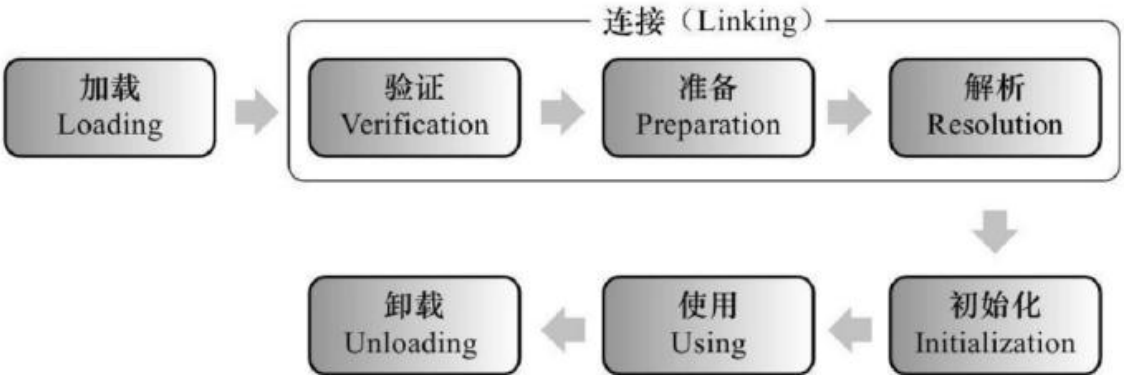
1 JVM 类加载机制

1.1 JVM 类加载的五个阶段

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化



1.1.1 加载

加载是类加载过程中的一个阶段，这个阶段会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入口。注意这里不一定非得要从一个 Class 文件获取，这里既 可以从 ZIP 包中读取（比如从 jar 包和 war 包中读取），也可以在运行时计算生成（动态代理）， 也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。

1.1.2 验证

这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并 且不会危害虚拟机自身的安全。

1.1.3 准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使 用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

public static int v = 8080;

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是 程序被编译后，存放于类构造器<client>方法之中。

但是注意如果声明为：

public static final int v = 8080;

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

1.1.4 解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中 的：

1. CONSTANT_Class_info
2. CONSTANT_Field_info
3. CONSTANT_Method_info 等类型的常量。

1.1.4.1 符号引用

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟 机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引 用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中。

1.1.4.2 直接引用

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有 了直接引用，那引用的目标必定已经在内存中存在。

1.1.5 初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载 器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

1.1.6 类构造器<client>

初始化阶段是执行类构造器<client>方法的过程。<client>方法是由编译器自动收集类中的类变 量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子<client>方法执行之前，父类 的<client>方法已经执行完毕，如果一个类中没有对静态变量赋值也没有静态语句块，那么编译 器可以不为这个类生成<client>()方法。

注意以下几种情况不会执行类初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触 发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化。
5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初 始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

1.2 类加载器

虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类，JVM 提 供了 3 种类

加载器：

1.2.1 启动类加载器(Bootstrap ClassLoader)

1. 负责加载 JAVA_HOME\lib 目录中的，或通过-Xbootclasspath 参数指定路径中的，且被 虚拟机认可（按文件名识别，如 rt.jar）的类。

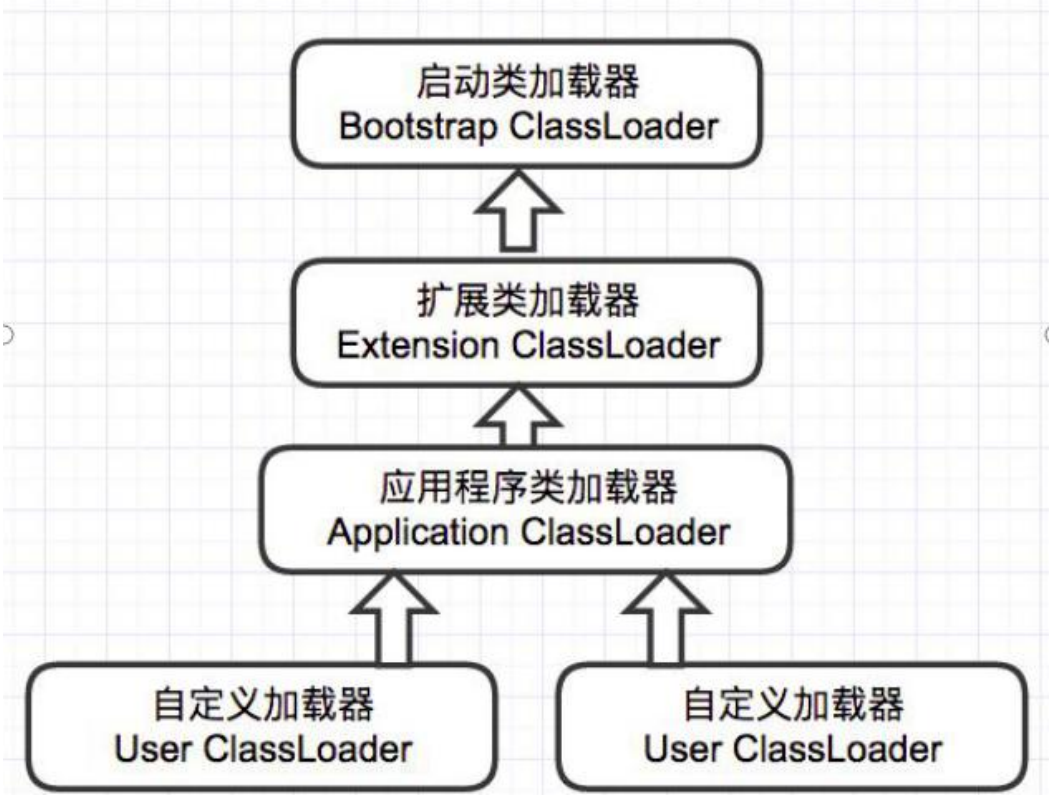
1.2.2 扩展类加载器(Extension ClassLoader)

2. 负责加载 JAVA_HOME\lib\ext 目录中的，或通过 java.ext.dirs 系统变量指定路径中的类 库。

1.2.3 应用程序类加载器(Application ClassLoader):

3. 负责加载用户路径（classpath）上的类库。

JVM 通过双亲委派模型进行类的加载，当然我们也可以通过继承 java.lang.ClassLoader 实现自定义的类加载器。

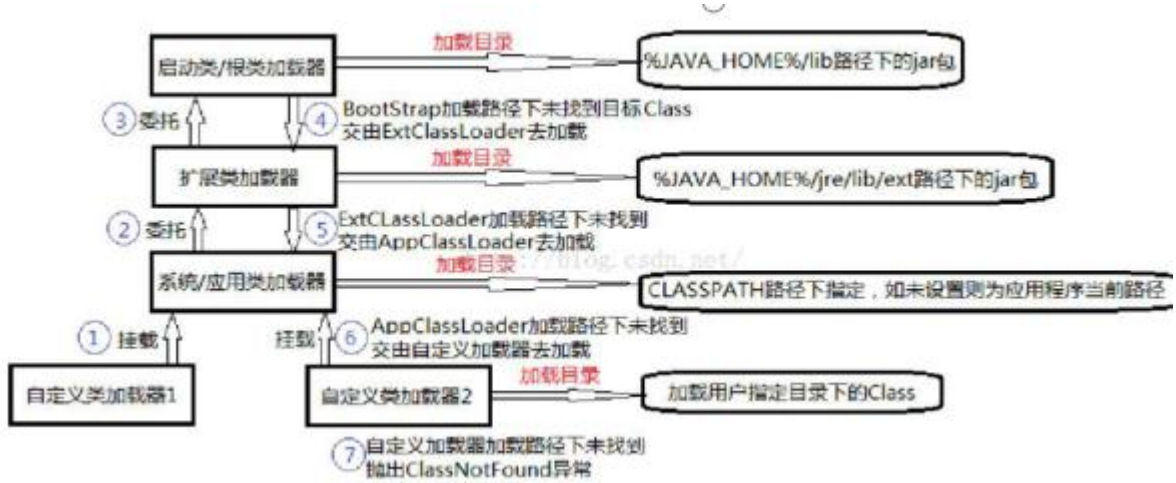


1.3 双亲委派

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父 类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中， 只有当父类加载器反馈自己无法完成

