX:SurvivorRatio、-XXNewSize、-XX:MaxNewSize:

- -Xmn

**设置新生代大小，大小是：eden+ 2 survivor space**

所有共享数据区大小=年轻代大小 + 年老代大小 + 持久代大小。一般永久代大小固定，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java堆的3/8**

- -XX:NewRatio

新生代（eden+2\*Survivor）和老年代（不包含永久区）的比值

例如：-XX:NewRatio=4，表示新生代:老年代=1:4，即新生代占整个堆的1/5。在Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。

- -XX:SurvivorRatio（幸存代）

设置两个Survivor区和eden的比值

例如：8，表示两个Survivor:eden=2:8，即一个Survivor占年轻代的1/10

- `-XX:NewSize`

设置年轻代大小

- `-XX:MaxNewSize`

设置年轻代最大值

### 3、`-XX:+HeapDumpOnOutOfMemoryError`、`-XX:+HeapDumpPath`

- `-XX:+HeapDumpOnOutOfMemoryError`

OOM时导出堆到文件

根据这个文件，我们可以看到系统dump时发生了什么。

- `-XX:+HeapDumpPath`

导出OOM的路径

例如我们设置如下的参数：

```
-Xmx20m -Xms5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump
```

上方意思是说，现在给堆内存最多分配20M的空间。如果发生了OOM异常，那就把dump信息导出到d:/a.dump文件中。

### 4、`-XX:OnOutOfMemoryError`：

- `-XX:OnOutOfMemoryError`

在OOM时，执行一个脚本。

可以在OOM时，发送邮件，甚至是重启程序。

例如我们设置如下的参数：

```
-XX:OnOutOfMemoryError=D:/tools/jdk1.7_40/bin/printstack.bat %p //p代表的是当前进程的pid
```

上方参数的意思是说，执行printstack.bat脚本，而这个脚本做的事情是：

D:/tools/jdk1.7\_40/bin/jstack -F %1 > D:/a.txt，即当程序OOM时，在D:/a.txt中将会生成线程的dump。

### 5、堆的分配参数总结：

- 根据实际事情调整新生代和幸存代的大小
- 官方推荐新生代占java堆的3/8
- 幸存代占新生代的1/10
- 在OOM时，记得Dump出堆，确保可以排查现场问题

### 6、永久区分配参数：

- `-XX:PermSize` `-XX:MaxPermSize`

设置永久区的初始空间（默认为物理内存的1/64）和最大空间（默认为物理内存的1/4）。也就是说，jvm启动时，永久区一开始就占用了PermSize大小的空间，如果空间还不够，可以继续扩展，但是不能超过MaxPermSize，否则会OOM。

他们表示，一个系统可以容纳多少个类型

## 栈的分配参数：

### 1、-Xss:

设置每个线程栈空间的大小。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。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

决定了函数调用的深度

每个线程都有独立的栈空间

局部变量、参数 分配在栈上

注：栈空间是每个线程私有的区域。栈里面的主要内容是栈帧，而栈帧存放的是局部变量表，局部变量表的内容是：局部变量、参数。

### 2、-XXThreadStackSize:

设置线程栈的大小(0 means use default stack size)

## JVM其他参数：

### 1、-XXThreadStackSize:

设置内存页的大小，不可设置过大，会影响Perm的大小

### 2、-XX:+UseFastAccessorMethods:

设置原始类型的快速优化

### 3、-XX:+DisableExplicitGC:

设置关闭System.gc()(这个参数需要严格的测试)

### 4、-XX:MaxTenuringThreshold

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率。该参数只有在串行GC时才有效。

### 5、-XX:+AggressiveOpts

加快编译

### 6、-XX:+UseBiasedLocking

锁机制的性能改善

### 7、-Xnoclassgc

禁用垃圾回收

### 8、-XX:SoftRefLRUPolicyMSPerMB

设置每兆堆空闲空间中SoftReference的存活时间，默认值是1s。（softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap）

