JVM

# 什么是Java

## 前言

让我们来看一下Java的广告词，来自<https://www.java.com/zh_CN/about/>：

▪ 97%的企业桌面运行Java；

▪ 美国有89%的桌面(或计算机)运行Java；

▪ 全球有900万Java开发人员；

▪ 开发人员的头号选择；

▪ 排名第一的部署平台；

▪ 有30亿部移动电话运行Java；

▪ 100%的蓝光盘播放器附带了Java；

▪ 有50亿张Java卡在使用；

▪ 1.25亿台TV设备运行Java；

▪ 前5个原始设备制造商提供了Java ME。

## 为何软件开发人员选择Java

Java已由专业的Java开发人员、设计师和爱好者团体进行测试、完善、扩展和验证。Java旨在尽所能为最广泛的计算平台开发可移植的高性能应用程序。通过使应用程序在异构环境之间可用，企业可以提供更多的服务，提高最终用户生产力加强沟通与写作，从而明显降低企业和消费类应用程序的拥有成本。Java是开发人员的无价之宝，使他们可以：

▪ 在一个平台上编写软件，然后即可在几乎所有其他平台上运行；

▪ 创建可在web浏览器中运行并可访问可用web服务的程序；

▪ 开发适用于在线论坛、存储、投票、HTML格式处理以及其他用途的服务器端应用程序；

▪ 将采用Java语言的应用程序或服务组合在一起，构成高度定制的应用程序或服务；

▪为移动电话、远程处理器、微控制器、无线模块、传感器、网关、消费产品及几乎其他电子设备编写强大而高效的应用程序。

## 什么是Java

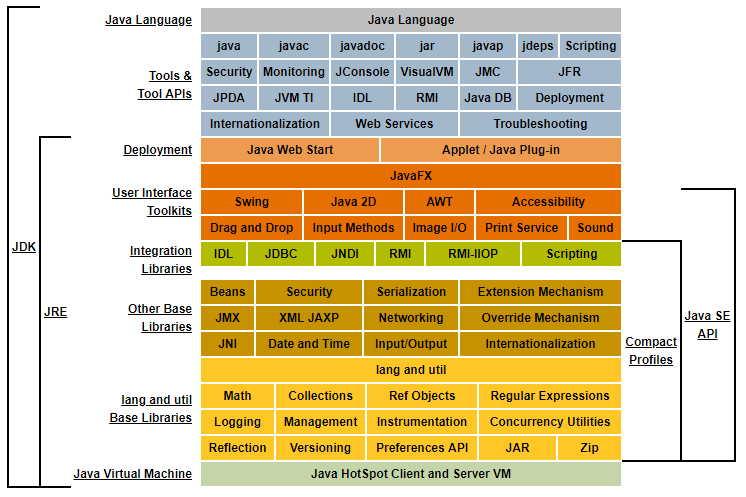
经过了多年的发展，Java早由一门单纯的计算机编程语言，演变为了一套强大的技术体系。Java设计者们将Java划分为3中结构独立但却彼此依赖的技术体系分支，他们分别对应着不同的规范集合和组件：

1. Java SE(标准版)，主要活跃在桌面领域，主要包含了Java API组件。
2. Java EE(企业版)，活跃在企业级领域，除了含Java API组件 外，还扩充有web组件、事务组件、分布式组件、EJB组件、消息组件等，综合这些技术，开发人员完全可以构建出一个具备高性能、结构严谨的企业级应用，并且Java EE也是用于构建SOA(面向服务架构)的首选平台。
3. Java ME(精简版)，活跃在嵌入式领域，称之为精简版的原因是，它仅保留了Java API中的部分组件，以及适应设备的一些特有组件。

组成Java技术体系的技术：

1. Java 编程语言；
2. 字节码；
3. Java API，包括Java API类库和来自商业机构以及开源社区的第三方类库；
4. Java 虚拟机。

也许我们只是关注了第一点,官方架构：<https://docs.oracle.com/javase/8/docs/>



## Java的优点

Java能获得如此广泛的认可，除了拥有一门结构严谨、面向对象的编程语言之外，还有许多不可忽视的优点：

1. 它能摆脱硬件平台的束缚，实现了“一次编写、到处运行”；
2. 它提供了一个相对安全的内存管理和访问机制，避免了绝大部分的内存泄漏和指针越界问题；
3. 它实现了热点代码检测和运行时编译及优化，这使得Java应用能随着运行时间的增加而获得更高的性能；
4. 它由一套完整的应用程序接口，还有无数来自商业机构和开元社区的第三方类库来帮助它实现各种各样的功能；
5. 它与身俱来对分布式技术的支持就比较完善。

但是，Java最大的优势和财富还不是以上这些，就像高翔龙老师在《Java虚拟机精讲》中写的，Java真正强大的地方是应为拥有全世界最多的技术拥护者和开源社区支持，他们无时无刻都保持着最充沛的体力与思维，一步一步地驱动着Java技术的走向。

## JDK和JRE

两个常见的重要概念。其实上面的图中已经划分出了JDK和JRE的范围了。我们队这张图做一个归纳，用我们的语言简单地总结一下什么是JDK和JRE:

1. JDK(Java Development Kit)，是用于支持Java程序开发的最小环境，基本上Java程序设计语言、Java虚拟机、Java API类库这三部分组成了JDK；
2. JRE(Java Runtime Environment)，是支持Java程序运行的标准环境，Java API类库中的Java SE API自己和Java虚拟机这两部分组成了JRE。

## OpenJDK

前面有讲过，“Java真正欠打的地方是因为拥有全世界最多的技术拥护者和开源社区支持，他们无时无刻都保持着最充沛的体力与思维，一步一步地驱动着Java技术的走向”。其实JDK在一开始并不是开源的，但是随着开源运动的蓬勃房展，2006年Sun公司宣布将对Java开放源代码，开源的Java平台开发主要集中在OpenJDK项目上。2009年4月15日，Sun公司正式发布OpenJDK，JDK 7 则是Java开源后发布的第一个版本，任何组织和个人都可以为Java的发展做出贡献。当然OpenJDK和真正的Oracle JDK(因为Sun公司被Oracle公司在2010年收购了，所以叫Oracle JDK)还是有区别的：

OpenJDK中的代码基本上都来自于Oracle JDK，属于Oracle JDK的一个分支，但是其中去除了一些非开源的组件和代码，替换成了开源的组件和代码，主要是加密和图形的部分。因此用OpenJDK代替Oracle JDK可能会有一些的不兼容。

对于OpenJDK感兴趣，OpenJDK官网<https://download.java.net/openjdk/jdk8/>下载OpenJDK的源代码。像Java虚拟机HotSpot、Java编译器Javac、JNI等，源代码都在里面。

## JCP和JSR

JCP(Java Community Process)是一套制定Java技术规范的机制，通过制定和审查JSR(Java Specification Requests)推动Java技术规范的发展。一个已经提交的JSR要想成为最终状态，需要经过正式的公开审查，并由JCP委员会投票决定，最终的JSR会提供一个参考实现，它是免费而且公开源代码的。JSR并非只由Oracle管理，任何个人都可以注册并参与审查JSR，对于Java语言发展动态感兴趣的人来说，跟踪JSR的动态发展是一条不错的学习途径。在JCP官网<https://www.jcp.org/en/home/index>中可以查看所有的JSR，下面列表几个大家熟悉的JSR:

