# 什么是HotSpot VM

提起HotSpot VM，相信所有Java程序员都知道，它是Sun JDK和OpenJDK中所带的虚拟机，也是目前使用范围最广的Java虚拟机。

但不一定所有人都知道的是，这个目前看起来“血统纯正”的虚拟机在最初并非由Sun公司开发，而是由一家名为“Longview Technologies”的小公司设计的；

甚至这个虚拟机最初并非是为Java语言而开发的，它来源于Strongtalk VM，

而这款虚拟机中相当多的技术又是来源于一款支持Self语言实现“达到C语言50%以上的执行效率”的目标而设计的虚拟机，

Sun公司注意到了这款虚拟机在JIT编译上有许多优秀的理念和实际效果，在1997年收购了Longview Technologies公司，从而获得了HotSpot VM。

HotSpot VM既继承了Sun之前两款商用虚拟机的优点（如前面提到的准确式内存管理），也有许多自己新的技术优势，

如它名称中的HotSpot指的就是它的热点代码探测技术（其实两个VM基本上是同时期的独立产品，HotSpot还稍早一些，HotSpot一开始就是准确式GC，

而Exact VM之中也有与HotSpot几乎一样的热点探测。

为了Exact VM和HotSpot VM哪个成为Sun主要支持的VM产品，在Sun公司内部还有过争论，HotSpot打败Exact并不能算技术上的胜利），

HotSpot VM的热点代码探测能力可以通过执行计数器找出最具有编译价值的代码，然后通知JIT编译器以方法为单位进行编译。

如果一个方法被频繁调用，或方法中有效循环次数很多，将会分别触发标准编译和OSR（栈上替换）编译动作。

通过编译器与解释器恰当地协同工作，可以在最优化的程序响应时间与最佳执行性能中取得平衡，而且无须等待本地代码输出才能执行程序，

即时编译的时间压力也相对减小，这样有助于引入更多的代码优化技术，输出质量更高的本地代码。

在2006年的JavaOne大会上，Sun公司宣布最终会把Java开源，并在随后的一年，陆续将JDK的各个部分（其中当然也包括了HotSpot VM）在GPL协议下公开了源码，

并在此基础上建立了OpenJDK。这样，HotSpot VM便成为了Sun JDK和OpenJDK两个实现极度接近的JDK项目的共同虚拟机。

在2008年和2009年，Oracle公司分别收购了BEA公司和Sun公司，这样Oracle就同时拥有了两款优秀的Java虚拟机：JRockit VM和HotSpot VM。

Oracle公司宣布在不久的将来（大约应在发布JDK 8的时候）会完成这两款虚拟机的整合工作，使之优势互补。

整合的方式大致上是在HotSpot的基础上，移植JRockit的优秀特性，譬如使用JRockit的垃圾回收器与MissionControl服务，

使用HotSpot的JIT编译器与混合的运行时系统。

# Java内存结构



## Java堆（Java Heap）

　java堆是java虚拟机所管理的内存中最大的一块，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，这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。

java堆是垃圾收集器管理的主要区域，因此也被成为“GC堆”（Garbage Collected Heap）。从内存回收角度来看java堆可分为：新生代和老生代（当然还有更细致的划分，在下一章会讲到）。从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。无论怎么划分，都与存放内容无关，无论哪个区域，存储的都是对象实例，进一步的划分都是为了更好的回收内存，或者更快的分配内存。

根据Java虚拟机规范的规定，java堆可以处于物理上不连续的内存空间中。当前主流的虚拟机都是可扩展的（通过 -Xmx 和 -Xms 控制）。如果堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

## Java虚拟机栈（Java Virtual Machine Stacks）

java虚拟机也是线程私有的，它的生命周期和线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

咱们常说的堆内存、栈内存中，栈内存指的就是虚拟机栈。局部变量表存放了编译期可知的各种基本数据类型（8个基本数据类型）、对象引用（地址指针）、returnAddress类型。

