<https://www.cnblogs.com/chanshuyi/p/jvm_serial_07_jvm_class_loader_mechanism.html>

**类的加载机制：**

加载、验证、准备、解析、初始化、使用、卸载。

加载：加载代码到内存中（加载class字节码文件）。

验证：验证文件是否符合jvm规范，验证代码是否有语法等错误。

准备：为static修饰的对象/方法分配内存，但不初始化。static final修饰的分配内存并初始化。

解析：JVM 针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类引用进行解析，将其在常量池中的符号引用替换成直接其在内存中的直接引用。

初始化：初始化类：按顺序初始化static修饰的。

初始化(new)对象：若类没有进行初始化，则先初始化类，再初始化对象；不初始化static修饰的。

初始化类时先初始化父类。

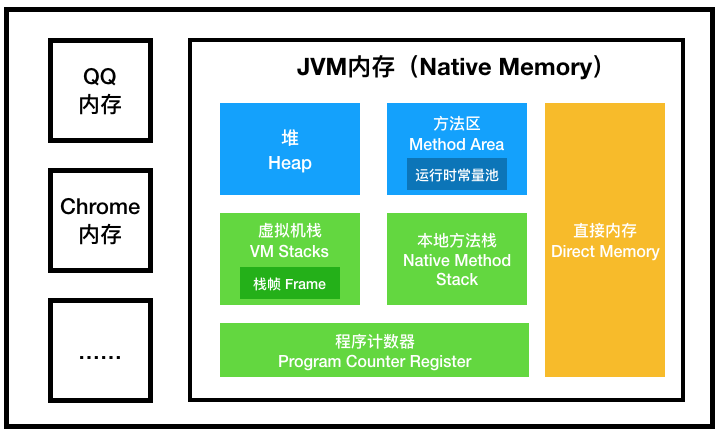
获取父类的静态变量时，只初始化父类。

使用：执行代码。

卸载：销毁class文件。

Spring项目启动时，jvm只加载并初始化在容器内的类及对象，不在容器内的类在用到时才会加载。

**JVM内存模型：**

****

**线程共享部分：java堆，方法区，常量池。**

方法区&常量池：

存储 Java 类字节码数据的一块区域，它存储了每一个类的结构信息，如运行时常量池、类的成员变量、类的方法等。

Java堆：

用于java实例对象的内存分配。Jvm GC指回收java堆的内存。

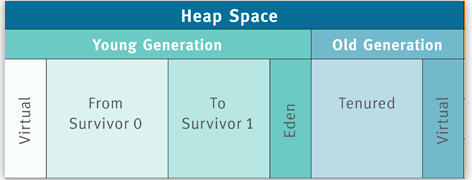
Java堆被分为年轻代和老年代两个区域。需要为对象分配内存时，优先分配在Eden区，Eden区内存不够时会进行GC，年轻代的对象经历过指定次数的GC则会被移动到老年代。年轻代空间大小默认分配Eden:from:to=8:1:1，因为大部分对象的存活时间都比较短，这样分配可以减少内存空间的浪费。

Java堆其实还未每一个线程单独分配了一块TLAB空间，这部分空间在分配时是线程独享的，在使用时是线程共享的。

为了保证内存分配的线程安全，防止出现两个不同对象指向同一个内存区域的情况。

每个线程在Java堆中预先分配一小块内存，然后再给对象分配内存的时候，直接在自己这块”私有”内存中分配，当这部分区域用完之后，再分配新的”私有”内存。（HotSpot虚拟机实现）

另外，TLAB仅作用于年轻代的Eden Space，对象被创建的时候首先放到这个区域，但是年轻代分配不了内存的大对象会直接进入老年代。因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效。



Tips:

虚拟机规范对方法区实现的位置并没有明确要求，在最著名的HotSopt虚拟机实现中（在Java 8 之前），方法区仅是逻辑上的独立区域，在物理上并没有独立于堆而存在，而是位于永久代中（java堆被分为年轻代、老年代、永久代）。所以，这时候方法区也是可以被垃圾回收的。

在Java 8中 ，HotSpot虚拟机移除了永久代，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）。