#### 9、-XX:PretenureSizeThreshold

设置对象超过多大时直接在旧生代分配，默认值是0。

#### 10、-XX:TLABWasteTargetPercent

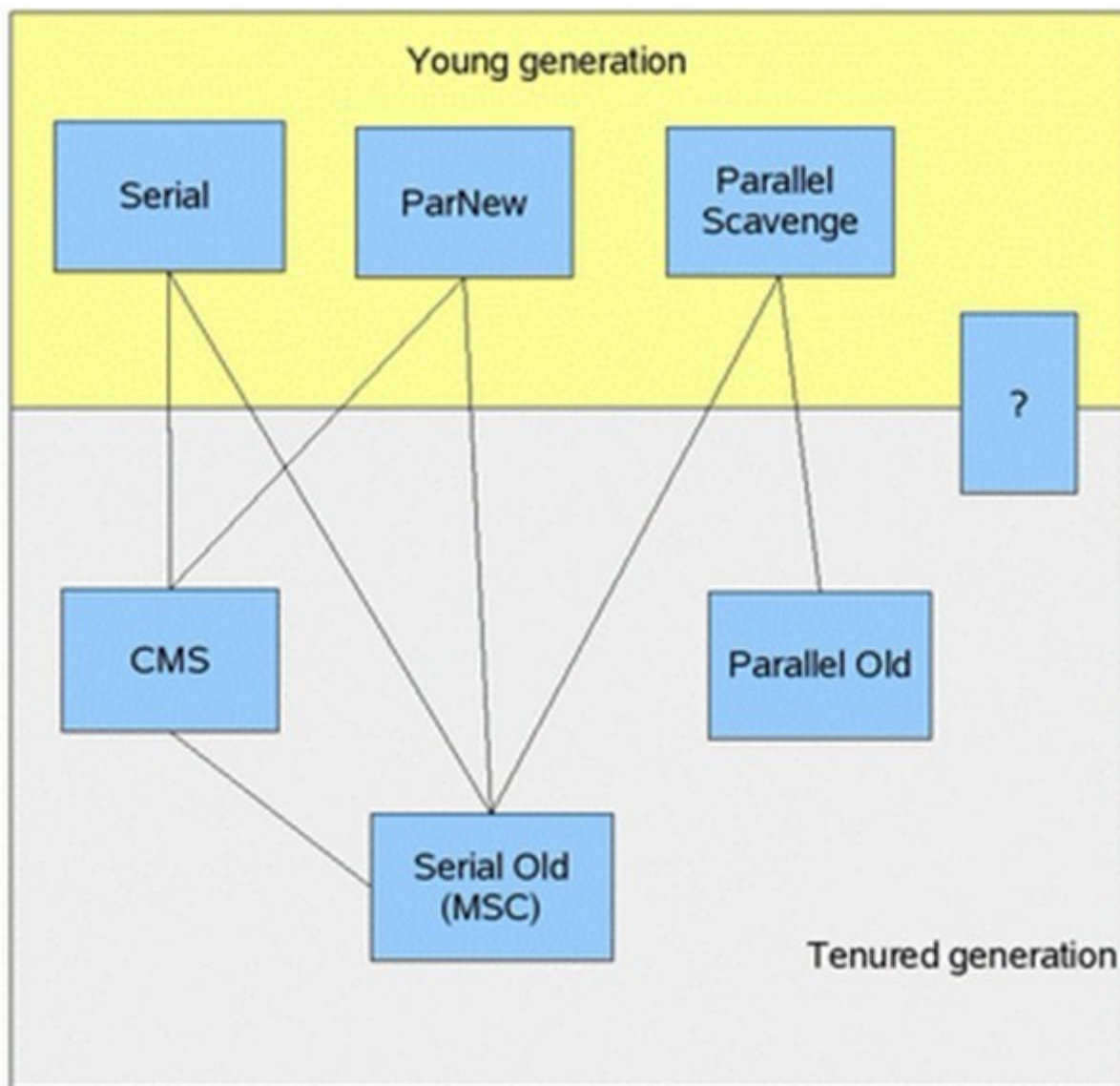
设置TLAB占eden区的百分比，默认值是1%。

#### 11、-XX:+CollectGen0First

设置FullGC时是否先YGC，默认值是false。

## 2、垃圾回收机制

JVM内存采用分代形式管理，则相应的内存垃圾回收机制也采用分代管理，不同的内存区采用不同的回收机制。根据垃圾回收运行方式分为三类：串行GC、并行GC和并发GC。





- 新生代
  - Serial: 串行, 单线程回收垃圾
  - Parallel Scavenge: 并行回收, 多线程回收垃圾
  - ParNew: 并行, 与年老代的CMS配合使用
- 老生代
  - Serial MSC: 串行
  - Parallel MSC: 并行
  - CMS: 并发

