JVM

1.文件上标记非重点并不是不重要，只是面试很少被问到，如果时间紧急可以只看前面的简介知道他是干什么的就行了。
2.重点模块中也有非重点的，需要了解的，我会在课程过程中点出来。
3.此课程帮助快速掌握面试中的技能，能面试通过并不代表什么都会了，在以后的学习中还需要进行努力。
4.我将讲成一个系统，不是和市场上的面试题一样，只有单个知识点去背诵。也不同于讲课，讲课太慢了，虽然能够让你知道的更多更细。

基本概念：

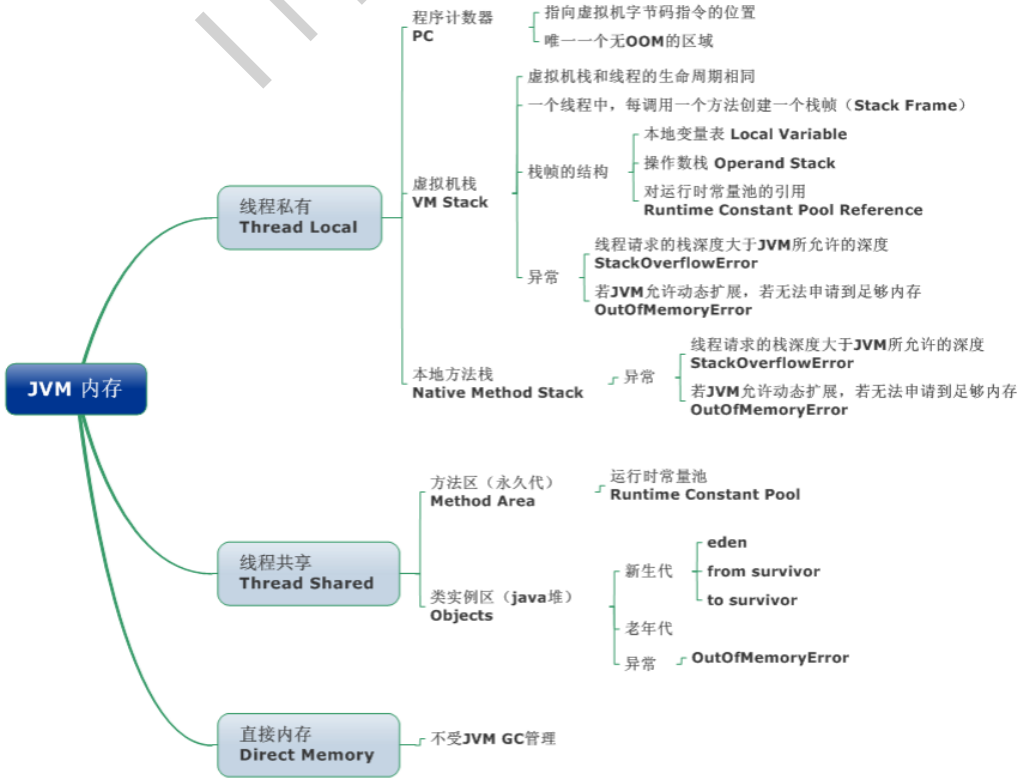
- JVM 是可运行 Java 代码的假想计算机 ，包括一套字节码指令集、一组寄存器、一个栈、 一个垃圾回收，堆 和 一个存储方法域。JVM 是运行在操作系统之上的，它与硬件没有直接的交互。