这个请求的时候 （在它的加载路径下没有找到所需加载的 Class），子类加载器才会尝试自己去加载。

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载 器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载 器最终得到的都是同样一个 Object 对象。



1.4 OSGI（动态模型系统）

OSGi(Open Service Gateway Initiative)，是面向 Java 的动态模型系统，是 Java 动态化模块化系 统的一系列规范。

1.4.1 动态改变构造

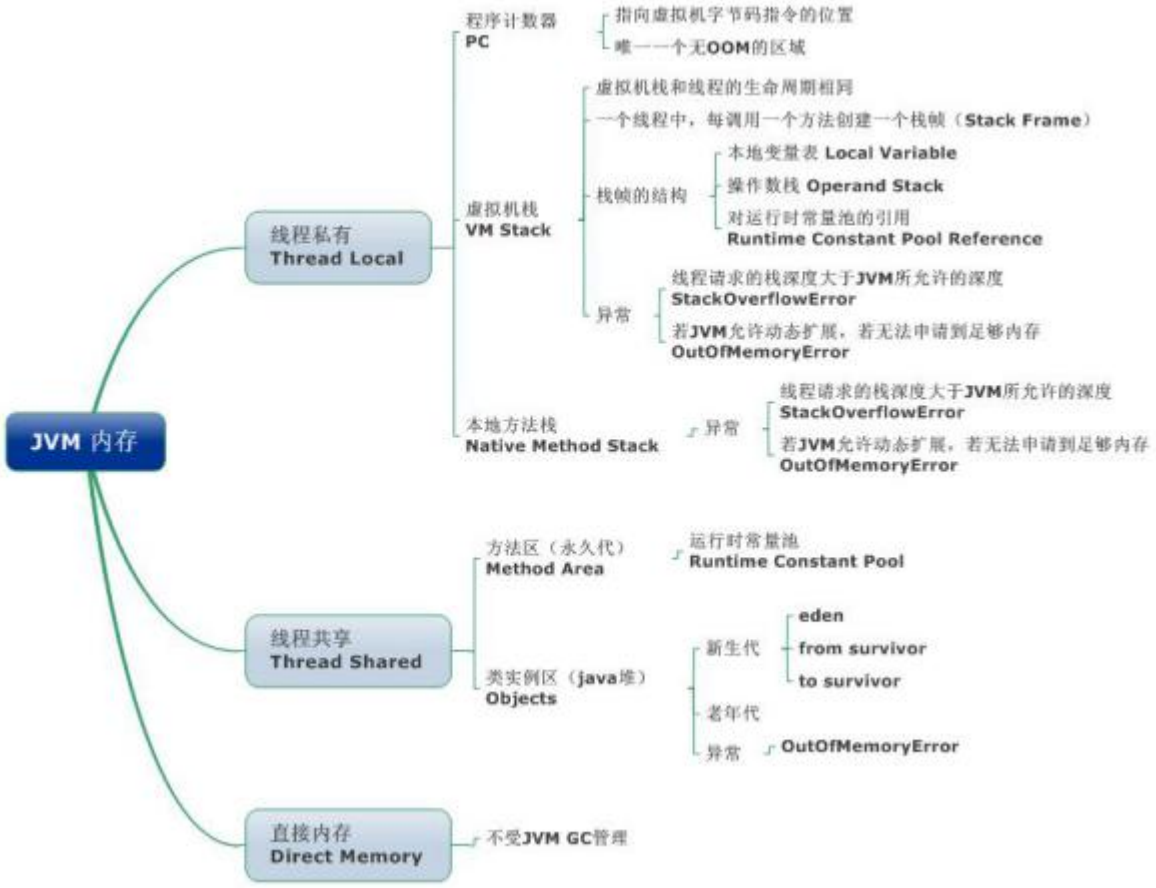
OSGi 服务平台提供在多种网络设备上无需重启的动态改变构造的功能。为了最小化耦合度和促使 这些耦合度可管理，OSGi 技术提供一种面向服务的架构，它能使这些组件动态地发现对方。

1.4.2 模块化编程与热插拔

OSGi 旨在为实现 Java 程序的模块化编程提供基础条件，基于 OSGi 的程序很可能可以实现模块级 的热插拔功能，当程序升级更新时，可以只停用、重新安装然后启动程序的其中一部分，这对企 业级程序开发来说是非常具有诱惑力的特性。

OSGi 描绘了一个很美好的模块化开发目标，而且定义了实现这个目标的所需要服务与架构，同时 也有成熟的框架进行实现支持。但并非所有的应用都适合采用 OSGi 作为基础架构，它在提供强大 功能同时，也引入了额外的复杂度，因为它不遵守了类加载的双亲委托模型。