**线程私有部分：pc寄存器，java虚拟机栈，native方法栈。**

PC寄存器：保存线程当前正在执行的方法的地址。

Java虚拟机栈：存储当前线程的局部变量表(各种基本数据类型,对象的引用)及操作数栈。

Native方法栈：存储当前线程调用native方法的局部变量表及操作数栈。

**直接内存：**

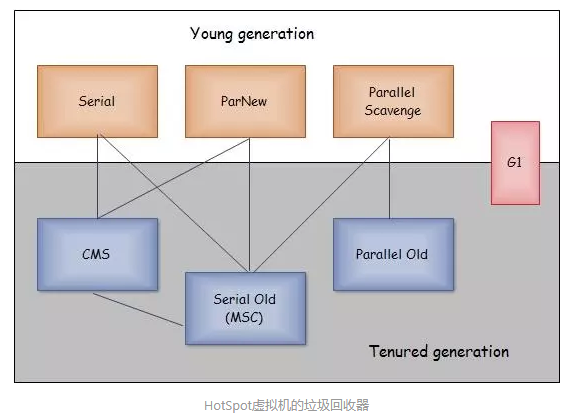
又称堆外内存，这部分内存不由jvm管理和回收，需要手动回收。

直接内存默认空间为-Xmx(最大堆空间)，可以通过-XX:MaxDirectMemorySize设置。

一般使用Unsafe类和NIO中的DirectByteBuffer(实际上还是使用Unsafe)来申请管理回收直接内存。

Jdk1.4中引入了NIO，DirectByteBuffer使用Unsafe类直接申请/回收直接内存。DirectByteBuffer对象指向申请到的直接内存，当处于java堆中的DirectByteBuffer对象被GC回收时，指向的直接内存也将被释放(使用Cleaner实现)。

**JVM垃圾回收器：**

****

年轻代GC（Minor GC）：

发生在年轻代的垃圾回收动作，Minor GC比较频繁，一般速度也比较快。

老年代GC（Major GC / Full GC）：

发生在老年代的垃圾回收动作，经常会伴随着至少一次的Minor GC。速度较慢。

吞吐量：

CPU用于运行用户代码的时间与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）。运行用户代码时间越长，吞吐量越大。

判断对象是否能被回收：

根搜索算法是JVM用来的判断对象是否存活的算法，此算法基本思路为通过一系列的“GC Roots”对象作为起始点，从这些节点往下搜索，当一个对象和GC Roots不可达时，则该对象是无用的，可被回收的。

**年轻代垃圾收集器：**

年轻代对象存活率低，所以年轻代收集器都使用复制算法来进行垃圾收集。

将存活的对象复制到未被使用的内存空间中，随后清除所有正在使用的内存中的所有对象。

缺点：浪费内存空间。

Serial收集器：

是一个单线程的收集器，在垃圾收集的时候必须暂停所有工作线程，直到收集完成(stop the world)。

是虚拟机运行在client模式下默认的年轻代收集器。

优点：简单。

ParNew收集器：

是Serial收集器的多线程版本。

是许多运行在Server模式下的虚拟机中首选的新生代收集器。因为老年代的CMS收集器只能与ParNew或Serial收集器配合使用。

优点：在多CPU的服务器上，性能较Serial收集器有明显提升。

Parallel Scavenge收集器：

是一个注重高吞吐量的多线程收集器。

适合应用于后台计算较多而用户交互少的任务。

Parallel scavenge收集器有自适应的调节策略，通过-XX:+UseAdaptiveSizePolicy配置。Jvm会监控当前的运行情况，动态调整新生代大小、eden与survival区的比例、晋升老年代所需的GC次数等参数，以提供最适合的停顿时间或最大的吞吐量。

**老年代垃圾收集器：**

CMS收集器使用标记清除算法。

标记所有存活对象，然后清除所有未被标记的对象。

缺点：会产生大量空间碎片，内存分配不连续，导致分配大对象时出现问题。

Serial old收集器及Parallel old收集器使用标记整理算法。

首先标记所有的存活对象，然后将所有标记的对象向内存空间的一端移动，最后释放边界外的所有内存。

是标记清除法的优化版，解决了空间碎片的问题。

Serial old收集器：

Serial收集器的老年代版本。使用标记整理算法。

Parallel old收集器：

Parallel scavenge收集器的老年代版本。使用标记整理算法。

CMS收集器：

是以达到最短回收停顿时间为目标的收集器。使用标记清除算法。

适合应用于与用户交互的，需要快速的响应速度的应用。

垃圾收集过程：

1.初始标记：仅标记GC root对象，需要stop the world。速度快。

2.并发标记：根据已标记的GC root对象继续检索标记。与用户线程同时运行。耗时长。

3.重新标记：并发标记期间程序运行会产生新对象，重新标记是为了标记这些新对象。

需要stop the world。速度比初始标记稍慢。

4.并发清除：清除垃圾。与用户线程同时运行。耗时长。

**JVM的client模式与server模式：**

client模式启动比较快，但运行时性能和内存管理效率不如server模式，通常用于客户端应用程序。相反，server模式启动比client慢，但可获得更高的运行性能。

在 windows上，缺省的虚拟机类型为client模式。在Linux上缺省采用server模式。