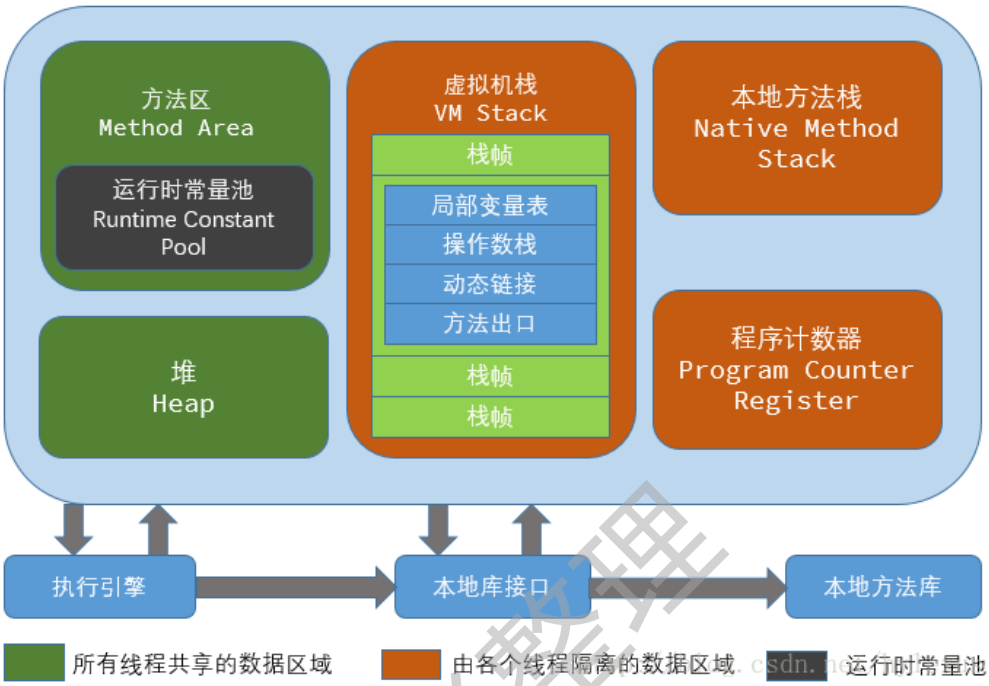
运行过程：

- 我们都知道 Java 源文件，通过编译器，能够生产相应的.Class 文件，也就是字节码文件，而字节码文件又通过 Java 虚拟机中的解释器，编译成特定机器上的机器码 。
- 也就是如下：
  - ① Java 源文件—->编译器—->字节码文件
  - ② 字节码文件—->JVM—->机器码
- 每一种平台的解释器是不同的，但是实现的虚拟机是相同的，这也就是 Java 为什么能够跨平台的原因了 ，当一个程序从开始运行，这时虚拟机就开始实例化了，多个程序启动就会存在多个虚拟机实例。程序退出或者关闭，则虚拟机实例消亡，多个虚拟机实例之间数据不能共享。

JVM内存区域

-



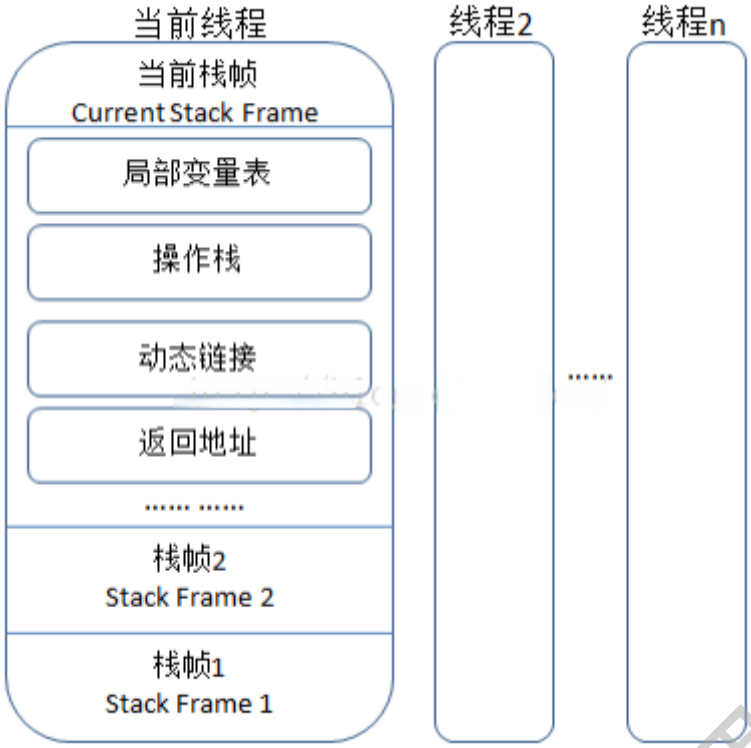
- JVM 内存区域主要分为线程私有区域【程序计数器、虚拟机栈、本地方法区】、线程共享区域【JAVA 堆、方法区】、直接内存。
  - 线程私有数据区域生命周期与线程相同, 依赖用户线程的启动/结束 而 创建/销毁(在 Hotspot VM 内, 每个线程都与操作系统的本地线程直接映射, 因此这部分内存区域的存/否跟随本地线程的生/死对应)。
  - 线程共享区域随虚拟机的启动/关闭而创建/销毁。
  - 