2 JVM 内存区域

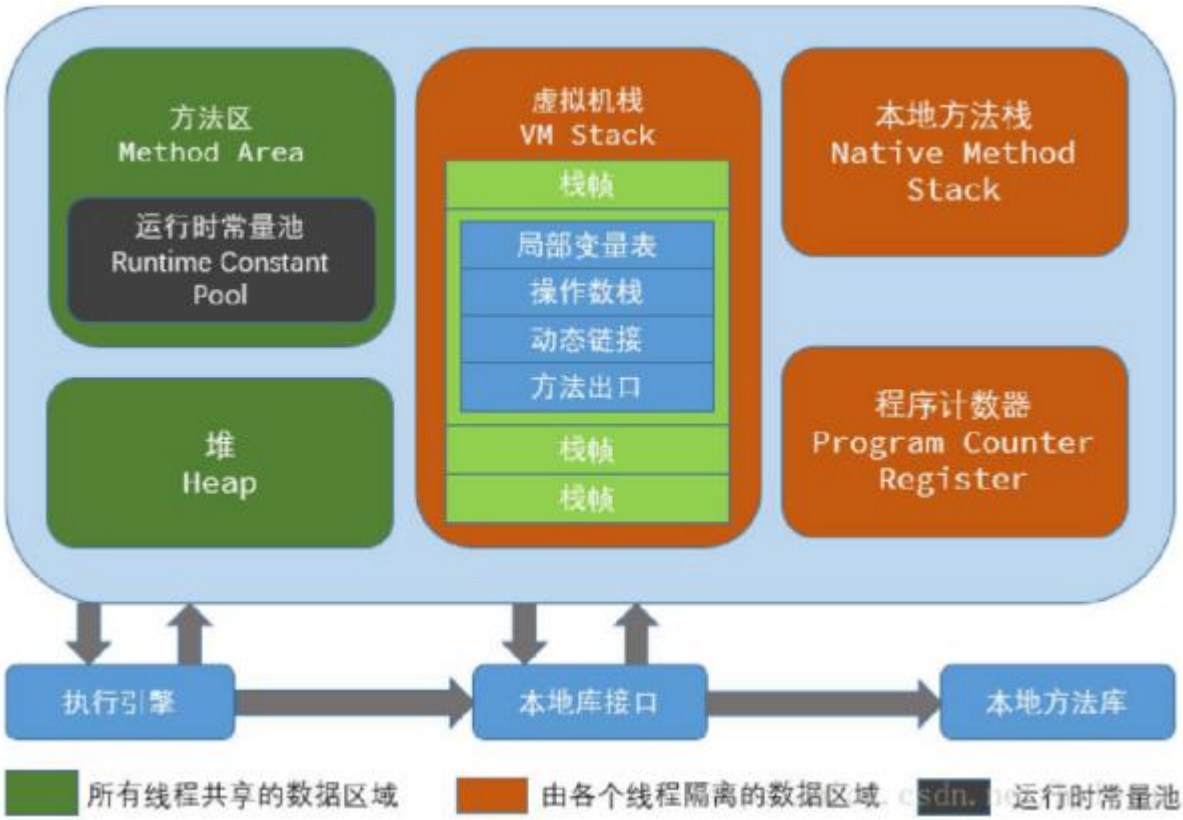


JVM 内存区域主要分为线程私有区域【程序计数器、虚拟机栈、本地方法区】、线程共享区 域【JAVA 堆、方法区】、直接内存。

线程私有数据区域生命周期与线程相同, 依赖用户线程的启动/结束 而 创建/销毁(在 Hotspot VM 内, 每个线程都与操作系统的本地线程直接映射, 因此这部分内存区域的存/否跟随本地线程的 生/死对应)。

线程共享区域随虚拟机的启动/关闭而创建/销毁。

直接内存并不是 JVM 运行时数据区的一部分, 但也会被频繁的使用: 在 JDK 1.4 引入的 NIO 提 供了基于 Channel 与 Buffer 的 IO 方式, 它可以使用 Native 函数库直接分配堆外内存, 然后使用 DirectByteBuffer 对象作为这块内存的引用进行操作(详见: Java I/O 扩展), 这样就避免了在 Java 堆和 Native 堆中来回复制数据, 因此在一些场景中可以显著提高性能。



2.1 程序计数器(线程私有)

一块较小的内存空间, 是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的 程序计数器，这类内存也称为“线程私有”的内存。

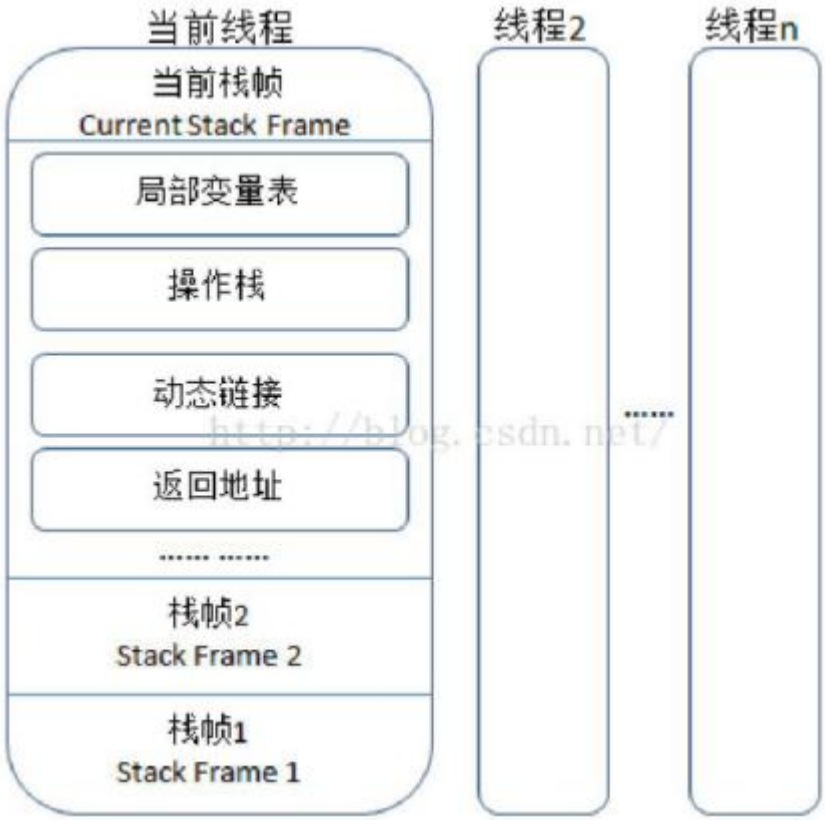
正在执行 java 方法的话，计数器记录的是虚拟机字节码指令的地址（当前指令的地址）。如 果还是 Native 方法，则为空。

这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域。

2.2 虚拟机栈(线程私有)

是描述 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。每一个方法从调用直至执行完成 的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

栈帧（ Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接 (Dynamic Linking)、 方法返回值和异常分派（ Dispatch Exception）。栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异 常）都算作方法结束。



2.3 本地方法区(线程私有)

本地方法区和 Java Stack 作用类似, 区别是虚拟机栈为执行 Java 方法服务, 而本地方法栈则为 Native 方法服务,如果一个 VM 实现使用 C-linkage 模型来支持 Native 调用, 那么该栈将会是一个 C 栈，但 HotSpot VM 直接就把本地方法栈和虚拟机栈合二为一。

2.4 堆（Heap-线程共享）-运行时数据区

是被线程共享的一块内存区域，创建的对象和数组都保存在 Java 堆内存中，也是垃圾收集器进行 垃圾收集的最重要的内存区域。由于现代 VM 采用分代收集算法, 因此 Java 堆从 GC 的角度还可以 细分为: 新生代(Eden 区、From Survivor 区和 To Survivor 区)和老年代。