1. JSR 14 泛型；
2. JSR 51 NIO；
3. JSR 175 注解；
4. JSR 201 枚举以及自动装箱等；
5. JSR 221 JDBC4.0 API。

## Java虚拟机

最后，轮到这个登场，也是之后文章的主角。为什么Java可以实现所谓的“一次编写、到处运行”，主要是因为虚拟机的存在。Java虚拟机负责Java程序设计语言的安全特性和平台无关性。Java虚拟机屏蔽了与具体操作系统平台相关的信息，使得Java语言编译器只需要生成在Java虚拟机上运行的字节码，就可以在多种平台上不加修改地运行。Java虚拟机使得Java摆脱了具体机器的束缚，是跨越不同平台编写程序成为了可能。

要多提一句，我们现在说的Java虚拟机基本上都是JDK自带的虚拟机HotSpot，这款虚拟机也是目前商用虚拟中市场份额最大的一款虚拟机，可以通过在命令行程序中输入“java -version”来查看：

那其实上面上还有很多别的优秀的虚拟机。Sun公司除了有大名鼎鼎的HotSpot外，还有KVM、Squawk VM、Maxine VM，BEA公司有JRockit VM、IBM公司J9 VM 等等。

# Java内存域及对象

## 几个计算机的概念

1. 计算机存储单位

1Byte = 8bit

1K = 1024Byte

1M = 1024K

1G = 1024M

1T = 1024G

1. 计算机存储元件

寄存器：中央处理器CPU的一部分，是计算机中读取速度最快的存储元件，但是容量很少；

内存：属于独立的一个部件，是和CPU沟通的桥梁，用于存放CPU中的运算数据以及与外部存储器交换的数据。尽管在今天，对内存的读写速度已经很快了，但是由于寄存器是在CPU上的，所以对于内存的读写速度和对于寄存器的读写速度上还是有几个数量级的差距。但是没办法，对于内存的读写I/O操作是很难消除的，寄存器数量有限，不可能通过寄存器来完成所有的运算任务。

1. 内核空间和用户空间

连接内存和寄存器的是地址总线地址总线的宽度影响了物理地址的索引范围，因为总线决定了处理器一次可以从寄存器或内存中获取多少个Bit，同时也决定了处理器最大可以寻址的地址的地址空间。比如32位CPU的系统，可寻址范围为0x00000000~0xFFFFFFFF，即232 = 4294967296个内存位置，每个内存位置1个字节，即32位CPU系统可以有4GB的内存空间。不过应用程序时不可以完全使用这些地址空间的，因为这些地址空间划分为了内核空间和用户空间，程序只能使用用户空间的内存。内核空间主要是指操作系统运行时所使用的用于程序调度、虚拟内存的使用或者链接硬件资源的程序逻辑。区分内核空间和用户空间的目的主要是从系统的稳定性的角度考虑的。Windows 32操作系统默认内核空间和用户空间的比例1:1，即2G内核空间、2G内存空间，32位Linux系统中默认比例则是1:3，即1G内核空间，3G内存空间。

1. 字长

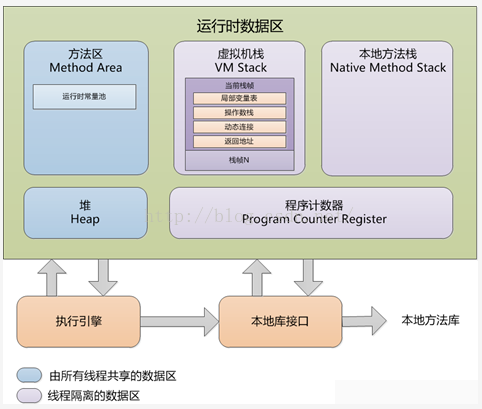
CPU的主要技术指标之一，指的是CPU一次能并行处理二进制的位数(Big)。通常称处理字长为8位数据的CPU为8位CPU，32位CPU就是在同一时间内处理字长为32位的二进制数据。不过目前虽然CPU大多是64位的，但还是以32位字长运行。

## 前言

说到Java内存区域，可能很多人第一反应是“堆栈”。首先堆栈不是一个概念，而是两个概念，堆和栈是两块不同的内存区域，简单理解的话，堆是用来存放对象而栈是用来执行程序的。其次，堆内存和栈内存的这种划分方式比较粗糙，这种划分方式只能说明大多数程序猿最关注的的、与对象内存分配关系最密切的内存区域是这两块，Java内存区域的划分实际上远比这复杂。对于Java程序猿来说，在虚拟机自动内存管理机制的帮助下，不在需要为每一个new操作去配对delete/free代码，不容易出现内存泄漏和内存溢出问题。但是，也正是因为Java把内存控制权交给了虚拟机，一旦出现内存泄漏和内存溢出的问题，就难以排查，因此一个好的Java程序猿应该去了解虚拟机的内存区域以及会引起内存泄漏和内存溢出的场景。

## 运行时数据区域

Java虚拟机内部定义了程序在运行时需要使用到的内存区域



之所以要划分这么多区域出来是因为这些区域都有自己的用途，以及创建和销毁的时间。有些区域随着虚拟机进程的启动而存在，有的区域则依赖用户线程的启动和结束而销毁和建立。图中绿色部分就是所有线程之间共享的内存区域，而白色部门则是线程运行时独有的数据区域，从这个分类角度来看一下这几个数据区：

1. 线程独有的内存区域
2. PROGRAM COUNTER REGISTER，程序计数器

这块内存区域很小，它是当前线程所执行的字节码的行号指示器，字节码解释器通过改变计数器的值来选取下一条需要执行的字节码指令。Java方法这个计数器才有值，如果执行的是一个Native方法，那这个计数器是空的。

1. JAVA STACK 虚拟机栈

生命周期和线程相同。每个方法执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，每一个方法从调用直至执行完毕的过程，就对应这一个栈帧在虚拟机中入栈和出栈的过程。栈的大小和具体JVM的实现有关，通常在256K~756K之间。

1. NATIVE METHOD STACK 方法栈

和虚拟机栈起的作用一样，只不过方法栈为虚拟机使用的Native方法服务。虚拟机规范并没有对这个区域有什么规定，因此我们使用HotSpot虚拟机，就干脆没有这块区域了，它和虚拟机栈是一起的。

1. 线程之间共享的内存区域
2. HEAP 堆

大多数应用，堆都是Java虚拟机所管理的内存中最大的一块，它在虚拟机启动时创建，此内存唯一的目的就是存放对象实例。由于现在垃圾收集器采用的基本都是分代收集算法，所以堆还可以细分为新生代和老年代，在细致一点还有Eden区、From Survivior区、To Survivor区，这个后面都会讲到的。

1. METHOD AREA 方法区

这块区域用于存储虚拟机加载的类信息、常量、静态变量、即时编译器后的代码等数据，虚拟机规范是把这块区域描述为堆的一个逻辑部分的，但实际它应该是要和堆区分开的。从上面提到的分代收集算法的角度看，HotSpot中，方法区≈永久代。不过JDK7之后，我们使用的HotSpot应该就没有永久这个概念了，会采用Native Memory来实现方法区的规划了。