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。

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。



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。

堆（Heap-线程共享）-运行时数据区

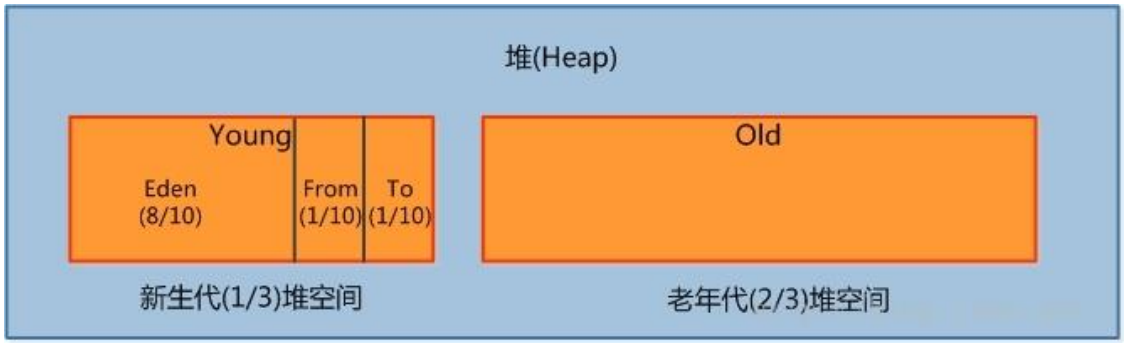
- 是被线程共享的一块内存区域，创建的对象和数组都保存在 Java 堆内存中，也是垃圾收集器进行垃圾收集的最重要的内存区域。由于现代 VM 采用**分代收集算法**, 因此 Java 堆从 GC 的角度还可以细分为: **新生代**(*Eden 区*、*From Survivor 区*和 *To Survivor 区*)和**老年代。**

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。

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。**

- 

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。
- **MinorGC 的过程（复制->清空->互换）**
  - MinorGC 采用复制算法。
  - 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）。
  - 然后，清空 Eden 和 ServicorFrom 中的对象。
  - ServicorTo和 ServicorFrom互换。
  - 最后，ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