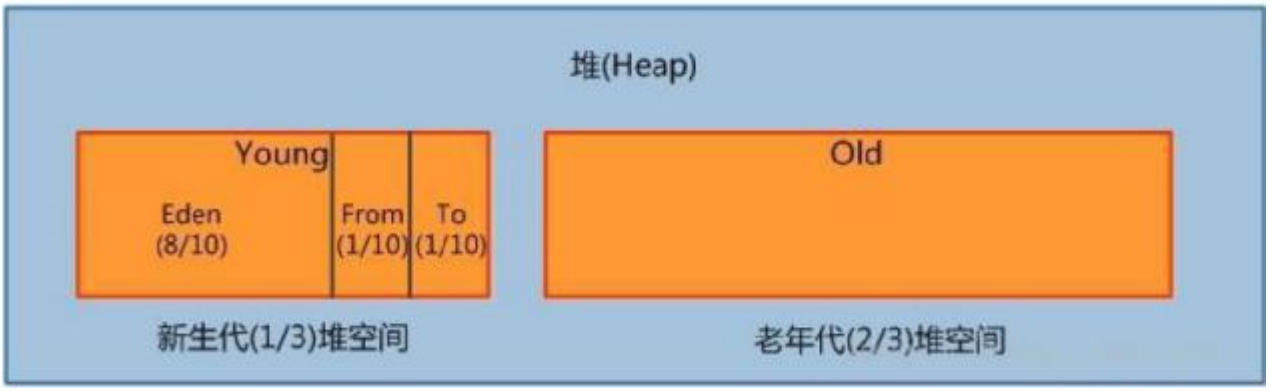
2.5 方法区/永久代（线程共享）

即我们常说的永久代(Permanent Generation), 用于存储被 JVM 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. HotSpot VM 把 GC 分代收集扩展至方法区, 即使用 Java 堆的永久代来实现方法区, 这样 HotSpot 的垃圾收集器就可以像管理 Java 堆一样管理这部分内存, 而不必为方法区开发专门的内存管理器(永久带的内存回收的主要目标是针对常量池的回收和类型 的卸载, 因此收益一般很小)。

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版 本、字段、方法、接口等描述等信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加 载后存放到方法区的运行时常量池中。 Java 虚拟机对 Class 文件的每一部分（自然也包括常量 池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会 被虚拟机认可、装载和执行。

3 JVM 运行时内存

Java 堆从 GC 的角度还可以细分为: 新生代(Eden 区、From Survivor 区和 To Survivor 区)和老年 代。



3.1 新生代

是用来存放新生的对象。一般占据堆的 1/3 空间。由于频繁创建对象，所以新生代会频繁触发 MinorGC 进行垃圾回收。新生代又分为 Eden 区、ServivorFrom、ServivorTo 三个区。

3.1.1 Eden 区

Java 新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老 年代）。当 Eden 区内存不够的时候就会触发 MinorGC，对新生代区进行 一次垃圾回收。

3.1.2 ServivorFrom

上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

3.1.3 ServivorTo

保留了一次 MinorGC 过程中的幸存者。

3.1.4 MinorGC 的过程（复制->清空->互换）

MinorGC 采用复制算法。

1：eden、servicorFrom 复制到 ServicorTo，年龄+1

首先，把 Eden 和 ServivorFrom 区域中存活的对象复制到 ServicorTo 区域（如果有对象的年 龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不 够位置了就放到老年区）；

2：清空 eden、servicorFrom

然后，清空 Eden 和 ServicorFrom 中的对象；

3：ServicorTo 和 ServicorFrom 互换

最后，ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

3.2 老年代

主要存放应用程序中生命周期长的内存对象。

老年代的对象比较稳定，所以 MajorGC 不会频繁执行。在进行 MajorGC 前一般都先进行 了一次 MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足 够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。

MajorGC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没 有标记的对象。MajorGC 的耗时比较长，因为要扫描再回收。MajorGC 会产生内存碎片，为了减 少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的 时候，就会抛出 OOM（Out of Memory）异常。

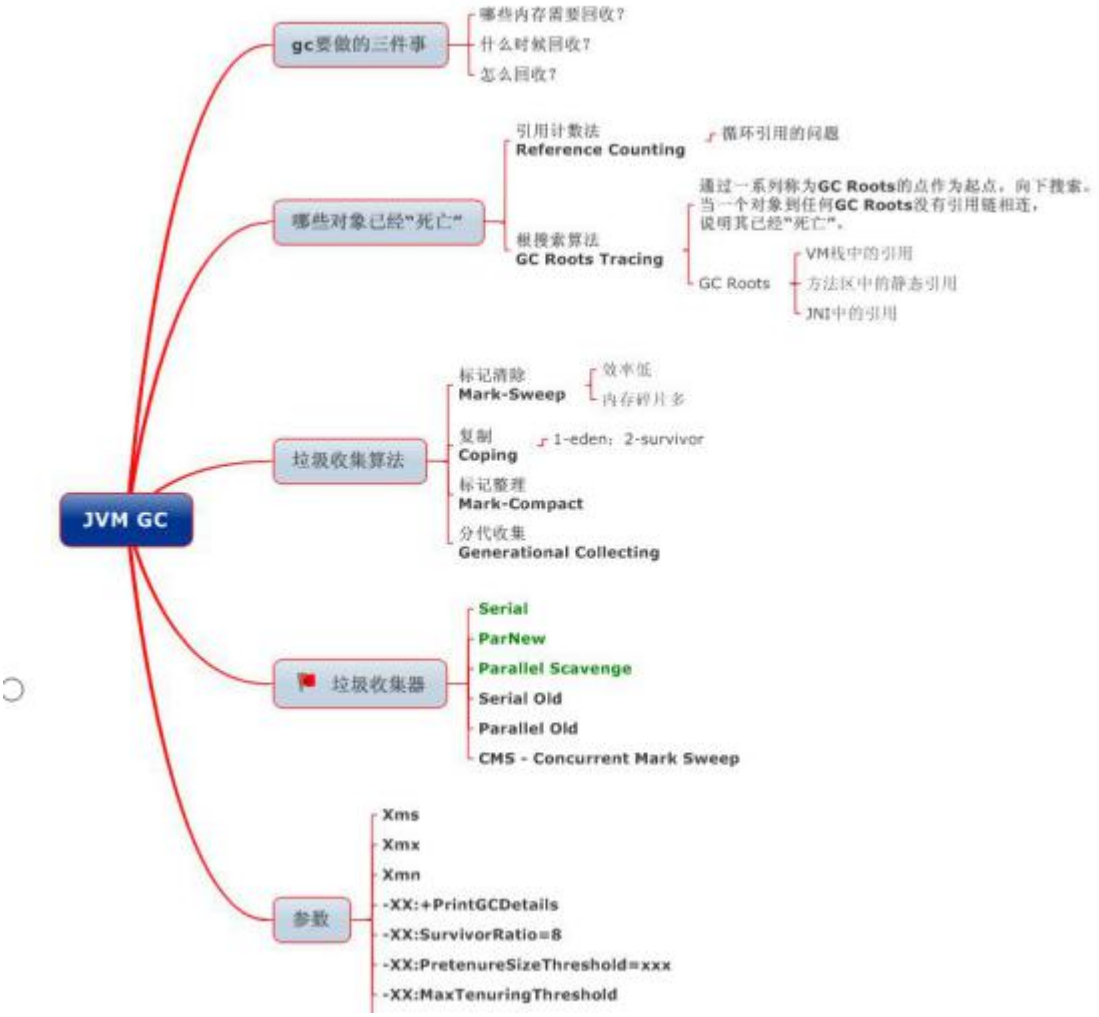
3.3 永久代

指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被 放入永久区域，它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这 也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

3.3.1 JAVA8 与元数据

在 Java8 中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间 的本质和永久代类似，元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用 本 地 内 存 。因此，默认情况下，元空间的大小仅受本地内存限制。类 的 元 数 据 放 入 native memory, 字 符串 池 和 类的 静 态 变 量 放入 java 堆中， 这 样 可 以 加 载多 少类 的 元 数 据 就 不 再由 MaxPermSize 控制, 而由系统的实际可用空间
来控制。