#### 垃圾回收机制的指定方式:

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智能判断。

- client, server模式默认GC策略

	新生代	年老代和持久代
client	串行GC	串行GC
server	并行回收GC	Parallel Mark Sweep GC

- GC组合方式



	新生代GC	旧生代和持久代GC
-XX:+UseSerialGC	串行GC	串行GC
-XX:+UseParallelGC	并行回收GC	Parallel Mark Sweep GC
-XX:+UseConcMarkSweepGC	并行GC	并发GC  当出现Concurrent Mode Failure时 采用串行GC
-XX:+UseParNewGC	并行GC	串行GC
-XX:+UseParallelOldGC	并行回收GC	Parallel Mark Compact
-XX:+UseConcMarkSweepGC -XX:-UseParNewGC	串行GC	并发GC  当出现Concurrent Mode Failure 或Promotion Failed时采用串行GC
不支持的组合方式	1. -XX:+UseParNewGC -XX:+UseParallelOldGC  2. -XX:+UseParNewGC -XX:+UseSerialGC	

## Tomcat优化配置

### 1、内存优化：

优化内存，主要是在bin/catalina.bat/sh 配置文件中进行。linux上，在catalina.sh中添加：

```
1 JAVA_OPTS="-server -Xms1G -Xmx2G -Xss256K -Djava.awt.headless=true -Dfile.encoding=utf-8 -XX:MaxPermSize=256m -XX:PermSize=128M -XX:MaxPermSize=256M"
```

其中：

- 1 • **-server**: 启用jdk的server版本。
- 2 • **-Xms**: 虚拟机初始化时的最小堆内存。
- 3 • **-Xmx**: 虚拟机可使用的最大堆内存。 #**-Xms**与**-Xmx**设成一样的值, 避免JVM因为频繁的GC导致性能大起大落
- 4 • **-XX:PermSize**: 设置非堆内存初始值, 默认是物理内存的1/64。
- 5 • **-XX:MaxNewSize**: 新生代占整个堆内存的最大值。
- 6 • **-XX:MaxPermSize**: Perm (俗称方法区) 占整个堆内存的最大值, 也称内存最大永久保留区域。

### 1) 错误提示: java.lang.OutOfMemoryError:Java heap space

Tomcat默认可以使用的内存为128MB, 在较大型的应用项目中, 这点内存是不够的, 有可能导致系统无法运行。常见的问题是报Tomcat内存溢出错误, Outof Memory(系统内存不足)的异常, 从而导致客户端显示500错误, 一般调整Tomcat的-Xms和-Xmx即可解决问题, 通常将-Xms和-Xmx设置成一样, 堆的最大值设置为物理可用内存的最大值的80%。

```
1 set JAVA_OPTS=-Xms512m-Xmx512m
```

### 2) 错误提示: java.lang.OutOfMemoryError: PermGenspace

PermGenspace的全称是Permanent Generationspace,是指内存的永久保存区域, 这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGenspace中, 它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGenspace进行清理, 所以如果你的应用中有很CLASS的话,就很可能出现PermGen space错误, 这种错误常见在web服务器对JSP进行precompile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。解决方法:

```
1 set JAVA_OPTS=-XX:PermSize=128M
```

### 3) 在使用-Xms和-Xmx调整tomcat的堆大小时, 还需要考虑垃圾回收机制。

如果系统花费很多的时间收集垃圾, 请减小堆大小。一次完全的垃圾收集应该不超过3-5 秒。如果垃圾收集成为瓶颈, 那么需要指定代的大小, 检查垃圾收集的详细输出, 研究垃圾收集参数对性能的影响。一般说来, 你应该使用物理内存的80% 作为堆大小。当增加处理器时, 记得增加内存, 因为分配可以并行进行, 而垃圾收集不是并行的。

## 2、连接数优化:

优化连接数, 主要是在conf/server.xml配置文件中进行修改。

## 2.1、优化线程数

找到Connectorport="8080" protocol="HTTP/1.1", 增加maxThreads和acceptCount属性（使acceptCount大于等于maxThreads），如下：

```
1 <Connectorport="8080" protocol="HTTP/1.1"connectionTimeout="20000" redirectPort="8443"acceptCount="500" maxThreads="400" />
```