局部变量表所需的内存空间在编译期间完成分配。在运行期间不会改变局部变量表的大小。

这个区域规定了两种异常状态：如果线程请求的栈深度大于虚拟机所允许的深度，则抛出StackOverflowError异常；如果虚拟机栈可以动态扩展，在扩展是无法申请到足够的内存，就会抛出OutOfMemoryError异常。

## 本地方法栈（Native Method Stack）

　本地方法栈与虚拟机栈所发挥作用非常相似，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的native方法服务。本地方法栈也是抛出两个异常。

## 方法区（Method Area）

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。它有个别命叫Non-Heap（非堆）。当方法区无法满足内存分配需求时，抛出OutOfMemoryError异常。

## 直接内存（Direct Memory）

　直接内存不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域。但这部分区域也呗频繁使用，而且也可能导致OutOfMemoryError异常

在JDK1.4中新加入的NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。

## 运行时常量池（Runtime Constant Pool）

　运行时常量池是方法区的一部分。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，这部分内容将在加载后进入方法区的运行时常量池中存放。

## 程序计数器（Program Counter Register）

程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，一个处理器都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储。称之为“线程私有”的内存。程序计数器内存区域是虚拟机中唯一没有规定OutOfMemoryError情况的区域。

## 执行引擎

虚拟机核心的组件就是执行引擎，它负责执行虚拟机的字节码，一般户先进行编译成机器码后执行。

## 垃圾收集系统

垃圾收集系统是Java的核心，也是不可少的，Java有一套自己进行垃圾清理的机制，开发人员无需手工清理

# 垃圾回收机制算法分析

## 什么是垃圾回收机制

不定时去堆内存中清理不可达对象。不可达的对象并不会马上就会直接回收， 垃圾收集器在一个Java程序中的执行是自动的，不能强制执行，即使程序员能明确地判断出有一块内存已经无用了，是应该回收的，程序员也不能强制垃圾收集器回收该内存块。程序员唯一能做的就是通过调用System.gc 方法来"建议"执行垃圾收集器，但其是否可以执行，什么时候执行却都是不可知的。这也是垃圾收集器的最主要的缺点。当然相对于它给程序员带来的巨大方便性而言，这个缺点是瑕不掩瑜的。

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  Test test = **new** Test();  test = **null**;  System.*gc*(); // 手动回收垃圾  }  @Override  **protected** **void** finalize() **throws** Throwable {  // gc回收垃圾之前调用  System.***out***.println("垃圾回收机制...");  }  } |

### finalize方法作用

Java技术使用finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object类中定义的，因此所有的类都继承了它。子类覆盖finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

## 新生代与老年代

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类的实例对象。

在 Java 中，堆被划分成两个不同的区域：新生代 ( Young )、老年代 ( Old )。新生代 ( Young ) 又被划分为三个区域：Eden、From Survivor、To Survivor。

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

堆的内存模型大致为：



（本人使用的是 JDK1.6，以下涉及的 JVM 默认值均以该版本为准。）  
默认的，新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ( 该值可以通过参数 –XX:NewRatio 来指定 )，即：新生代 ( Young ) = 1/3 的堆空间大小。老年代 ( Old ) = 2/3 的堆空间大小。其中，新生代 ( Young ) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。  
默认的，Edem : from : to = 8 : 1 : 1 ( 可以通过参数 –XX:SurvivorRatio 来设定 )，即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为

新生代和老年代。其中新生带存放新生的对象或者年龄不大的对象，老年代则存放老年对象。

新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色的空间。

绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次

新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，则进入老年代。

## 如何判断对象是否存活

### 引用计数法

#### 概念

引用计数法就是如果一个对象没有**被任何引用指向**，则可视之为垃圾。这种方法的缺点就是不能检测到环的存在。

首先需要声明，至少主流的Java虚拟机里面都没有选用引用计数算法来管理内存。   
什么是引用计数算法：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值加１；当引用失效时，计数器值减１.任何时刻计数器值为０的对象就是不可能再被使用的。那为什么主流的Java虚拟机里面都没有选用这种算法呢？其中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。