1. RUNTIME CONSTANT POOL 运行时常量池

上面的图中没有画出来，因为他是方法区的一部分。Class文件中除了有类的版本信息、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期间生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中，另外翻译出来的直接引用也会存储在这个区域汇总。这个区域另外一个特点就是动态性，Java并不要求常量就一定要在编译期间才能产生，运行期间也可以在这个区域放入新的内容，String.intern()方法就是这个特性的应用。

1. 直接内存

直接内存并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。但是这部分内存也被频繁的使用，而且也可能导致内存溢出问题。JDK1.4中新增加了NIO，引入了一种基于通道与缓冲区的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。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native中来回复制数据。显然，本机直接内存的分配不会受到Java堆大小限制，但是，既然是内存，肯定还是会受到本机总内存(包括RAM、SWAP区)大小以及处理器寻址空间的限制。

## 对象创建

Java是一门面向对象的语言，Java程序运行过程中无时无刻都有对象被创建出来。在语言层面上，创建对象(克隆、反序列化)就是一个new 关键字而已，但是虚拟机层面上却不是如此。看一下在虚拟机层面上创建对象的步骤：

1. 虚拟机遇到一条new指令，首先去检查这个指令的参数能否在常量池中定位定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析、初始化。如果没有，那么必须先执行类的初始化过程。
2. 类加载检查通过后，虚拟机为新生对象分配内存。兑现所需内存大小在类加载完成后便可以完全确定，为对象分配空间无非就是从Java堆中划分出一块确定大小的内存而已。这个地方会有两个问题：
   1. 如果内存时规整的，那么虚拟机将采用的是指针碰撞法来为对象分配内存。意思是所有用过的内存在一边，空闲的内存在另一边，中间放着一个指针作为分界点的指示器，分配内存就是仅仅是把指针指向空闲那一边挪动一段与对象大小相同的距离罢了。如果垃圾回收器选择的是Serial、ParNew这种基于压缩算法的，虚拟机采用这种分配方式。
   2. 如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表法为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。如果垃圾收集器选择的是CMS这种基于标记-清除算法的，虚拟机采用这种分配方式。

另外一个问题及时保证new对象时候的线程安全性。因为可能出现虚拟机正在给对象A分配内存，指针还没有来得及修改，对象B又同时使用了原来的指针来分配内存的情况。虚拟机采用了CAS配上失败重试的方式保证更新操作的原子性和TLAB两种方式来解决这个问题。

1. 内存分配结束，虚拟机将分配到的内存空间都初始化为零值(不包括对象头)。这一步保证了对象的实例字段在Java代码中可以不用赋初始值就可以直接使用，程序能访问到这些字段的数据类型多对应的零值。
2. 对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息，这些信息存放在对象的对象头中。
3. 执行<init>方法，把对象按照程序猿的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

## 对象定位方式

建立对象是为了使用对象，Java程序需要通过栈上的reference(引用)数据来操作堆上的具体对象。比如：

|  |
| --- |
| Object obj = new Object(); |

而new Object()之后其实有两部分内容，一部分是类数据(比如代表类的Class对象)、一部分是实例数据。

由于reference在Java虚拟机规范中只是一个指向对象new Object()引用obj，并没有规定obj应该通过何种方式去定位、访问堆中对象的具体位置，所以对象访问方式也是取决于虚拟机而定的。主流方式有两种：

1. 句柄访问。Java堆中划分出一块句柄池，obj很仔细那个的是对象的句柄地址，句柄中则包含了类数据的地址和实例数据的地址。
2. 指针访问。对象中存储所有的实例数据和类数据的地址，obj指向的是这个对象HotSpot虚拟机采用的是后者，不过前者的对象访问方式也是十分常见的。

# 常用JVM命令参数

1. -Xms20M

表示设置堆容量的最小值20M，必须以M为单位

1. -Xmx20M

表示设置堆容量的最大值为20M，必须以M为单位。将-Xmx和-Xms设置为一样可以避免堆自动扩展。大的项目-Xmx和-Xms一般都要设置到10G、20G甚至更大

1. -verbise:gc

表示输出虚拟机中GC的详细情况

1. -Xss128k

表示输出虚拟机中GC的详细情况

1. -Xoss128k\

表示设置本地方法栈的大小为128k。不过HotSpot并不区分虚拟机栈和本地方法栈，因此对于HotSpot来说这个参数是无效的。

1. -XX:PermSize=10M

表示JVM初始化分配的永久代的容量，必须以M为单位

1. -XX:MaxPermSize=10M

表示JVM允许分配的永久代的最大容量，必须以M为单位，大部分情况下这个参数默认为64M

1. -Xnoclassgc

表示关闭JVM对类的垃圾回收

1. -XX:+TraceClassLoading

表示查看类的加载信息

1. -XX:+TraceClassUnLoading

表示查看类的卸载信息

1. -XX:NewRatio=4

表示设置年轻代：老年代的大小比值为1:4，这意味着年轻代占整个堆的1/5

1. -XX:SurvivorRatio=8

表示设置2个Survivor区：1个Eden区的大小比值为2:8，这意味着Survivor区占整个年轻代的1/5,这个参数默认为8

1. -Xmn20M

表示设置年轻代的大小为20M

1. -XX:+HeapDumpOnOutOfMemoryErro

表示可以让虚拟机在出现内存溢出异常时Dump出当前的堆内存转换快照

1. -XX:+UserG1GC

表示让JVM使用G1垃圾收集器

1. -XX:+PrintGCDetails

表示在控制台上打印出GC具体细节

1. -XX:+PrintGC

表示在控制台上打印出GC信息

1. -XX:PretenureSizeThreshold=3145728

表示对象大于3145728(3M)时直接进入老年代分配，这里只能以字节作为单位

1. -XX:MaxTenuringThreshold=1

表示对象年龄大于1，自动进入老年代

1. -XX:CompileThreshold=1000

表示一个方法被调用1000次之后，会被认为是热点代码，并触发及时编译

1. -XX:+PrintHeapAtGC

表示可以看到每次GC前后堆内存布局

1. -XX:+PrintTLAB

表示可以看出TLAB的使用情况

1. -XX:+UserSpining

开启自旋锁

1. -XX:PreBlockSpin

更改自旋锁的自旋次数，使用这个参数必须先开启自旋锁

# 内存溢出

## 堆溢出

Java堆唯一的作用就是存储对象实例，只要保证不断创建对象并且对象不被回收，那么对象数量达到最大堆容量限制后就会产生内存溢出异常了。所以测试的时候把堆的大小固定住并且让堆不可扩展即可。测试代码如下：

|  |
| --- |
| /\*\*  \* 测试内容：堆溢出  \*  \* 虚拟机参数：-Xms20M -Xmx20M -XX:+HeapDumpOnOutOfMemoryError  \*/  public class HeapOverflowTest  {  public static void main(String[] args)  {  List<HeapOverflowTest> list = new ArrayList<HeapOverflowTest>();  while (true)  {  list.add(new HeapOverflowTest());  }  }  } |