其中：

- 1 • maxThreads: tomcat可用于请求处理的最大线程数，默认是200
- 2 • minSpareThreads: tomcat初始线程数，即最小空闲线程数
- 3 • maxSpareThreads: tomcat最大空闲线程数，超过的会被关闭
- 4 • acceptCount: 当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。默认100

## 2.2、使用线程池

在server.xml中增加executor节点，然后配置connector的executor属性，如下：

```
1 <Executorname="tomcatThreadPool" namePrefix="req-exec-"maxThreads="1000" minSpareThreads="50"maxIdleTime="60000"/>
2 <Connectorport="8080" protocol="HTTP/1.1"executor="tomcatThreadPool"/>
```

其中：

- 1 • namePrefix: 线程池中线程的命名前缀
- 2 • maxThreads: 线程池的最大线程数
- 3 • minSpareThreads: 线程池的最小空闲线程数
- 4 • maxIdleTime: 超过最小空闲线程数时，多的线程会等待这个时间长度，然后关闭
- 5 • threadPriority: 线程优先级

注：当tomcat并发用户量大的时候，单个jvm进程确实可能打开过多的文件句柄，这时会报java.net.SocketException:Too many open files错误。可使用下面步骤检查：

- 1 • ps -ef |grep tomcat 查看tomcat的进程ID，记录ID号，假设进程ID为10001
- 2 • lsof -p 10001|wc -l 查看当前进程id为10001的 文件操作数
- 3 • 使用命令: ulimit -a 查看每个用户允许打开的最大文件数

## 3、Tomcat Connector三种运行模式（BIO, NIO, APR）

### 3.1、三种模式比较：

1) BIO：一个线程处理一个请求。缺点：并发量高时，线程数较多，浪费资源。Tomcat7或以下在Linux系统中默认使用这种方式。

2) NIO: 利用Java的异步IO处理, 可以通过少量的线程处理大量的请求。Tomcat8在Linux系统中默认使用这种方式。Tomcat7必须修改Connector配置来启动 (conf/server.xml配置文件) :

```
1 <Connectorport="8080"protocol="org.apache.coyote.http11.Http11NioProtocol"
1" connectionTimeout="20000"redirectPort="8443"/>
```

3) APR(Apache Portable Runtime): 从操作系统层面解决io阻塞问题。Linux如果安装了apr和native, Tomcat直接启动就支持apr。

### 3.2、apr模式

安装apr以及tomcat-native

```
1 yum -y install apr apr-devel
```

进入tomcat/bin目录, 比如:

```
1 cd /opt/local/tomcat/bin/
2 tar xzfv tomcat-native.tar.gz
3 cd tomcat-native-1.1.32-src/jni/native
4 ./configure --with-apr=/usr/bin/apr-1-config
5 make && make install
```

#注意最新版本的tomcat自带tomcat-native.war.gz, 不过其版本相对于yum安装的apr过高, configure的时候会报错。

解决: yum remove apr apr-devel -y, 卸载yum安装的apr和apr-devel, 下载最新版本的apr源码包, 编译安装; 或者下载低版本的tomcat-native编译安装。安装成功后还需要对tomcat设置环境变量, 方法是在catalina.sh文件中增加1行:

```
1 CATALINA_OPTS="-Djava.library.path=/usr/local/apr/lib"
```

#apr下载地址: <http://apr.apache.org/download.cgi>

#tomcat-native下载地址: <http://tomcat.apache.org/download-native.cgi>

修改8080端对应的conf/server.xml

protocol="org.apache.coyote.http11.Http11AprProtocol"

```
1 <Connector executor="tomcatThreadPool"
2 port="8080"
3 protocol="org.apache.coyote.http11.Http11AprProtocol"
4 connectionTimeout="20000"
5 enableLookups="false"
6 redirectPort="8443"/>
```

```
7  URLEncoder="UTF-8" />
```

PS:启动以后查看日志 显示如下表示开启 apr 模式

```
1  Sep 19, 2016 3:46:21 PM org.apache.coyote.AbstractProtocol start
2  INFO: Starting ProtocolHandler ["http-apr-8081"]
```