### 根搜索算法

#### 概念

根搜索算法的基本思路就是通过一系列名为”GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，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。

这个算法的基本思想是通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索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（即GC Roots到对象不可达）时，则证明此对象是不可用的。

那么问题又来了，如何选取GCRoots对象呢？在Java语言中，可以作为GCRoots的对象包括下面几种：

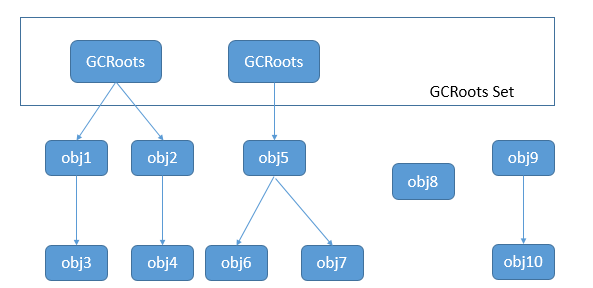
(1). 虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）中引用的对象。

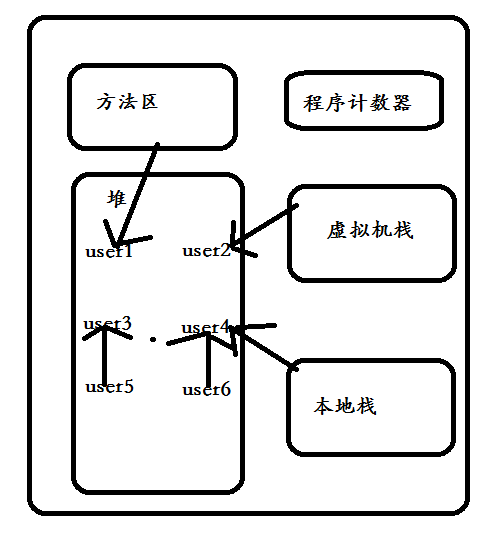
(2). 方法区中的类静态属性引用的对象。

(3). 方法区中常量引用的对象。

(4). 本地方法栈中JNI(Native方法)引用的对象。

下面给出一个GCRoots的例子，如下图，为GCRoots的引用链。





根搜索算法的基本思路就是通过一系列名为”GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，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。

从上图，reference1、reference2、reference3都是GC Roots，可以看出：   
reference1-> 对象实例1；   
reference2-> 对象实例2；   
reference3-> 对象实例4；   
reference3-> 对象实例4 -> 对象实例6；   
可以得出对象实例1、2、4、6都具有GC Roots可达性，也就是存活对象，不能被GC回收的对象。   
而对于对象实例3、5直接虽然连通，但并没有任何一个GC Roots与之相连，这便是GC Roots不可达的对象，这就是GC需要回收的垃圾对象。

## 垃圾回收机制策略

### 标记清除算法

#### 概念

该算法有两个阶段。

1. 标记阶段：找到所有可访问的对象，做个标记

2. 清除阶段：遍历堆，把未被标记的对象回收

#### 应用场景

该算法一般应用于老年代,因为老年代的对象生命周期比较长。

#### 优缺点

标记清除算法的优点和缺点

1. 优点

- 是可以解决循环引用的问题

- 必要时才回收(内存不足时)

2. 缺点：

- 回收时，应用需要挂起，也就是stop the world。

- 标记和清除的效率不高，尤其是要扫描的对象比较多的时候

- 会造成内存碎片(会导致明明有内存空间,但是由于不连续,申请稍微大一些的对象无法做到),

### 复制算法

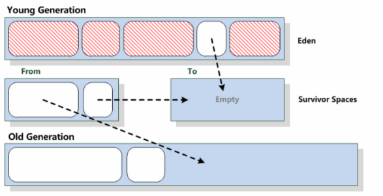
#### 概念

如果jvm使用了coping算法，一开始就会将可用内存分为两块，from域和to域， 每次只是使用from域，to域则空闲着。当from域内存不够了，开始执行GC操作，这个时候，会把from域存活的对象拷贝到to域,然后直接把from域进行内存清理。

