## Java内存区域

运行时数据区

程序计数器

Program Counter Register

本地方法栈

Native Method Stack

虚拟机栈

VM Stack

堆

Heap

方法区

Method Area

如上图所示，虚拟机管理的内存分为五个区域。

1. 程序计数器

程序计数器是一小块内存空间，他是线程私有的，每个线程都会有一个线程计数器。程序计数器保存了下一条指令的地址，每个线程的执行指令不一样，故需要将该计数器设置成线程私有。

1. Java虚拟机栈

虚拟机栈也是线程私有。当一个线程执行到一个方法的时候，会创建一个栈帧，然后将该栈帧压入私有的虚拟机栈中。栈帧中保存的是局部变量表、操作数栈、动态链接、方法出口等。Java虚拟机栈的作用是为虚拟机执行Java方法服务。

1. 本地方法栈

本地方法栈于虚拟机栈的作用相似，他为虚拟机使用到的本地方法提供服务。同理，他也是线程私有的。

1. Java堆

Java堆是被所有线程共享的一块区域，也是最大的一块区域，所有的对象实例和数组都要在堆上面分配（不绝对）。Java堆是垃圾收集器管理的主要区域。

1. 方法区

方法区也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。

1. 直接内存

在使用NIO的时候会使用到非Java虚拟机管理的内存。直接内存的分配是不会受到Java堆大小的影响的，但是会受到本地内存大小的影响。

## Java对象

1. Java对象的创建
   1. 虚拟机读取到new指令时， 检查常量池中是否有对应的类型，如果没有，就将其加载到内存中，同时，对象需要内存的大小也确定下来了。
   2. 之后虚拟机需要为新生对象分配内存，如果内存空间很规整，空闲的在一起，使用过的在一起，那么就直接可以通过移动指针分配固定值的空间（指针碰撞）；如果内存空间不规整的，那么就需要建立一个空闲列表，根据该列表来分配空间。内存空间的规整与否和垃圾收集器有关系。此外，分配内存的时候可能有线程安全的问题，解决方案一种是通过CAS+失败重试的方式来保证原子性，还有就是在每个线程的本地线程分配缓存（TLAB）中分配内存，之后当TLAB使用完了后，才需要同步锁来进行TLAB的扩容。
   3. 将分配好的内存地址初始化为0。
   4. 设置对象头，例如对象属于哪个类型，对象的哈希码，对象的GC分代年龄等信息。
2. 对象的内存布局

对象在内存中的布局可以分为3块区域：对象头，实例数据，对齐填充。

1. 对象头一部分用于存储对象自身运行时数据，如hash码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID。对象头另一部分是类型指针，即对象指向它的类元数据。
2. 实例数据是对象真正存储的有效信息，也就是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的还是子类中定义的，都需要记录下来。
3. 对齐填充不是必须，仅仅起到占位符的作用。因为对象的起始地址必须是8的整数倍。
4. 对象的访问定位
5. 使用句柄的方式：对象移动的时候不需要修改引用的地址，只需要修改句柄指向的地址。
6. 使用直接地址的方式：速度快。

## 对象是否已经死亡

1. 引用计数算法

对于每一个对象，都可以给其添加一个引用计数器，每当有一个地方引用它的时候，计数器就加一，当引用失效时，计数器的值就减一。当计数器的值为0的时候，该对象就不可以再使用了。但这样会存在循环引用的问题。栈中的引用都置为null，而对于A和B的实例，A的属性会引用B实例，B属性会引用A实例，这样就形成了一种循环引用。

Instance A

堆

栈

Instance B

Instance A

1. 可达性分析算法

通过一系列的称为“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，当一个对象到GC Roots没有任何引用链相关联时，则证明此对象是不可用的。

可以作为GC Roots的对象包括下面几种：

* 虚拟机栈中引用的对象
* 方法区中类静态属性引用
* 方法区中常量引用的对象
* 本地方法栈中JNI引用的对象

## 垃圾收集算法

1. 标记-清除算法

该算法顾名思义，就是先将那些可以回收的对象标记为可以回收，然后再将那些标记过的对象进行清除。该算法有两个问题，一是效率问题：标记和清除两个过程的效率都不高，另一个问题是空间问题，清除后留下的空间是非连续的，所以如果有大对象的分配，可能就会触发另一次的垃圾回收机制。

1. 复制算法

复制算法将堆内存分为三块，Eden，From Survivor，To Survivor：每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活的对象一次性复制到另一块Survivor空间里，然后清理掉Eden和刚刚使用过的Survivor空间。默认的大小比值是8:1。如果另一块Survivor空间没有足够的空间来存放上一次新生代留下收集留下的存活对象，这些对象将直接通过分配担保机制进入老生代。

1. 标记-整理

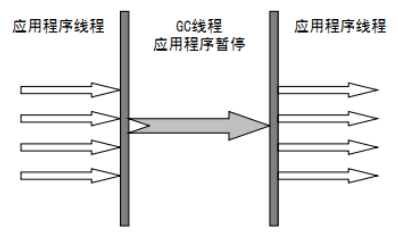
标记-整理和标记-清除方法类似，第一步都是要标记对象，第二步就不是直接对可回收对象进行清理了，而是让所有存活对象都向一端移动，然后直接清理掉端边界以外的内存。