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。

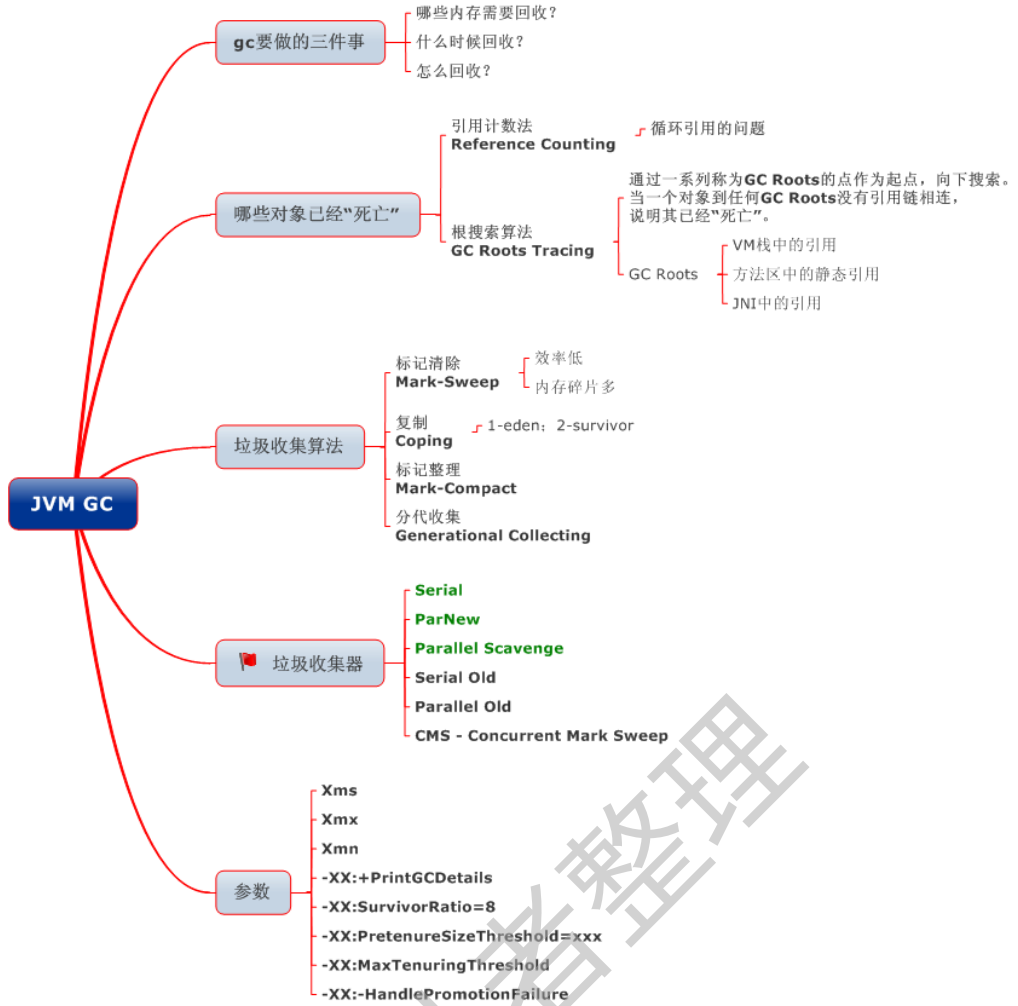
永久代

- 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。

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。

垃圾回收与算法

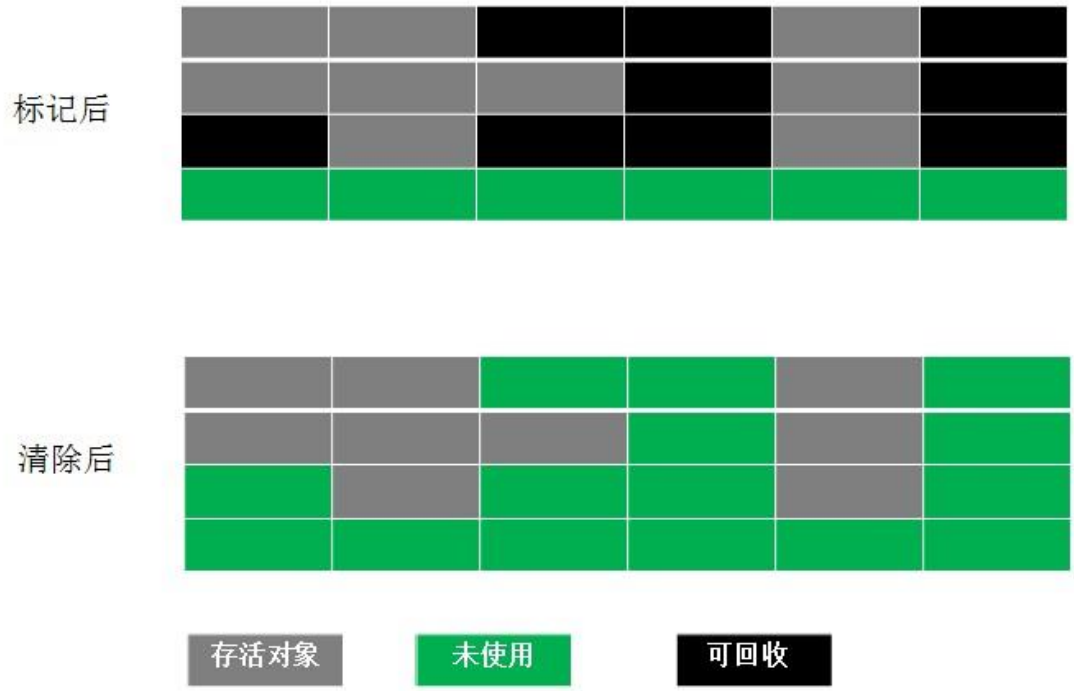
- 

如何确定垃圾

- **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。
- **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。