运行结果：

|  |
| --- |
| Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  at java.util.Arrays.copyOf(Arrays.java:3210)  at java.util.Arrays.copyOf(Arrays.java:3181)  at java.util.ArrayList.grow(ArrayList.java:261)  at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)  at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)  at java.util.ArrayList.add(ArrayList.java:458)  at com.daily.dailytest.jvm.HeapOverflowTest.main(HeapOverflowTest.java:24) |

这种异常很常见，也很好发现，因为都提示了“Java heap space”了，定位问题的话，根据异常堆栈分析就好了，行号都有指示。解决方案的话，可以调大堆的大小或者从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，长时间少程序运行期间的内存消耗。

## 栈溢出

Java虚拟机规范中描述了如果线程请求的栈深度太深(换句话说方法调用深度太深)，就会产生栈溢出了。那么，我们只要写一个无线调用自己的方法，自然就会出现方法调用的深度太深的场景了。测试代码如下：

|  |
| --- |
| public class StackOverflowTest {  private int stackLength = 1;  public void stackLeak() {  stackLength ++;  stackLeak();  }  public static void main(String[] args) {  StackOverflowTest stackOverflow = new StackOverflowTest();  try {  stackOverflow.stackLeak();  } catch (Throwable e) {  System.out.println("stack length: " + stackOverflow.stackLength);  throw e;  }  }  } |

运行结果：

|  |
| --- |
| stack length: 20527  Exception in thread "main" java.lang.StackOverflowError  at com.daily.dailytest.jvm.StackOverflowTest.stackLeak(StackOverflowTest.java:22)  at com.daily.dailytest.jvm.StackOverflowTest.stackLeak(StackOverflowTest.java:22)  at com.daily.dailytest.jvm.StackOverflowTest.stackLeak(StackOverflowTest.java:22) |

后面都是一样的，忽略。通过不断创建线程的方式可以产生OutOfMemoryError，因为每个线程都有自己的栈空间。不过这个操作有文献就不做了，姻缘是Windows平台下，Java的线程是直接映射到操作系统的内核线程上的，如果写个死循环无限产生线程，那么可能会造成操作系统的假死。

上面无线产生线程的场景，从另一个角度说，就是为每个线程的栈分配的内存空间越大，反而越容易产生内存溢出。其实这也很好理解，操作系统分配给进程的内存时有限的，比如32位的Windows限制为2GB。虚拟机提供了了参数来控制Java堆和方法区这两部分内存的最大值，剩余内存为2GB-最大堆容量-最大方法区容量，程序计数器很小就忽略了，虚拟机进程本身的耗费也不算，剩下的内存就是栈的了。每个线程分配到的栈容量越大，可建立的线程数目越少，建立线程时就越容易把剩下的内存耗尽。

StackOverFlowError这个异常，有错误堆栈可以阅读，比较好定位。而且如果适用虚拟机默认参数，栈深度在大多数情况下，达到1000~2000完全没有问题，正常方法的调用这个深度应该是完全足够啦。但是如果建立过多的线程导致的OutOfMemoryError，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减小最大堆容量和减小栈容量来换取更多的线程了。

## 方法区和运行时常量池溢出

运行时常量池也是方法区的一部分，所以这两个区域一起看就可以了。这个区域的OutOfMemoryError可以利用String.intern()方法来产生。这是一个Native方法，意思是如果常量池中有一个String对象的字符串就返回池中的这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中去，并且返回此String对象的引用。测试代码如下：

|  |
| --- |
| public class ConstantPoolOverflowTest {  public static void main(String[] args) {  List<String> list = new ArrayList<>();  int i = 0;  while (true) {  list.add(String.valueOf(i++).intern());  }  }  } |

运行结果：

|  |
| --- |
| Exception in thread "Reference Handler" Exception in thread "main" java.lang.OutOfMemoryError: PermGen space  at java.lang.String.intern(Native Method)  at com.xrq.test.ConstantPoolOverflowTest.main(ConstantPoolOverflowTest.java:19)  java.lang.OutOfMemoryError: PermGen space  at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:123) |

上文中说过，对于HotSpot而言，方法区=永久代，这里看到OutOfMemoryError的区域是“PermGen space”，即永久代，那其实也就是方法区溢出了。注意JDK1.7 下是不会有这个异常的，while循环将一直进行下去，因为JDK1.7之后溢出了永久代并采用Native Memory来实现方法区的规划了。

# Java垃圾回收机制(GC)

## 哪些内存需要回收？

哪些垃圾需要回收是垃圾回收机制第一个要考虑的问题，所谓“要回收的垃圾”无非就是那些不可能再被任何途径使用的对象。那么如何找到这些对象？

1. 引用计数法

这个算法的实现是，给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器-1。任何时刻计数值为0的对象就是不可能再被使用的。这种算法使用场景很多，但是，Java中却没有使用这种算法，因为这种算法很难解决对象之间相互引用的情况。看一段代码：

|  |
| --- |
| public class ReferenceCountingGC {  private Object instance = null;  private static final int \_1MB = 1024 \* 1024;  private byte[] bigSize = new byte[2 \* \_1MB];  public static void main(String[] args) {  ReferenceCountingGC objectA = new ReferenceCountingGC();  ReferenceCountingGC objectB = new ReferenceCountingGC();  objectA.instance = objectB;  objectB.instance = objectA;  objectA = null;  objectB = null;  System.gc();  }  } |

运行结果：

|  |
| --- |
| [GC 4417K->288K(61440K), 0.0013498 secs]  [Full GC 288K->194K(61440K), 0.0094790 secs] |

看到，两个对象互相引用着，但是虚拟机还是把这两个对象回收掉了，这也说明虚拟机并不是通过引用计数法来判定对象是否存活的。

1. 可达性分析法

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。在Java语言中可以作为GC Roots的对象包括：

▪ 虚拟机栈中引用的对象；

▪ 方法区中静态属性引用的对象；

▪ 方法区中常量引用的对象

▪ 本地方法栈中JNI(即Native方法)引用的对象。

## 四种引用状态

在JDK1.2之前，Java中引用的定义很传统：如果引用类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过于狭隘，一个对象只有被引用或者没被引用两种状态。我们希望描述这样一类对象：当内存空间还足够时，则能保留在内存中；如果内存空间在进行垃圾收集后还是紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。在JDK1.2之后，Java对引用的概念进行了补充，将引用分为强引用、软引用、弱引用、虚引用4种，这4种引用强度一次减弱。

1. 强引用

代码中普遍存在的类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾回收期永远不会回收掉被引用的对象

1. 软引用

描述有些还有用但并非必须的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用

1. 弱引用

描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾回收器工作之后，无论当前内存是否足够，都会回收掉只被若引用关联的对象。Java中类WeakReference表示弱引用

1. 虚引用

这个引用存在的唯一目的就是在这个对象被收集器回收时收到一个系统通知，被虚引用关联的对象，和其生存时间完全没关系。Java中的类PhantomReference表示虚引用