4 垃圾回收与算法



4.1 如何确定垃圾

4.1.1 引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单 的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关 联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收 对象。

4.1.2 可达性分析

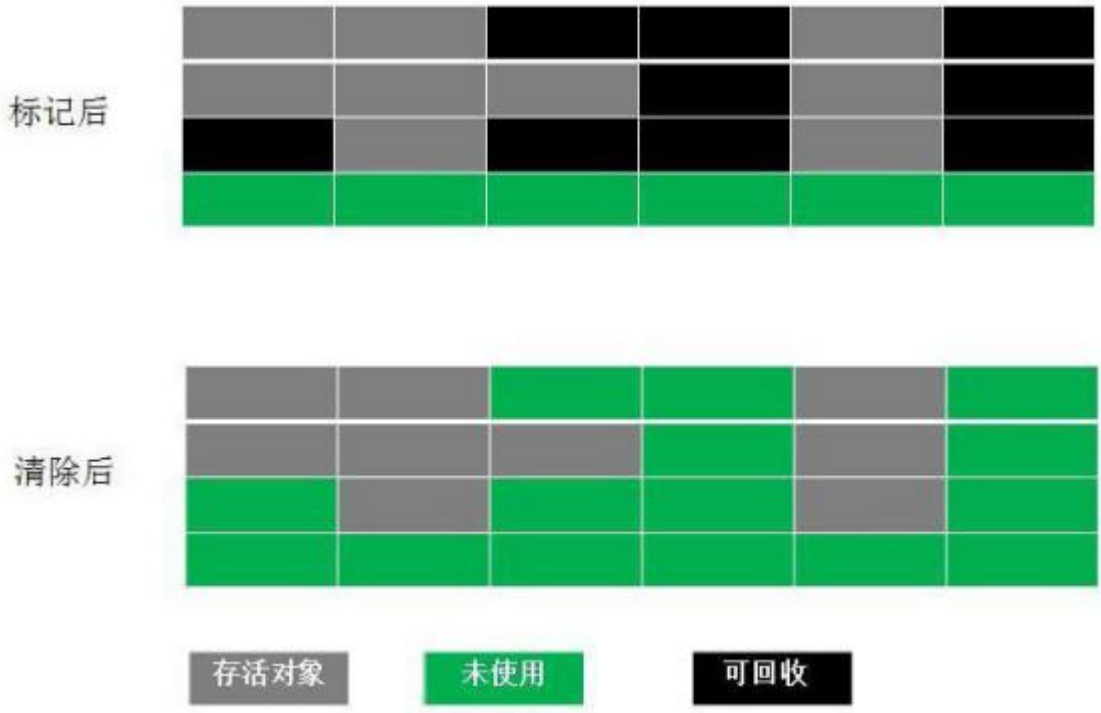
为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots”对象作为起点

搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。

要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记 过程。两次标记后仍然是可回收对象，则将面临回收。

4.2 标记清除算法（Mark-Sweep）

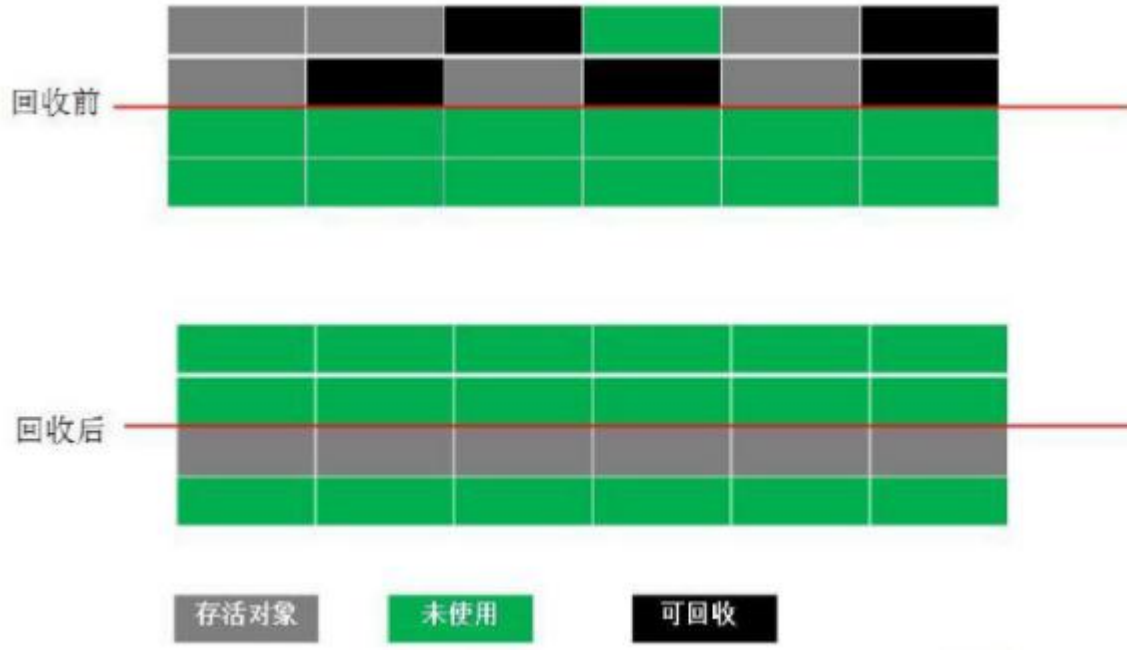
最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清 除阶段回收被标记的对象所占用的空间。如图



从图中我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可 利用空间的问题。

4.3 复制算法（copying）

为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小 的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用 的内存清掉，如图：



这种算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原 本的一半。且存活对象增多的话，Copying 算法的效率会大大降低。

4.4 标记整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同，标记后不是清 理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：

4.5 分代收集算法

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存 划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(Young Generation)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃 圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

4.5.1 新生代与复制算法

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要 回收大部分对象，即要复制的操作比较少，但通常并不是按照 1：1 来划分新生代。一般将新生代 划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用 Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另 一块 Survivor 空间中。

4.5.2 老年代与标记复制算法

而老年代因为每次只回收少量对象，因而采用 Mark-Compact 算法。

1. JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)，它用来存储 class 类， 常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。
2. 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目 前存放对象的那一块)，少数情况会直接分配到老生代。
3. 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后，Eden Space 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 From Space 进行清理。
4. 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。
5. 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。
6. 当对象在 Survivor 区躲过一次 GC 后，其年龄就会+1。默认情况下年龄到达 15 的对象会被 移到老生代中。

4.6 GC 分代收集算法 VS 分区收集算法

4.6.1 分代收集算法

当前主流 VM 垃圾收集都采用”分代收集”(Generational Collection)算法，这种算法会根据 对象存活周期的不同将内存划分为几块，如 JVM 中的 新生代、老年代、永久代，这样就可以根据 各年代特点分别采用最适当的 GC 算
法

4.6.1.1 在新生代-复制算法

每次垃圾收集都能发现大批对象已死，只有少量存活. 因此选用复制算法，只需要付出少量 存活对象的复制成本就可以完成收集.