标记清除算法（Mark-Sweep）

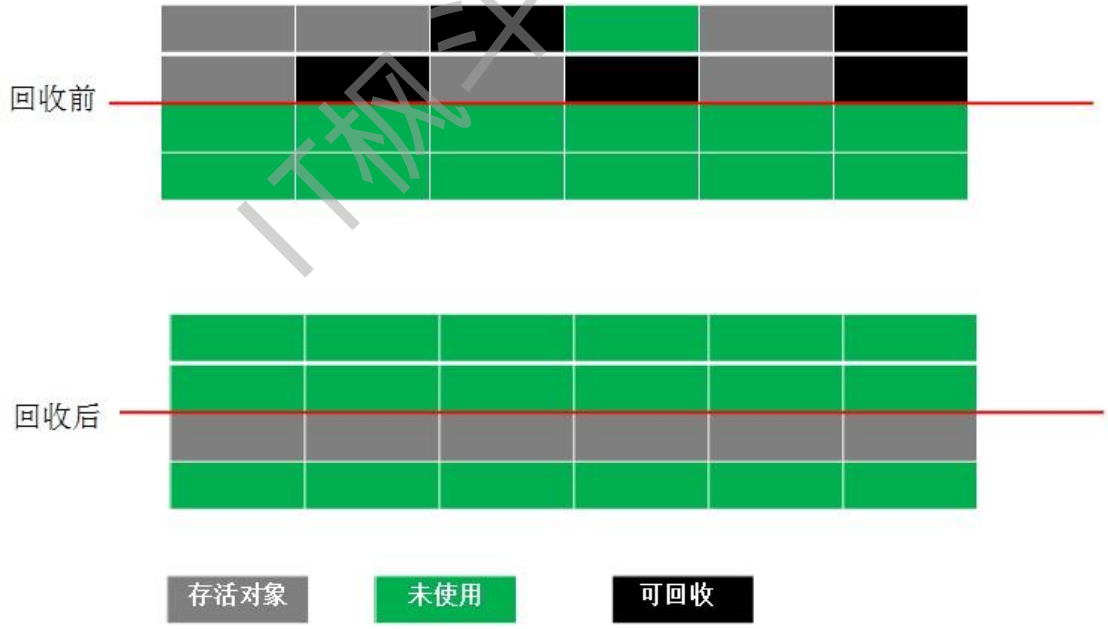
- 最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图



- 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。

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。

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
- **新生代与复制算法**

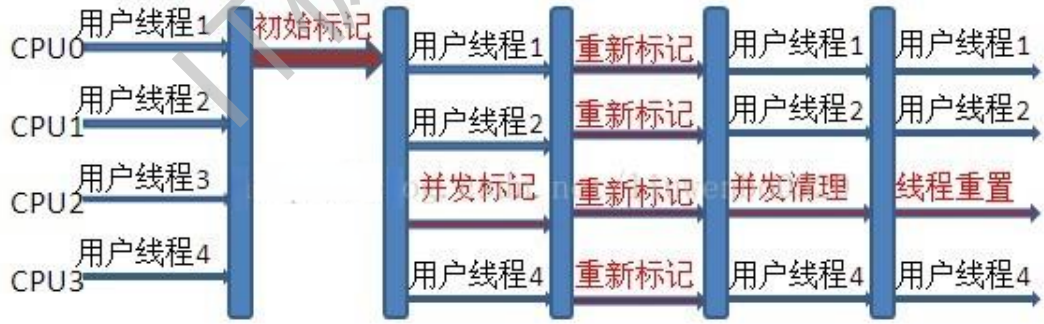
    - 每次垃圾收集都能发现大批对象已死, 只有少量存活. 因此选用复制算法, 只需要付出少量存活对象的复制成本就可以完成收集.
- **在老年代-标记整理算法**
    - 因为对象存活率高、没有额外空间对它进行分配担保, 就必须采用“标记—清理”或“标记—整理”算法来进行回收, 不必进行内存复制, 且直接腾出空闲内存.