#### 应用场景

coping算法一般是使用在新生代中，因为新生代中的对象一般都是朝生夕死的，存活对象的数量并不多，这样使用coping算法进行拷贝时效率比较高。jvm将Heap 内存划分为新生代与老年代，又将新生代划分为Eden(伊甸园) 与2块Survivor Space(幸存者区) ,然后在Eden –>Survivor Space 以及From Survivor Space 与To Survivor Space 之间实行Copying 算法。 不过jvm在应用coping算法时，并不是把内存按照1:1来划分的，这样太浪费内存空间了。一般的jvm都是8:1。也即是说,Eden区:From区:To区域的比例是

始终有90%的空间是可以用来创建对象的,而剩下的10%用来存放回收后存活的对象。



1、当Eden区满的时候,会触发第一次young gc,把还活着的对象拷贝到Survivor From区；当Eden区再次触发young gc的时候,会扫描Eden区和From区域,对两个区域进行垃圾回收,经过这次回收后还存活的对象,则直接复制到To区域,并将Eden和From区域清空。

2、当后续Eden又发生young gc的时候,会对Eden和To区域进行垃圾回收,存活的对象复制到From区域,并将Eden和To区域清空。

3、可见部分对象会在From和To区域中复制来复制去,如此交换15次(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代

注意: 万一存活对象数量比较多，那么To域的内存可能不够存放，这个时候会借助老年代的空间。

#### 优缺点

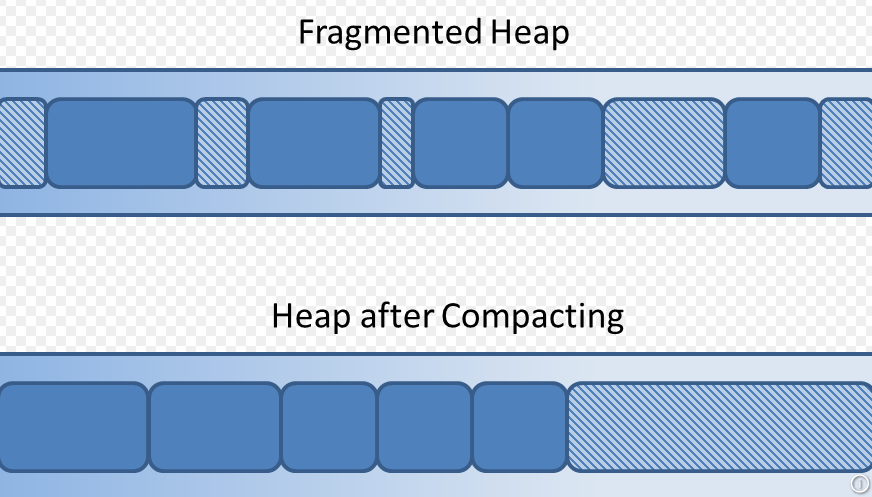
优点:在存活对象不多的情况下，性能高，能解决内存碎片和java垃圾回收算法之-标记清除 中导致的引用更新问题。