## 方法区回收

虚拟机规范中不要求方法区一定要实现垃圾回收，而且方法区中进行垃圾回收的效率也确实比较低，但是HotSpot对方法区也是进行回收的，主要回收的是废弃常量和无用的类两部分。判断一个常量是否“废弃常量”比较简单，只要当前系统中没有任何一处引用该常量就好了，但是要判断一个类是否“无用的类”条件就要苛刻很多，类需要同时满足一下三个条件：

1. 该类所有实例都已经被回收，也就是说Java堆中不存在该类的任何实例；
2. 加载该类的ClassLoader已经被回收；
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法

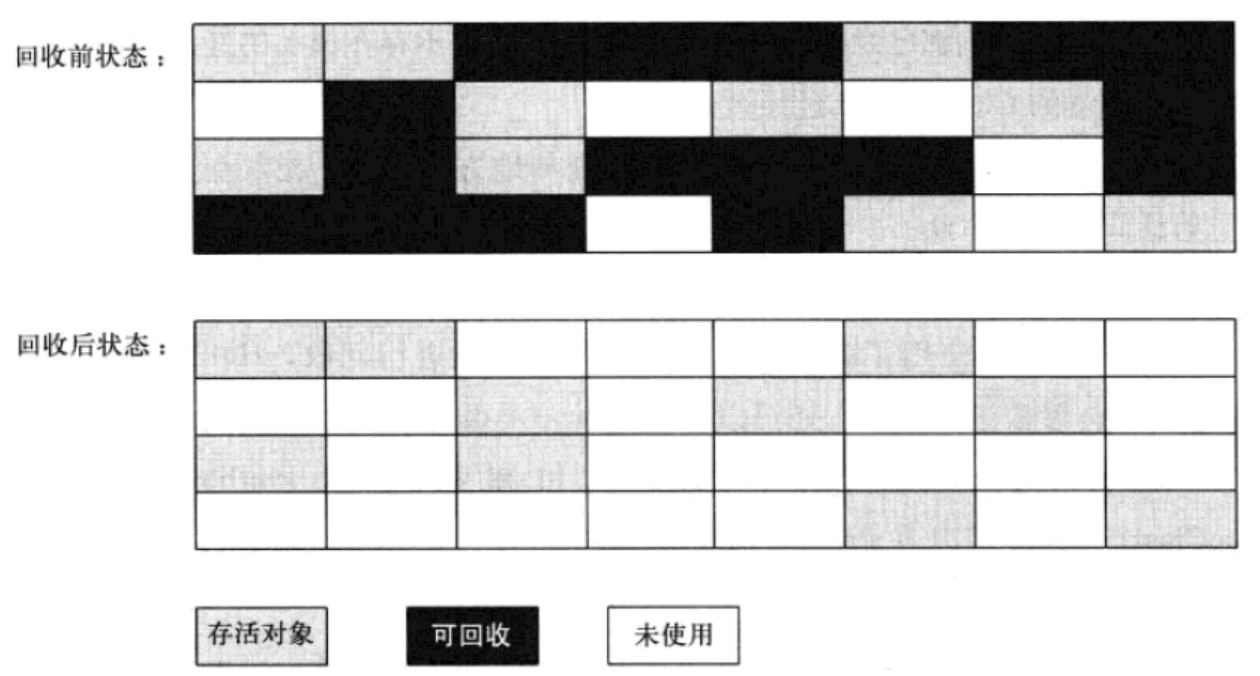
在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGI这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载功能，以保证方法区不会溢出。

## 垃圾回收算法

第一步考量了哪些对象进行回收后，第二步自然是如何对对象进行回收了。这里主要写几种垃圾回收算法的思想。

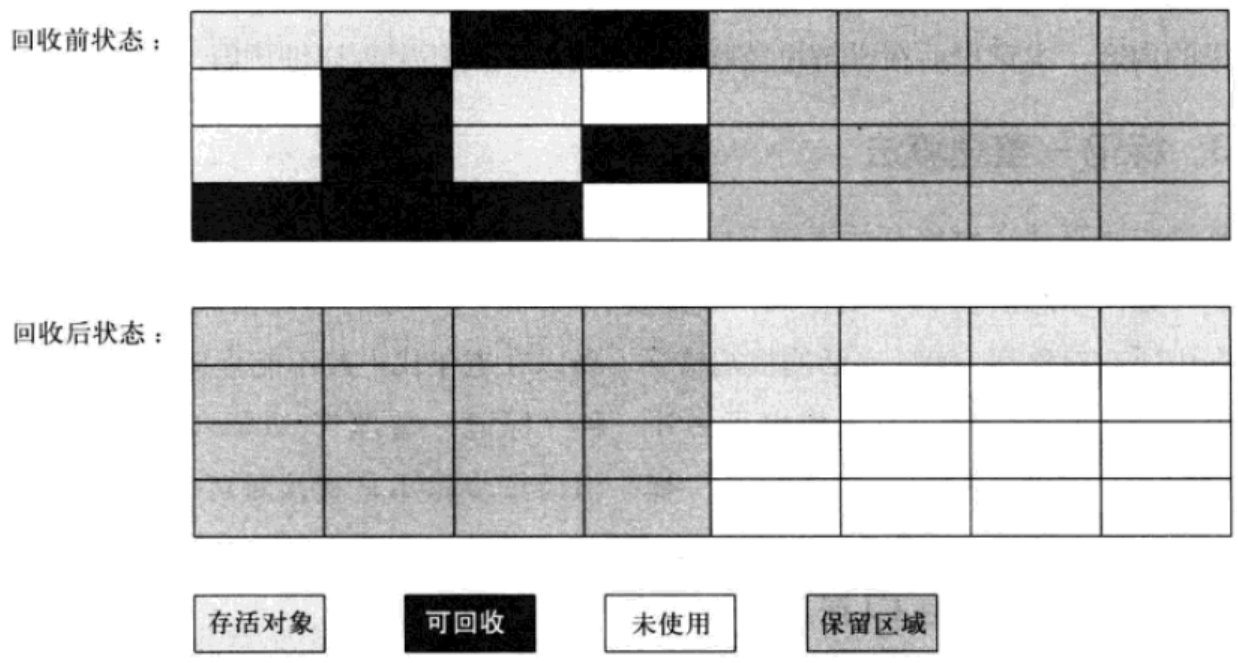
1. 标记-清除(Mark-Sweep)算法

这是最基础的算法，标记-清除算法就是如同它的名字一样，分别“标记”和“删除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。这种算法的不足主要体现在效率和空间，从效率的角度讲，标记和清除两个过程的效率都不高；从空间的角度讲，标记清除后会产生大量不连续的内存碎片，内存碎片太多可能会导致以后程序运行过程中在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出发一次垃圾收集动作。标记-清除算法执行过程如图：