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。

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：
    - **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。
    - **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。
    - **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。
    - **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：
    -



G1 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：
    - 基于标记-整理算法，不产生内存碎片。
    - 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。

JAVA 四中引用类型

- **强引用**
- 在 Java 中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。
- **软引用**
- 软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。
- **弱引用**
- 弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。
- **虚引用**
- 虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。虚引用的主要作用是跟踪对象被垃圾回收的状态。

JAVA IO/NIO

阻塞 IO 模型

- 最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出 CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除 block 状态。典型的阻塞 IO 模型的例子为：data = socket.read();如果数据没有就绪，就会一直阻塞在 read 方法。

非阻塞 IO 模型

- 当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞 IO 不会交出 CPU，而会一直占用 CPU。典型的非阻塞 IO 模型一般如下：

```
while(true){
data = socket.read();
if(data!= error){
处理数据
break;
}}
```

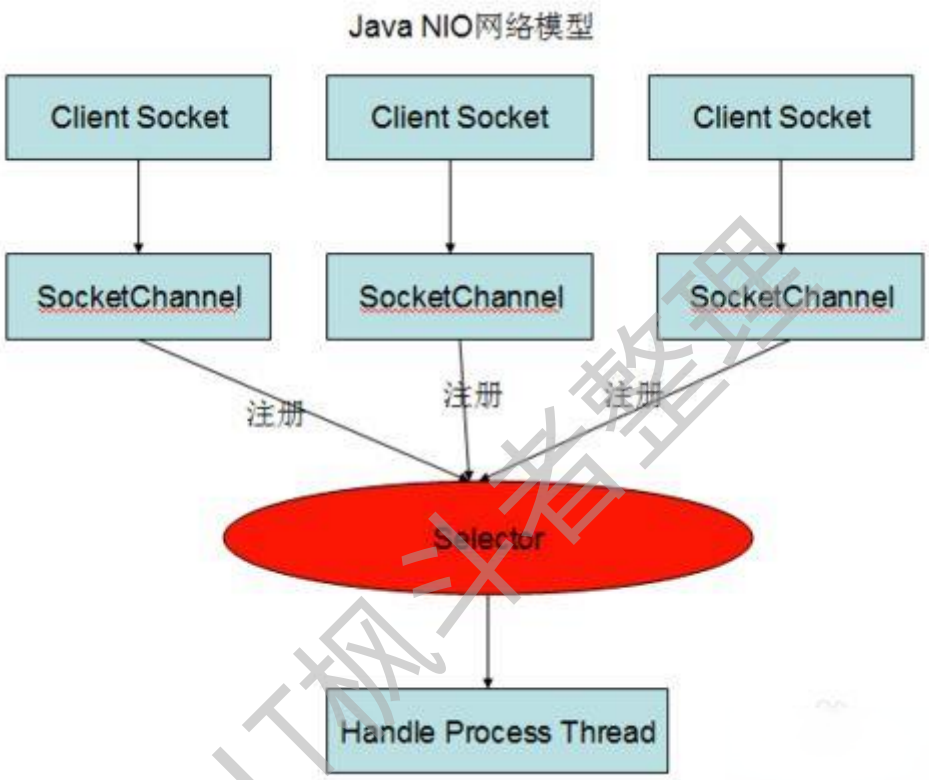
- 但是对于非阻塞 IO 就有一个非常严重的问题，在 while 循环中需要不断地去询问内核数据是否就绪，这样会导致 CPU 占用率非常高，因此一般情况下很少使用 while 循环这种方式来读取数据。