4.6.1.2 在老年代-标记整理算法

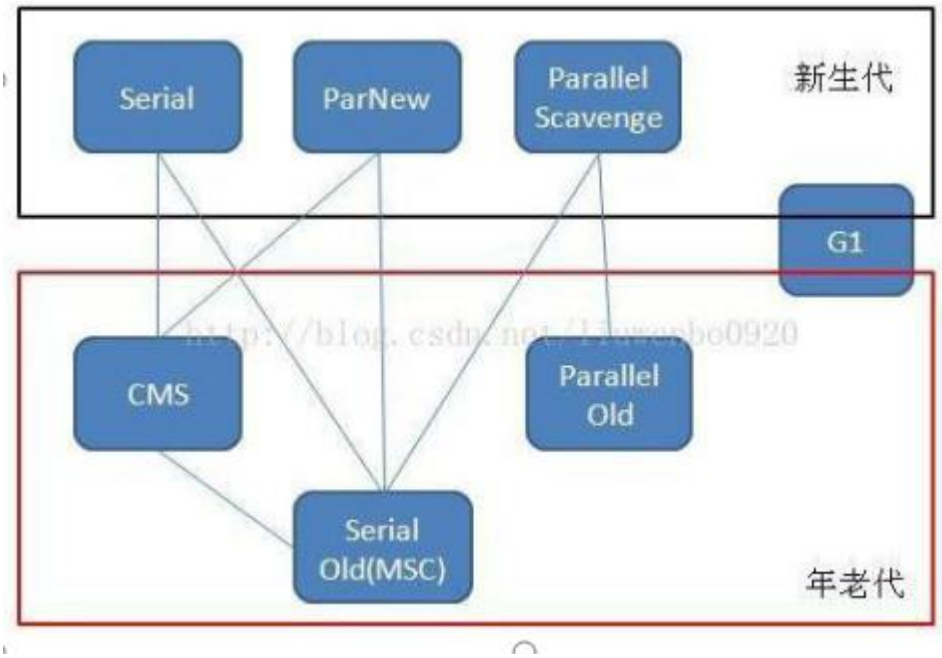
因为对象存活率高、没有额外空间对它进行分配担保，就必须采用“标记—清理”或“标 记—整理”算法来进行回收，不必进行内存复制，且直接腾出空闲内存.

4.6.2 分区收集算法

分区算法则将整个堆空间划分为连续的不同小区间，每个小区间独立使用，独立回收. 这样做的 好处是可以控制一次回收多少个小区间 ，根据目标停顿时间，每次合理地回收若干个小区间(而不是 整个堆)，从而减少一次 GC 所产生的停顿。

4.7 GC 垃圾收集器

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法； 年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和年老代分别提供了多种不 同的垃圾收集器，JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器如下：



4.7.1 Serial 垃圾收集器（单线程、复制算法）

Serial（英文连续）是最基本垃圾收集器，使用复制算法，曾经是 JDK1.3.1 之前新生代唯一的垃圾 收集器。Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工 作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限 定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

4.7.2 ParNew 垃圾收集器（Serial+多线程）

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃 圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也 要暂停所有其他的工作线程。ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限 制垃圾收集器的线程数。【Parallel：平行的】

ParNew 虽然是除了多线程外和 Serial 收集器几乎完全一样，但是 ParNew 垃圾收集器是很多 java 虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

4.7.3 Parallel Scavenge 收集器（多线程复制算法、高效）

Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃 圾收集器，它重点关注的是程序达到一个可控制的吞吐量（Thoughput，CPU 用于运行用户代码 的时间/CPU 总消耗时间，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)）， 高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而 不需要太多交互的任务。自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个 重要区别。

4.7.4 Serial Old 收集器（单线程标记整理算法 ）

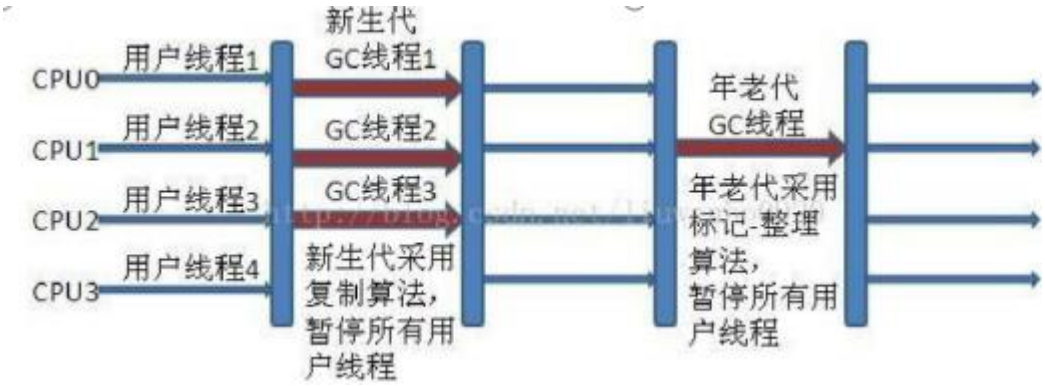
Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法， 这个收集器也主要是运行在 Client 默认的 java 虚拟机默认的年老代垃圾收集器。

在 Server 模式下，主要有两个用途：

1. 在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。
2. 作为年老代中使用 CMS 收集器的后备垃圾收集方案。

新生代 Serial 与年老代 Serial Old 搭配垃圾收集过程图：

新生代 Parallel Scavenge 收集器与 ParNew 收集器工作原理类似，都是多线程的收集器，都使 用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。新生代 Parallel Scavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：



4.7.5 Parallel Old 收集器（多线程标记整理算法）

Parallel Old 收集器是 Parallel Scavenge 的年老代版本，使用多线程的标记-整理算法，在 JDK1.6 才开始提供。在 JDK1.6 之前，新生代使用 ParallelScavenge 收集器只能搭配年老代的 Serial Old 收集器，只 能保证新生代的吞吐量优先，无法保证整体的吞吐量，Parallel Old 正是为了在年老代同样提供吞 吐量优先的垃圾收集器，如果系统对吞吐量要求比较高，可以优先考虑新生代 Parallel Scavenge 和年老代 Parallel Old 收集器的搭配策略。

4.7.6 CMS 收集器（多线程标记清除算法）

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾 回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。

最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。

CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：

4.7.6.1 初始标记

只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

4.7.6.2 并发标记

进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

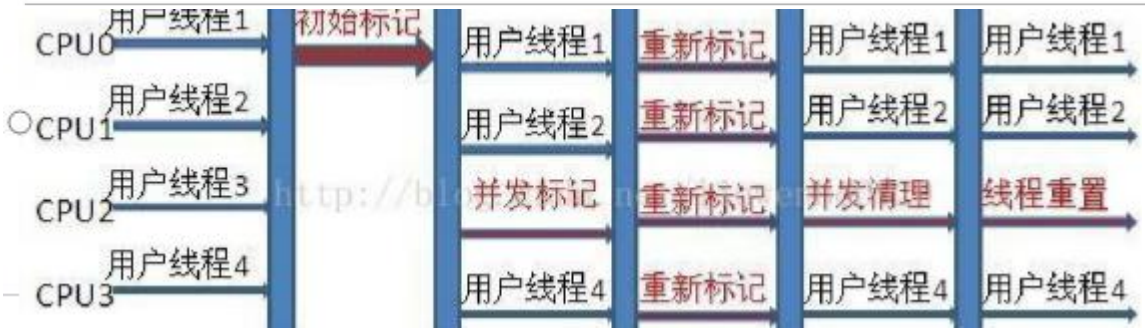
4.7.6.3 重新标记

为了修正在并发标记期间，因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，仍然需要暂停所有的工作线程。