1. 赋值(Coping)算法

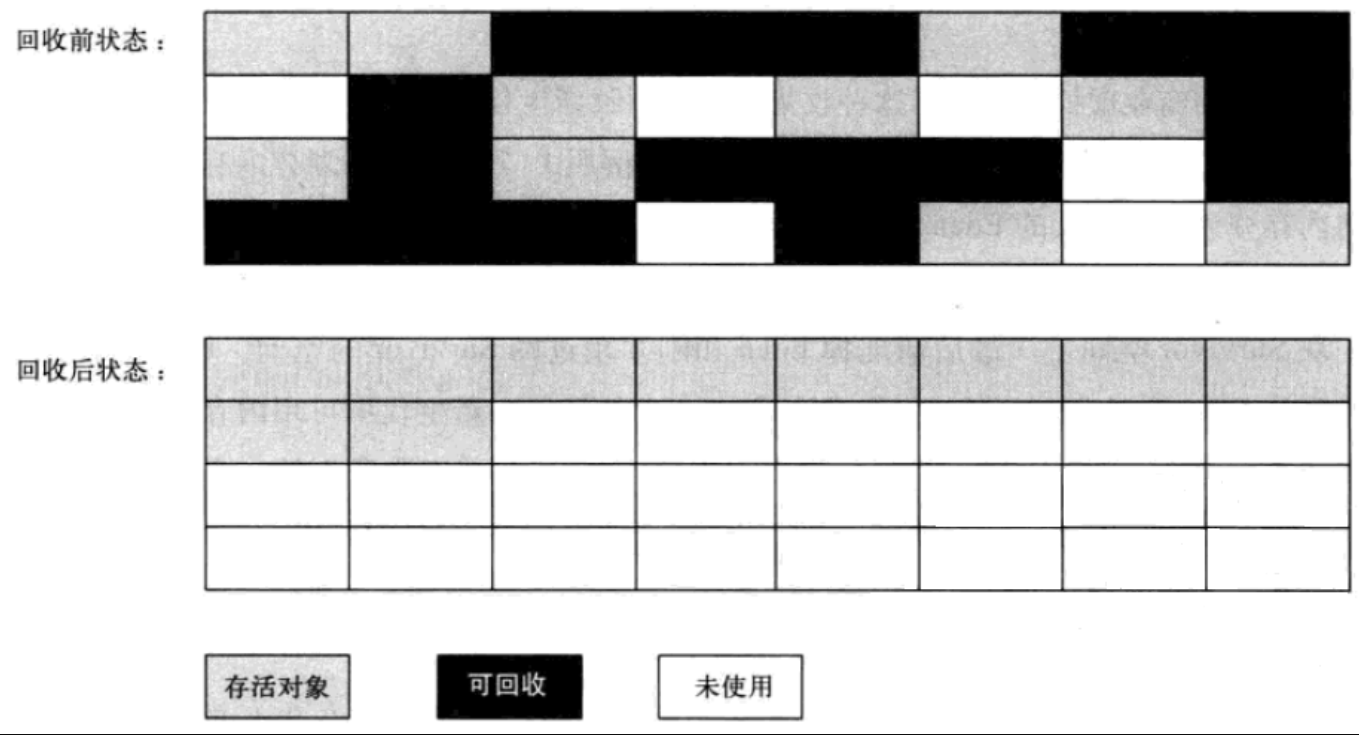
赋值算法是为了解决效率问题而出现的，它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。复制算法的执行过程如图：



不过这种算法有个缺点，内存缩小为原来的一半，这样代价太高了。现在的商用虚拟机都采用这种算法来回收新生代，不过研究表明1:1的比例非常不科学，因此新生代的内存被划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用过Eden和其中一块Survivor。每次回收时，将Eden和Survivor中还存活着的对象一次性复制到另一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden区和Survivor区的比例为8:1，意思是每次新生代中可用内存空间为整个新生代容量的90%。当然，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够时，需要依赖老年代进行分配承担(Handle Promotion)

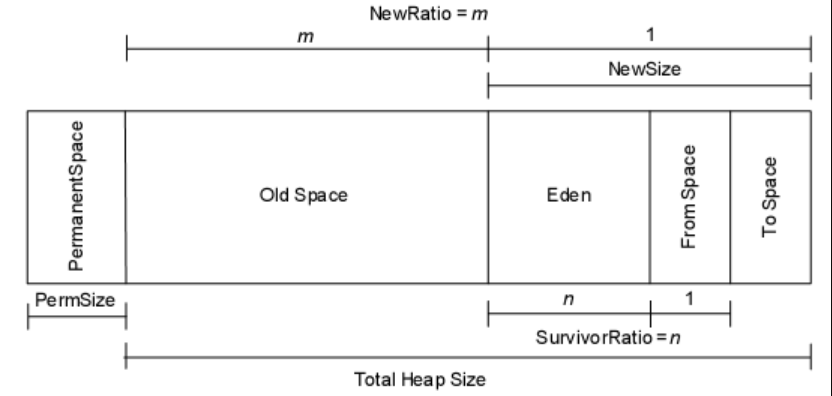
1. 标记-整理(Mark-Compact)算法

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%存活，那么需要有额外的空间进行分配担保。老年代但是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，有人提出了另外一种标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉便捷以外的内存。标记-整理算法的工作过程如图：



## 分代收集

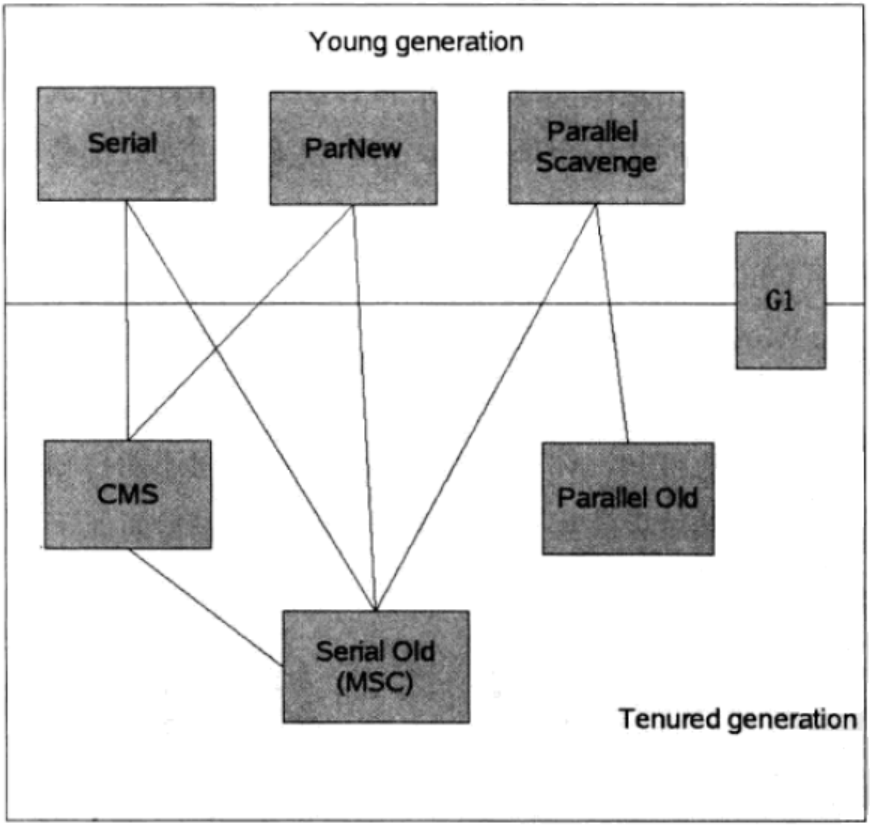
根据上面的内容，用一张概括一下堆内存的布局



现在商用虚拟机基本上都采用分代收集算法进行垃圾回收。这种算法没什么特别的，无非是上面内容的结合罢了，根据对象的生命周期的不同将内存划分为几块，然后根据各块的特点采用最适当的收集算法。大批对象死去、少量对象存活的，使用负值算法，复制成本低；对象存活率高、没有额外空间进行分配担保的，采用标记-清理法或者标记-整理算法。