缺点: 会造成一部分的内存浪费。不过可以根据实际情况，将内存块大小比例适当调整；如果存活对象的数量比较大，coping的性能会变得很差。

### 标记压缩算法

标记清除算法和标记压缩算法非常相同，但是标记压缩算法在标记清除算法之上解决内存碎片化

#### 概念



#### 压缩算法简单介绍

任意顺序 : 即不考虑原先对象的排列顺序，也不考虑对象之间的引用关系，随意移动对象；

线性顺序 : 考虑对象的引用关系，例如a对象引用了b对象，则尽可能将a和b移动到一块；

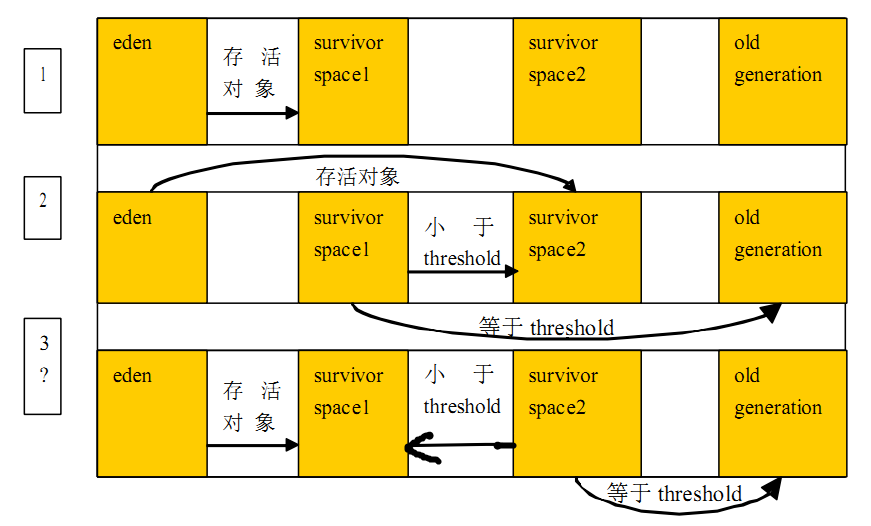
滑动顺序 : 按照对象原来在堆中的顺序滑动到堆的一端。

#### 优缺点

优点:解决内存碎片问题，缺点压缩阶段，由于移动了可用对象，需要去更新引用。

### Minor GC和Full GC区别

概念：  
 新生代 GC（Minor GC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具  
备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。  
 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（但非绝对的，在 ParallelScavenge 收集器的收集策略里  
就有直接进行 Major GC 的策略选择过程） 。MajorGC 的速度一般会比 Minor GC 慢 10  
倍以上。  
  
Minor GC触发机制：  
当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GCFull GC触发机制：  
  
当年老代满时会引发Full GC，Full GC将会同时回收年轻代、年老代，当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载其中Minor GC如下图所示  
虚拟机给每个对象定义了一个对象年龄（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 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold (阈值)来设置。



### JVM的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区  
(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

### 分代算法

#### 概述

这种算法，根据对象的存活周期的不同将内存划分成几块，新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。可以用抓重点的思路来理解这个算法。

新生代对象朝生夕死,对象数量多，只要重点扫描这个区域，那么就可以大大提高垃圾收集的效率。另外老年代对象存储久，无需经常扫描老年代，避免扫描导致的开销。

#### 新生代

在新生代，每次垃圾收集器都发现有大批对象死去，只有少量存活，采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集；可以参看我之前写的java垃圾回收算法之-coping复制

#### 老年代

而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须“标记－清除-压缩”算法进行回收。参看java垃圾回收算法之-标记\_清除压缩

新创建的对象被分配在新生代，如果对象经过几次回收后仍然存活，那么就把这个对象划分到老年代。

老年代区存放Young区Survivor满后触发minor GC后仍然存活的对象，当Eden区满后会将存活的对象放入Survivor区域，如果Survivor区存不下这些对象，GC收集器就会将这些对象直接存放到Old区中，如果Survivor区中的对象足够老，也直接存放到Old区中。如果Old区满了，将会触发Full GC回收整个堆内存。

# JVM参数调优配置

对于jvm内存配置参数：   
-Xmx10240m -Xms10240m -Xmn5120m -XXSurvivorRatio=3

## 实战OutOfMemoryError异常

### Java堆溢出

### 虚拟机栈和本地方法栈溢出

### 方法和运行时常量池溢出

### 内存溢出与内存泄漏区别

# 垃圾回收策略

## 对象优先在eden分配

## 大对象直接进入老年代

## 长期存活的对象将进入老年代

## 动态对象年龄判断

## 空间分配担保

# 理解GC日志

## JDK命令行工具

## JDK可视化工具

# 垃圾收集器

## serial收集器

## parnew收集器

## parallel scavenge收集器

## serial old收集器

## parallel old收集器

## cms收集器

## g1收集器

## 理解gc日志

## 垃圾收集器参数总结

# 垃圾收集器