多路复用 IO 模型

- 多路复用 IO 模型是目前使用得比较多的模型。Java NIO 实际上就是多路复用 IO。在多路复用 IO 模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个 socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有 socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。在 Java NIO 中，是通过 selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

- 另外多路复用 IO 为何比非阻塞 IO 模型的效率高是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。
- 不过要注意的是，多路复用 IO 模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用 IO 模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

JAVA NIO

- NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区), Selector。传统 IO 基于字节流和字符流进行操作，而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。
- 

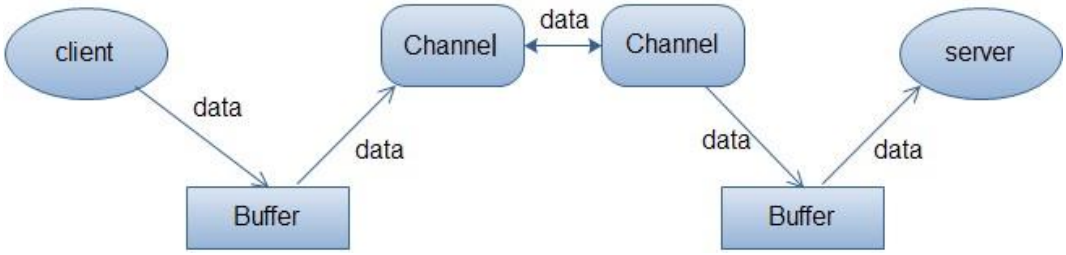
- NIO 和传统 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。

NIO 的缓冲区

- Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。NIO 的缓冲导向方法不同。数据读取到一个它稍后处理的缓冲区，需要时可在 缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所 有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

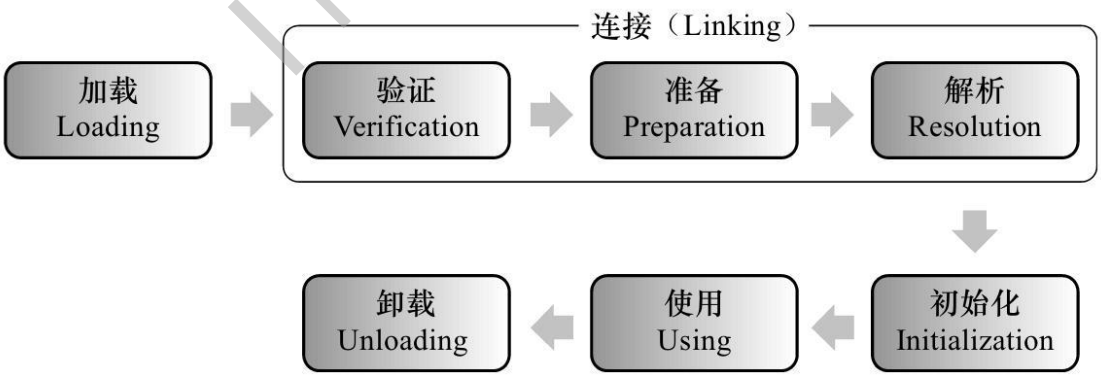
NIO 的非阻塞

- IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 NIO 的非阻塞模式， 使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上 执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

- **Channel**
  - 首先说一下 Channel，国内大多翻译成“通道”。Channel 和 IO 中的 Stream(流)是差不多一个等级的。只不过 Stream 是单向的，譬如：InputStream, OutputStream，而 Channel 是双向 的，既可以用来进行读操作，又可以用来进行写操作。
- **Buffer**
  - Buffer，故名思意，缓冲区，实际上是一个容器，是一个连续数组。Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer。
  - 

  - 上面的图描述了从一个客户端向服务端发送数据，然后服务端接收数据的过程。客户端发送数据时，必须先将数据存入 Buffer 中，然后将 Buffer 中的内容写入通道。服务端这边接收数据必 须通过 Channel 将数据读入到 Buffer 中，然后再从 Buffer 中取出数据来处理。
  - 在 NIO 中，Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 的子类有：ByteBuffer、IntBuffer、 CharBuffer、 LongBuffer、 DoubleBuffer、FloatBuffer、ShortBuffe
- **Selector**
  - Selector 类是 NIO 的核心类，Selector 能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护 多个线程，并且避免了多线程之间的上下文切换导致的开销。