4.7.6.4 并发清除

清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并 发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作， 所以总体上来看 CMS 收集器的内存回收和用户线程是一起并发地执行。

CMS 收集器工作过程：



4.7.7 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器，G1 收 集器两个最突出的改进是：

1. 基于标记-整理算法，不产生内存碎片。
2. 可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域 的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾 最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收 集效率。

5 JVM 参数详解

5.1 通用 JVM 参数

5.1.1 -server

如果不配置该参数，JVM 会根据应用服务器硬件配置自动选择不同模式，server 模式启动比较慢，但是运行期速度得到了优化，适合于服务器端运行的 JVM。

5.1.2 -client

启动比较快，但是运行期响应没有 server 模式的优化，适合于个人 PC 的服务开发和测试。

5.1.3 -Xmx

设置 java heap 的最大值，默认是机器物理内存的 1/4。这个值决定了最多可用的 Java 堆内存：分配过少就会在应用中需要大量内存作缓存或者临时对象时出现 OOM（Out Of Memory）的问题；如果分配过大，那么就会因 PermSize 过小而引起的另外一种 Out Of Memory。所以如何配置还是根据运行过程中的分析和计算来确定，如果不

能确定还是采用默认的配置。

5.1.4 -Xms

设置 Java 堆初始化时的大小，默认情况是机器物理内存的 1/64。这个主要是根据应用启动时消耗的资源决定，分配少了申请起来会降低运行速度，分配多了也浪费。

5.1.5 -XX:PermSize

初始化永久内存区域大小。永久内存区域全称是 Permanent Generation space，是指内存的永久保存区域，程序运行期不对 PermGen space 进行清理，所以如果你的 APP 会 LOAD 很多 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 pre compile 的时候。 如果你的 WEB APP 下用了大量的第三方 jar，其大小超过了 jvm 默认的 PermSize 大小(4M)那么就会产生此错误信息了。

5.1.6 -XX:MaxPermSize

设置永久内存区域最大大小。

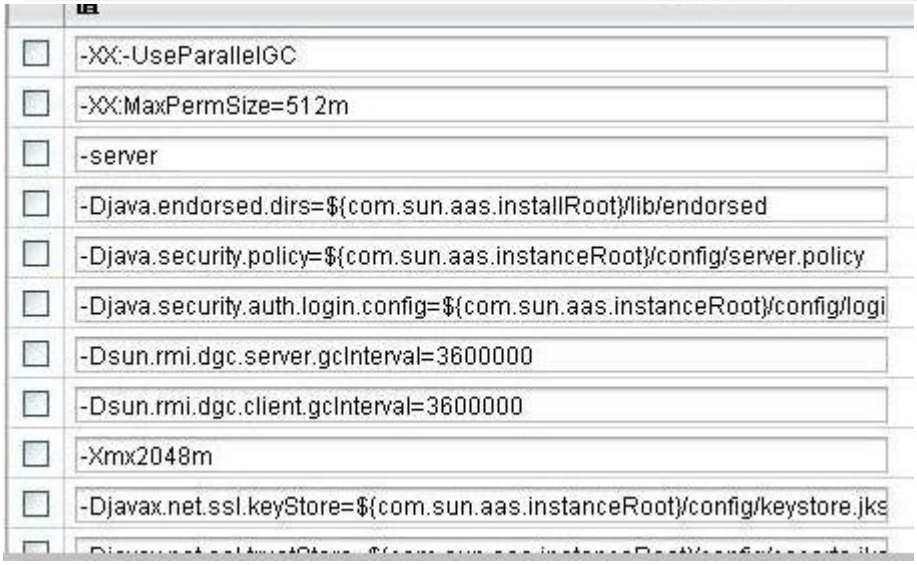
5.1.7 -Xmn

直接设置青年代大小。整个 JVM 可用内存大小=青年代大小 + 老年代大小 + 持久代大小 。持久代一般固定大小为 64m，所以增大年轻代后，将会减小老年代大小。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。

按照 Sun 的官方设置比例，则上面的例子中年轻代的大小应该为 2048*3/8=768M。

5.1.8 -XX:NewRatio

控制默认的 Young 代的大小，例如，设置-XX:NewRatio=3 意味着 Young 代和老年代的比率是 1:3。换句话说，Eden 和 Survivor 空间总和是整个堆大小的 1/4。



如图中的实际设置，-XX:NewRatio=2，-Xmx=2048，则年轻代和老年代的分配比例为 1:2，即年轻代的大小为 682M，而老年代的大小为 1365M。查看实际系统的 jvm 监控结果为：

内存池名称: Tenured Gen

Java 虚拟机最初向操作系统请求的内存量: 3,538,944 字节

Java 虚拟机实际能从操作系统获得的内存量: 1,431,699,456 字节

Java 虚拟机可从操作系统获得的最大内存量: 1,431,699,456 字节。请注意，并不一定能获得该内存量。

Java 虚拟机此时使用的内存量: 1,408,650,472 字节

即：1,408,650,472 字节=1365M，证明了上面的计算是正确的。

5.1.9 -XX:SurvivorRatio

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。越大的 survivor 空间可以允许短期对象尽量在年青代消亡；如果 Survivor 空间太小，Copying 收集将直接将其转移到老年代中，这将加快老年代的空间使用速度，引发频繁的完全垃圾回收。如下图：

SurvivorRatio 的值设为 3，Xmn 为 768M，则每个 Survivor 空间的大小为 768M/5=153.6M。

5.1.10 -XX:NewSize

为了实现更好的性能，您应该对包含短期存活对象的池的大小进行设置，以使该池中的对象的存活时间不会超过一个垃圾回收循环。新生成的池的大小由 NewSize 和 MaxNewSize 参数确定。通过这个选项可以设置 Java 新对象生产堆内存。在通常情况下这个选项的数值为 1024 的整数倍并且大于 1MB。这个值的取值规则为，一般情况下

这个值-XX:NewSize 是最大堆内存（maximum heap size）的四分之一。增加这个选项值的大小是为了增大较大数量

的短生命周期对象。增加 Java 新对象生产堆内存相当于增加了处理器的数目。并且可以并行地分配内存，但是请注意内存的垃圾回收却是不可以并行处理的。作用跟-XX:NewRatio 相似， -XX:NewRatio 是设置比例而-XX:NewSize 是设置精确的数值。

5.1.11 -XX:MaxNewSize

通过这个选项可以设置最大 Java 新对象生产堆内存。通常情况下这个选项的数值为 1 024 的整数倍并且大于

1MB，其功用与上面的设置新对象生产堆内存-XX:NewSize 相同。一般要将 NewSize 和 MaxNewSize 设成一致。

5.1.12 -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，直接进入老年代。对于老年代比较多的应用，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，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率。

如下图：

```
<jvm-options>-XX:MaxTenuringThreshold=5</jvm-options>
```

-XX:MaxTenuringThreshold 参数被设置成 5，表示对象会在 Survivor 区进行 5 次复制后如果还没有被回收才会被复制到老年代。

5.1.13 -XX:GCTimeRatio

设置垃圾回收时间占程序运行时间的百分比。该参数设置为 n 的话，则垃圾回收时间占程序运行时间百分比的公式为 1/(1+n) ，如果 n=19 表示 java 可以用 5%的时间来做垃圾回收，1/(1+19)=1/20=5%。