## 垃圾收集器

垃圾回收期就是上面讲的理论知识的具体实现了。不同虚拟机所提供的垃圾收集器可能会有很大差别，我们使用的是HotSpot，HotSpot这个虚拟机所包含的所有收集器如图：



上图展示了7中作用于不同分代的收集器，如果两个收集器之间存在连线，那说明他们可以搭配使用。虚拟机所处的区域说明他是属于新生代收集器还是老年代收集器。多说一句，我们必须要明白一个道理：没有最好的垃圾收集器，更加没有万能的收集器，只能选对具体应用最合适的收集器。这也是HotSpot为什么要实现这么多收集器的原因。OK，下面一个一个看一下收集器：

1. Serial收集器

最基本、发展历史最久的收集器，这个收集器是一个采用赋值算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束。后者意味着，在用户不可见的情况下要把用户正常工作的线程全部停掉，这对很多应用是难以接受的。不过实际上到目前为止，Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器，因为他简单高效。用户桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，手机几十兆甚至一两百兆的新生代停顿时间再几十毫秒最多一百毫秒，只要不是频繁发生，这点停顿是完全可以接受的。

1. 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外，其余行为和Serial收集器完全一样，包括使用的也是赋值算法。ParNew收集器除了多线程意外和Serial收集器并没有太多创新的地方，但是他却是Server模式下的虚拟机首选的新生代收集器，其中有一个很重要的和性能无关的原因是，除了Serial收集器外，目前只有他能与CMS收集器配合工作。CMS收集器是一款集合可以认为有划时代意义的垃圾收集器，因为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。ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至优于线程交互的开销，该收集器在两个CPU环境中都不能百分百保证可以超越Serial收集器。当然，随着可用CPU数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU数量相同，在CPU数量非常多的情况下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

1. Parallel收集器

Parallel收集器也是一个新生代收集器，也是用赋值算法的收集器，也是并行的多线程收集器，但是他的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍吞吐量的概念。CMS等收集器的关注点是尽可能缩短垃圾收集时用户 线程的停顿时间，而Parallel收集器的目标则是达到一个可控制的吞吐量。所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)，虚拟机总运行100分钟，垃圾收集器1分钟，那吞吐量就是99%。另外，Parallel收集器是虚拟机运行在Server模式下的默认垃圾收集器。

停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；高吞吐则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。

虚拟机提供了-XX:MaxGCPauseMillis和-XX:GCTimeRatio两个参数来精确控制最大垃圾收集停顿时间和吞吐量大小。不过不要以为前者越小越好，GC停顿时间的缩短是以牺牲吞吐量和新生代空间换取的。由于与吞吐量关系密切，Parallel收集器也被称为“吞吐量优先收集器”。Parallel收集器有一个-XX:+UseAdaptiveSizePolicy参数，这是一个开关参数，这个参数打开之后，就不需要手动指定新生代大小，Eden区和Survivor参数等细节参数了，虚拟机会根据当前系统的运行情况手机性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。如果对于垃圾收集器运行原理不太了解，以至于在优化比较困难的时候，使用Parallel收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。

1. Serial Old收集器

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”，这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

1. Parallel收集器

Parallel收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器在JDK1.6之后的出现，“吞吐量优先收集器”终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel收集器+Parallel Old收集器的组合

1. CMS收集器

CMS收集器是一种以获取最短回收停顿时间为目标的老年代收集器。目前很大一部分Java应用 中在互联网站或者B/S系统的服务端上，这类应用尤其注重服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验，CMS收集器就非常符合这类应用的需求。CMS收集器从名字就能看出是基于“标记-清除”算法实现的

1. G1收集器

G1(Garbage-First)收集器是当今收集器技术发展的最前沿成果之一，JDK7 Update 4后开始进入商用。在G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆分为多个大小相等的独立区域(Region)，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，他们都是一部分Region的集合。G1收集器跟踪各个Region里面的垃圾堆积的价值大小，在后台维护一个有限列表，每次根据允许的手机时间，优先回收价值最大的Region(这也是Garbage-First名称的由来)。这种使用Region划分内存空间以及优先级的区域回收方式，保证了G1收集器在有限的时间尽可能高的收集效率

## 垃圾收集器总结

|  |  |  |  |
| --- | --- | --- | --- |
| GC组合 | Minor GC | Full GC | 描述 |
| -XX:+UseSerialGC | Serial收集器串行回收 | Serial Old收集器串行回收 | 该选项可以手动指定Serial收集器+Serial Old收集器组合执行内存回收 |
| -XX:+UseParNewGC | ParNew收集器并行回收 | Serial Old收集器串行回收 | 该选项可以手动指定ParNew收集器+Serilal Old组合执行内存回收 |
| -XX:+UseParallelGC | Parallel收集器并行回收 | Serial Old收集器串行回收 | 该选项可以手动指定Parallel收集器+Serial Old收集器组合执行内存回收 |
| -XX:+UseParallelOldGC | Parallel收集器并行回收 | Parallel Old收集器并行回收 | 该选项可以手动指定Parallel收集器+Parallel Old收集器组合执行内存回收 |
| -XX:+UseConcMarkSweepGC | ParNew收集器并行回收 | 缺省使用CMS收集器并发回收，备用采用Serial Old收集器串行回收 | 该选项可以手动指定ParNew收集器+CMS收集器+Serial Old收集器组合执行内存回收。优先使用ParNew收集器+CMS收集器的组合，当出现ConcurrentMode Fail或者Promotion Failed时，则采用ParNew收集器+Serial Old收集器的组合 |
| XX:+UseConcMarkSweepGC-XX:-UseParNewGC | Serial收集器串行回收 |
| -XX:+UseG1GC | G1收集器并发、并行执行内存回收 | 暂无 | |

## GC日志

每种收集器的日志形式都是有它们自身的实现所决定的，换言之，每种收集器的日志格式都可以不一样。不过虚拟机为了方便用户阅读，将各个收集器的日志都维持了一定的共性，就以最前面的对象间相互引用的那个类ReferenceCountingGC的代码为例：