JVM 类加载机制

- JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。
- 

- **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）。
- **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。
- **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。
- **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
        - CONSTANT_Class_info
        - CONSTANT_Field_info
        - CONSTANT_Method_info
        - 等类型的常量。
- **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 程序码。

引用类型

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。

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。

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：

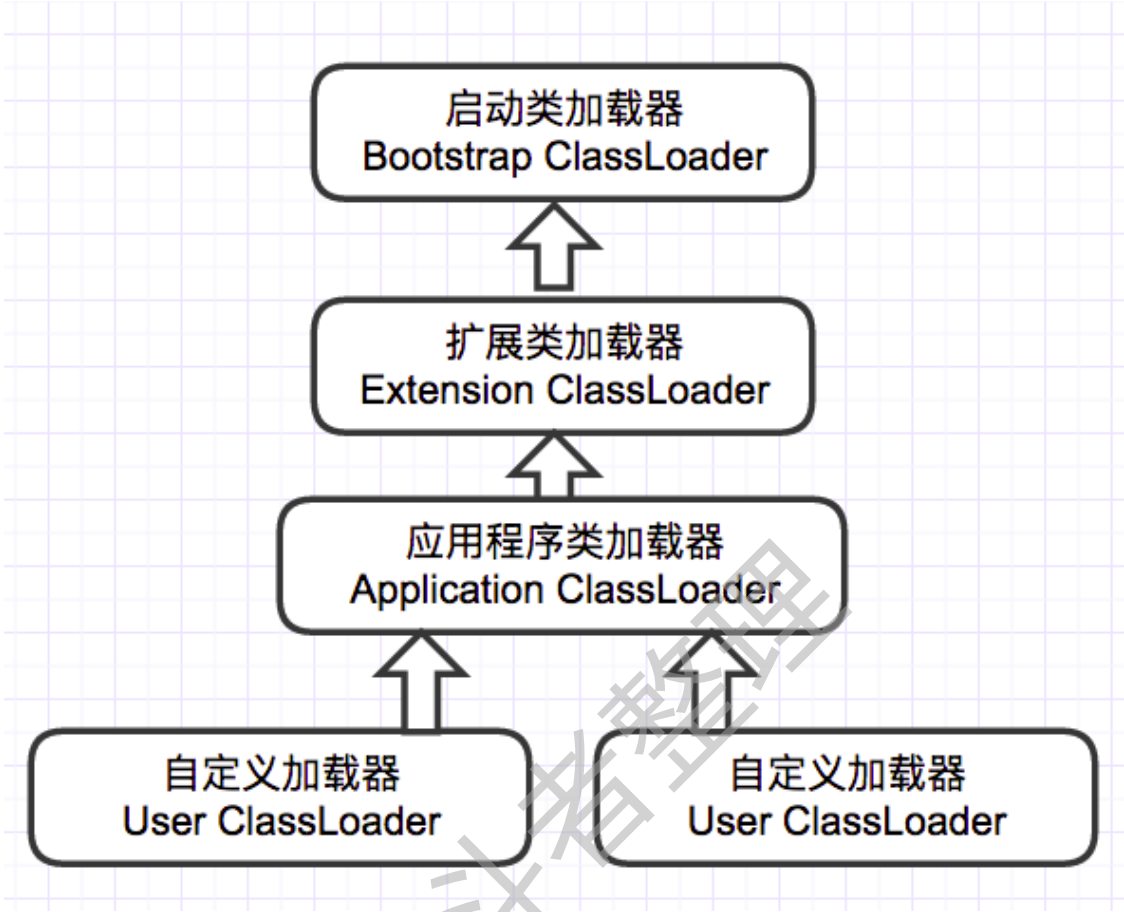
启动类加载器(Bootstrap ClassLoader)

- 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。

扩展类加载器(Extension ClassLoader)

- 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。

应用程序类加载器(Application ClassLoader):

- 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。
- 

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。
-