1. 分代收集算法

分代收集的意思是，将内存分为新生代，老生代，由于新生代每次回收都会有很多对象死去，只有少量存活，所以需要使用复制算法，而对于老生代，因为其对象存活率高，没有额外的空间对它进行担保，就必须使用标记-清理或者标记-整理算法

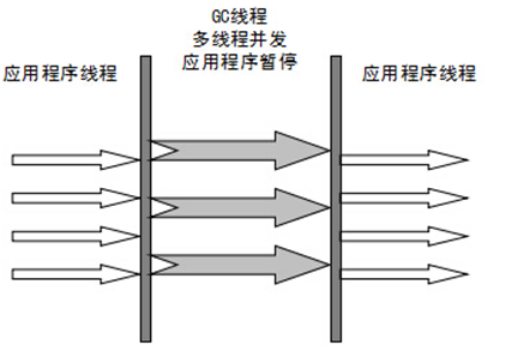
## 垃圾收集器

1. Serial收集器（新生代）



Serial收集器是最基本、发展历史最悠久的收集器，这个收集器是一个单线程的收集器，在进行垃圾收集的时候，必须要暂停其他所有的工作线程，直到它收集结束。Stop The World，单个CPU的环境中，Serial收集器没有线程交互的开销。

1. ParNew收集器（新生代）



ParNew收集器是Serial收集器的多线程版本。

1. Parallel Scavenge收集器（新生代）

Parallel Scavenge收集器是一个新生代收集器，它使用的是复制算法，并且是并行的多线程收集器，但是它的关注点与其他收集器不同，该收集器的目标是达到一个可控的吞吐量。吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）。吞吐量高则可以高效地利用CPU的时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

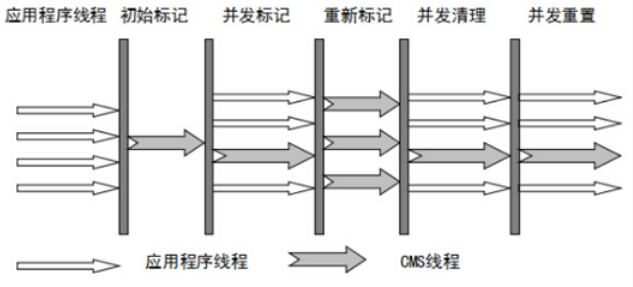
1. Serial Old收集器（老年代）

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

1. Parallel Old（老年代）

Parallel Old是Parallel Scavenge收集器的老年代版本，使用标记-整理算法

1. CMS收集器（老年代）



CMS收集器是一款以获取最短回收停顿时间为目标的收集器。CMS是基于标记-清除算法实现的。他的整个过程分为4个步骤，包括

* 初始标记：标记一下与GC Root能直接关联到的对象（需要stw）
* 并发标记：进行GC Roots Tracing，对堆中的对象进行可达性分析
* 重新标记：重新标记阶段是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记录（需要stw）
* 并发清除：对需要删除的对象进行清理

CMS的不足：

1. CMS对CPU资源非常敏感。CMS默认开启的回收线程数是（Cpu数量+3）/4，如果CPU数量只有两个，那么就需要有一半的CPU资源去执行收集器线程。
2. CMS收集器无法处理浮动垃圾，由于CMS并发请管理阶段用户线程还在运行，伴随着程序的运行自然会有新的垃圾产生，这一部分垃圾出现在标记过程之后，CMS无法再当次收集中处理掉它们。
3. CMS是一款基于“标记-清理”算法实现的收集器，空间碎片过多，可能会导致无法分配内存的问题。
4. G1收集器（新生代，老年代）

运作步骤：

* 初始化标记：仅仅将和GC\_root关联的进行标记（stw）
* 并发标记：对堆中的对象进行可达性分析
* 最终标记：修正在并发标记期间因为用户程序继续运作而导致标记产生变动的那一部分。（STW）
* 筛选回收：对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来指定回收计划。（STW）

## 内存分配策略

* 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（达到MaxTenuringThreshold）
* 当Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代

## 类加载的时机

链接

加载

卸载

初始化

使用

准备

解析

验证

**加载**

加载阶段需要做下面三件事：

1. 通过一个类的全限定名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

**初始化**

初始化阶段是类加载过程的最后一步，在**准备阶段**，变量已经根据系统的要求赋值过一次初始值，而初始化阶段则根据程序制定的主管计划去初始化类变量和其他资源。初始化阶段就是执行类构造器<cinit>()方法的过程，**<clnit>方法是由编译器收集类中所有类变量的赋值动作和静态语句块（static{}）**中的语句合并产生，编译器收集的顺序是由语句在源文件中出现的顺序决定的。

## 类加载器

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其Java虚拟机中的唯一性，**比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义**。

启动类加载器，它是虚拟机自带的，该类加载器负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中，并且能够被虚拟机识别的类库加载到虚拟机内存中。

拓展加载器，它负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用拓展类加载器。

应用程序加载器，一般称它为系统类加载器，它负责加载用户类路径下的所指定的类库，一般情况下这个就是程序中默认的类加载器。

应用程序类加载器

自定义类加载器

自定义类加载器

扩展类加载器

启动类加载器

双亲委派：如果一个类夹杂器收到了加载类的请求，那么他首先不会自己去尝试加载这个类，而是把请求委托给父类加载器去完成，每个层次的类加载器都是如此。

例如java.lang.Object类，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此object类在程序的各种类加载器环境中都是同一个类。