虚拟机参数为“**-XX:+PrintGCDetails -XX:+UseSerialGC**”，使用Serial+Serial Old组合进行垃圾回收的日志

|  |
| --- |
| [GC [DefNew: 310K->194K(2368K), 0.0269163 secs] 310K->194K(7680K), 0.0269513 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]  [GC [DefNew: 2242K->0K(2368K), 0.0018814 secs] 2242K->2241K(7680K), 0.0019172 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (System) [Tenured: 2241K->193K(5312K), 0.0056517 secs] 4289K->193K(7680K), [Perm : 2950K->2950K(21248K)], 0.0057094 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  Heap  def new generation total 2432K, used 43K [0x00000000052a0000, 0x0000000005540000, 0x0000000006ea0000)  eden space 2176K, 2% used [0x00000000052a0000, 0x00000000052aaeb8, 0x00000000054c0000)  from space 256K, 0% used [0x00000000054c0000, 0x00000000054c0000, 0x0000000005500000)  to space 256K, 0% used [0x0000000005500000, 0x0000000005500000, 0x0000000005540000)  tenured generation total 5312K, used 193K [0x0000000006ea0000, 0x00000000073d0000, 0x000000000a6a0000)  the space 5312K, 3% used [0x0000000006ea0000, 0x0000000006ed0730, 0x0000000006ed0800, 0x00000000073d0000)  compacting perm gen total 21248K, used 2982K [0x000000000a6a0000, 0x000000000bb60000, 0x000000000faa0000)  the space 21248K, 14% used [0x000000000a6a0000, 0x000000000a989980, 0x000000000a989a00, 0x000000000bb60000)  No shared spaces configured. |

虚拟机参数为“**-XX:+PrintGCDetails -XX:+UseParNewGC**”，使用ParNew+Serial Old组合进行垃圾回收的日志

|  |
| --- |
| [GC [ParNew: 310K->205K(2368K), 0.0006664 secs] 310K->205K(7680K), 0.0007043 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC [ParNew: 2253K->31K(2368K), 0.0032525 secs] 2253K->2295K(7680K), 0.0032911 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (System) [Tenured: 2264K->194K(5312K), 0.0054415 secs] 4343K->194K(7680K), [Perm : 2950K->2950K(21248K)], 0.0055105 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  Heap  par new generation total 2432K, used 43K [0x0000000005550000, 0x00000000057f0000, 0x0000000007150000)  eden space 2176K, 2% used [0x0000000005550000, 0x000000000555aeb8, 0x0000000005770000)  from space 256K, 0% used [0x0000000005770000, 0x0000000005770000, 0x00000000057b0000)  to space 256K, 0% used [0x00000000057b0000, 0x00000000057b0000, 0x00000000057f0000)  tenured generation total 5312K, used 194K [0x0000000007150000, 0x0000000007680000, 0x000000000a950000)  the space 5312K, 3% used [0x0000000007150000, 0x0000000007180940, 0x0000000007180a00, 0x0000000007680000)  compacting perm gen total 21248K, used 2982K [0x000000000a950000, 0x000000000be10000, 0x000000000fd50000)  the space 21248K, 14% used [0x000000000a950000, 0x000000000ac39980, 0x000000000ac39a00, 0x000000000be10000)  No shared spaces configured. |

虚拟机参数为“**-XX:+PrintGCDetails -XX:+UseParallelGC**”，使用Parallel+Serial Old组合进行垃圾回收的日志

|  |
| --- |
| [GC [PSYoungGen: 4417K->288K(18688K)] 4417K->288K(61440K), 0.0007910 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (System) [PSYoungGen: 288K->0K(18688K)] [PSOldGen: 0K->194K(42752K)] 288K->194K(61440K) [PSPermGen: 2941K->2941K(21248K)], 0.0032663 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]  Heap  PSYoungGen total 18688K, used 321K [0x0000000034190000, 0x0000000035660000, 0x0000000048f90000)  eden space 16064K, 2% used [0x0000000034190000,0x00000000341e05c0,0x0000000035140000)  from space 2624K, 0% used [0x0000000035140000,0x0000000035140000,0x00000000353d0000)  to space 2624K, 0% used [0x00000000353d0000,0x00000000353d0000,0x0000000035660000)  PSOldGen total 42752K, used 194K [0x000000000a590000, 0x000000000cf50000, 0x0000000034190000)  object space 42752K, 0% used [0x000000000a590000,0x000000000a5c0810,0x000000000cf50000)  PSPermGen total 21248K, used 2982K [0x0000000005190000, 0x0000000006650000, 0x000000000a590000)  object space 21248K, 14% used [0x0000000005190000,0x0000000005479980,0x0000000006650000) |

虚拟参数为“**-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC**”,使用ParNew+CMS+Serial Old组合进行垃圾回收的日志

|  |
| --- |
| [Full GC (System) [CMS: 0K->194K(62656K), 0.0080796 secs] 4436K->194K(81792K), [CMS Perm : 2941K->2940K(21248K)], 0.0081589 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]  Heap  par new generation total 19136K, used 340K [0x0000000005540000, 0x0000000006a00000, 0x0000000006a00000)  eden space 17024K, 2% used [0x0000000005540000, 0x0000000005595290, 0x00000000065e0000)  from space 2112K, 0% used [0x00000000065e0000, 0x00000000065e0000, 0x00000000067f0000)  to space 2112K, 0% used [0x00000000067f0000, 0x00000000067f0000, 0x0000000006a00000)  concurrent mark-sweep generation total 62656K, used 194K [0x0000000006a00000, 0x000000000a730000, 0x000000000a940000)  concurrent-mark-sweep perm gen total 21248K, used 2981K [0x000000000a940000, 0x000000000be00000, 0x000000000fd40000) |

这四段GC日志中提炼出一些共性：

1. 日志的开头“GC”、“Full GC”表示这次垃圾收集器手机的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有Full， 则说明本次停止了其他所有工作线程。看到Full GC的写法是“Full GC(System)”，这说明是调用System.gc()方法所触发的GC。
2. “GC”中接下来的“DefNew”、“ParNew”、“PSYoungGen”、“CMS”表示的是老年代垃圾收集器的名称，“PSYoungGen”中的“PS”指的是“Parallel Scavenge”，它是Parallel收集器的全称。
3. 以第一个为例，方括号内部的“320->194K(2368K)”、“2242K->0K(2368K)”，指的是该区域已使用的容量->GC后该内存区域已使用的容量(该内存的总容量)。方括号外面的”310K->194K(7680K)”、“2242K->2241K(7680K)”则指的是GC前Java堆已使用的容量->GC后Java堆已使用的容量(Java堆总容量)。
4. 还以第一个为例，再往后“0.0269163secs”表示该内存区域GC所占用的时间，单位是秒。最后的“[Times:user=0.00 sys=0.00 real=0.03secs]”则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的钟墙时间。后面两个的区别是，钟墙时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间所以如果user或sys超过real是完全正常的。
5. “Heap”后面就列举出堆内存目前各个年代的区域的内存情况

## 触发GC的时机

最后总结一下什么时候会触发一次GC，个人经验看，有三种场景会触发GC：

1. 第一种场景应该很明显，当年轻代或者老年代满了，Java虚拟机无法再为新的对象分配内存空间了，那么Java虚拟机就会触发一次GC去回收掉那些已经不会再被使用到的对象
2. 手动调用System.gc()方法，通常这样会触发一次的Full GC以及至少一次的Minor GC
3. 程序运行的时候有一条低优先级的GC线程，它是一条守护线程，当这条线程处于运行状态的时候，自然就触发了一次GC了。这点也很好证明，不过要用到WeakReference的知识，后面 写WeakReference的时候会专门讲到这个。