5.1.14 -XX:TargetsurvivorRatio

该值是一个百分比，控制允许使用的救助空间的比例，默认值是 50。该参数设置较大的话可提高对 survivor 空间的使用率。当较大的堆栈使用较低的 SurvivorRatio 时，应增加该值到 80 至 90，以更好利用救助空间。

5.1.15 -Xss

设置每个线程的堆栈大小，**根据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。当这个选项被设置的较大（>2MB）时将会在很大程度上降低系统的性能。因此在设置这个值时应该格外小心，调整后要注意观察系统的性能，不断调整以期达到最优。

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。

5.1.16 -Xnoclassgc

这个选项用来取消系统对特定类的垃圾回收。它可以防止当这个类的所有引用丢失之后，这个类仍被引用时不会再一次被重新装载，因此这个选项将增大系统堆内存的空间。禁用类垃圾回收，性能会高一点；

5.2 串行收集器参数

5.2.1 -XX:+UseSerialGC:

设置串行收集器 。

5.3 并行收集器参数

5.3.1 -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，年轻代使用并行收集，而老年代仍旧使用串行收集。采用了多线程并行管理和回收垃圾对象，提高了回收效率，提高了服务器的吞吐量，适合于多处理器的服务器。

5.3.2 -XX:ParallelGCThreads

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。

5.3.3 -XX:+UseParallelOldGC：

采用对于老年代并发收集的策略，可以提高收集效率。JDK6.0 支持对老年代并行收集。

5.3.4 -XX:MaxGCPauseMillis

设置每次年轻代并行收集最大暂停时间，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。

5.3.5 -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响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。

5.4 并发收集器参数

5.4.1 -XX:+UseConcMarkSweepGC

指定在 老年代 使用 concurrent cmark sweep gc。gc thread 和 app thread 并行 (在 init-mark 和 remark 时 pause app thread)。app pause 时间较短 , 适合交互性强的系统 , 如 web server。它可以并发执行收集操作，降低应用停止时间，同时它也是并行处理模式，可以有效地利用多处理器的系统的多进程处理。

5.4.2 -XX:+UseParNewGC

指定在 New Generation 使用 parallel collector, 是 UseParallelGC 的 gc 的升级版本 , 有更好的性能或者优点 , 可以和 CMS gc 一起使用

5.4.3 -XX:+UseCMSCompactAtFullCollection：

打开对老年代的压缩。可能会影响性能，但是可以消除碎片,在 FULL GC 的时候， 压缩内存， CMS 是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。

5.4.4 -XX:+CMSIncrementalMode

设置为增量模式。适用于单 CPU 情况

5.4.5 -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。

5.4.6 -XX:+CMSClassUnloadingEnabled

使 CMS 收集持久代的类，而不是 fullgc

5.4.7 -XX:+CMSPermGenSweepingEnabled

使 CMS 收集持久代的类，而不是 fullgc。

5.4.8 -XX:-CMSParallelRemarkEnabled

在使用 UseParNewGC 的情况下，尽量减少 mark 的时间。

5.4.9 -XX:CMSInitiatingOccupancyFraction

说明老年代到百分之多少满的时候开始执行对老年代的并发垃圾回收（CMS），这个参数设置有很大技巧，基本上满足公式：

(Xmx-Xmn)*(100-CMSInitiatingOccupancyFraction)/100>=Xmn

时就不会出现 promotion failed。在我的应用中 Xmx 是 6000，Xmn 是 500，那么 Xmx-Xmn 是 5500 兆，也就是老年代有 5500 兆，CMSInitiatingOccupancyFraction=90 说明老年代到 90%满的时候开始执行对老年代的并发垃圾回收（CMS），这时还剩 10%的空间是 5500*10%=550 兆，所以即使 Xmn（也就是年轻代共 500 兆）里所有对象都搬到老年代里，550 兆的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的 promotion failed；

如果按照 Xmx=2048,Xmn=768 的比例计算，则 CMSInitiatingOccupancyFraction 的值不能超过 40，否则就容易出现垃圾回收时的 promotion failed。

5.4.10 -XX:+UseCMSInitiatingOccupancyOnly

指示只有在老年代在使用了初始化的比例后 concurrent collector 启动收集

5.4.11 -XX:SoftRefLRUPolicyMSPerMB

相对于客户端模式的虚拟机（-client 选项），当使用服务器模式的虚拟机时（-server 选项），对于软引用（soft reference）的清理力度要稍微差一些。可以通过增大-XX:SoftRefLRUPolicyMSPerMB 来降低收集频率。默认值是 1000，也就是说每秒一兆字节。Soft reference 在虚拟机中比在客户集中存活的更长一些。其清除频率可以用命令行参数-XX:SoftRefLRUPolicyMSPerMB=<N> 来控制，这可以指定每兆堆空闲空间的 soft reference 保持存活（一旦

它不强可达了）的毫秒数，这意味着每兆堆中的空闲空间中的 soft reference 会（在最后一个强引用被回收之后）存活 1 秒钟。注意，这是一个近似的值，因为 soft reference 只会在垃圾回收时才会被清除，而垃圾回收并不总在发生。

5.4.12 -XX:LargePageSizeInBytes

内存页的大小， 不可设置过大，会影响 Perm 的大小。

5.4.13 -XX:+UseFastAccessorMethods

原始类型的快速优化，get,set 方法转成本地代码。

5.4.14 -XX:+DisableExplicitGC

禁止 java 程序中的 full gc， 如 System.gc() 的调用。 最好加上防止程序在代码里误用了，对性能造成冲击。

5.4.15 -XX:+AggressiveHeap

特别说明下：(我感觉对于做 java cache 应用有帮助)

试图是使用大量的物理内存长时间大内存使用的优化，能检查计算

资源（内存， 处理器数量）

至少需要 256MB 内存

大量的 CPU / 内存， （在 1.4.1 在 4CPU 的机器上已经显示有提升）

5.4.16 -XX:+AggressiveOpts

加快编译

5.4.17 -XX:+UseBiasedLocking

锁机制的性能改善。

6 JVM 调优工具介绍

6.1 jmap 命令

查看堆内存分配和使用情况

./jmap -heap 31　　//31 为程序的进程号

6.2 Top 命令监控结果

通过使用 top 命令进行持续监控发现此时 CPU 空闲比例为 85.7%，剩余物理内存为 3619M，虚拟内存 8G 未使用。持续的监控结果显示进程 29003 占用系统内存不断在增加，已经快得到最大值。

6.3 Jstat 命令监控结果

./jstat –cutil 29003 1000 100

使用 jstat 命令对 PID 为 29003 的进程进行 gc 回收情况检查，发现由于 Old 段的内存使用量已经超过了设定的 80%的警戒线，导致系统每隔一两秒就进行一次 FGC，FGC 的次数明显多余 YGC 的次数，但是每次 FGC 后 old 的内存占用比例却没有明显变化—系统尝试进行 FGC 也不能有效地回收这部分对象所占内存。同时也说明年轻代的参数配置可能有问题，导致大部分对象都不得不放到老年代来进行 FGC 操作