<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。

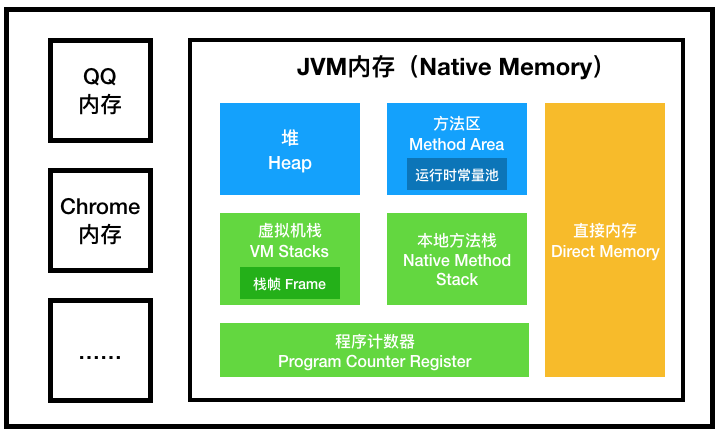
获取父类的静态变量时，只初始化父类。

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。

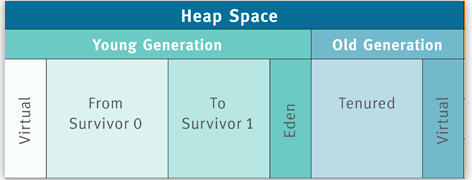
Java堆被分为年轻代和老年代两个区域。需要为对象分配内存时，优先分配在Eden区，Eden区内存不够时会进行GC，年轻代的对象经历过指定次数的GC则会被移动到老年代。年轻代空间大小默认分配Eden:from:to=8:1:1，因为大部分对象的存活时间都比较短，这样分配可以减少内存空间的浪费。

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：**

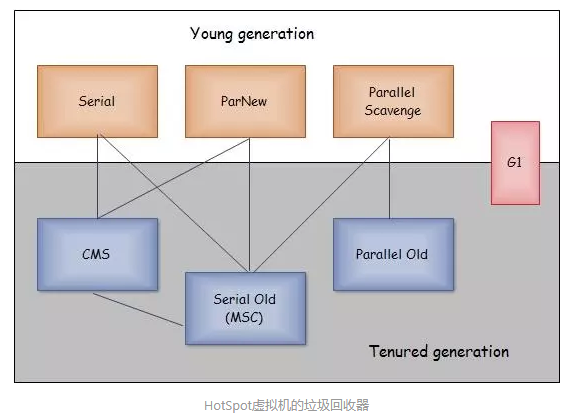
又称堆外内存，这部分内存不由jvm管理和回收，需要手动回收。

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。

可作为GC roots的对象：

1.虚拟机栈中引用的对象 2.方法区中的类静态属性引用的对象 3.方法区中的常量引用的对象 4.本地方法栈引用的对象

**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。（不适用大堆，10G以内）

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。速度比初始标记慢。

4.并发清除：清除垃圾。与用户线程同时运行。耗时长。

优点：并发收集，低停顿。

缺点：

1.会产生大量空间碎片：

由于使用标记清除算法，会产生大量空间碎片，容易出现老年代有很大空间剩余，却

找不到足够的连续空间来分配大对象的情况，不得不提前触发一次full GC。

为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollection开关

参数，用于在Full  GC之后增加一个碎片整理过程，还可通过

-XX:CMSFullGCBeforeCompaction参数设置执行多少次不压缩的Full  GC之后，跟着来

一次碎片整理过程。

2.对CPU资源敏感：

由于在并发标记与并发清除阶段和用户线程同时运行，会占用CPU资源。导致在CPU

少的时候，对用户程序的影响较大。

CMS默认启动的回收线程数是：(CPU数量+3) / 4。

3.无法处理浮动垃圾：

浮动垃圾：由于CMS并发清理阶段用户线程还在运行，用户线程的运行会产生新的 垃圾，这一部分垃圾出现在标记过程之后，无法在本次收集中处理，只能下一次GC 时清理。这一部分垃圾称为“浮动垃圾”。

CMS收集器需要预留部分内存空间给并发运行阶段的用户线程使用，如果预留的内 存空间无法满足用户线程的需要，则会触发“Concurrent Mode Failure”，临时启动serial old收集器重新进行垃圾收集，造成停顿时间长的现象。

CMS收集器默认在老年代使用了68%的空间时激活，可以通过参

数-XX:CMSInitiatingOccupancyFraction的值来提供触发百分比。

**G1(Garbage-First)收集器：**

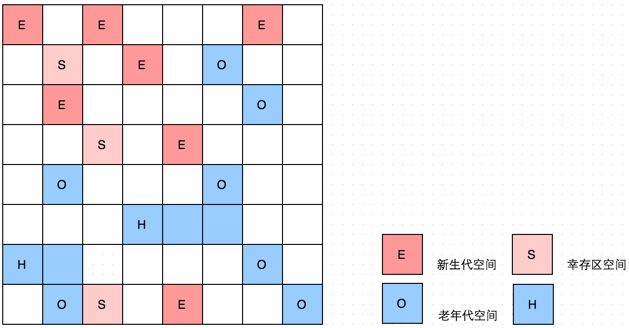
于java7推出，java9设置为默认收集器。

是一种服务器端的垃圾收集器，应用在多处理器和大容量内存环境中，在实现高吞吐量 的同时，尽可能的满足垃圾收集暂停时间的要求。它是专门针对以下应用场景设计的: \* 像CMS收集器一样，能与应用程序线程并发执行。 \* 整理空闲空间更快。 \* 需要GC 停顿时间更好预测。 \* 不希望牺牲大量的吞吐性能。 \* 不需要更大的Java Heap

**G1的内存布局：**

其它收集器都将java堆分为固定大小的连续3个空间(年轻代、老年代、永久代/元空间)。

G1将java堆划分为若干个大小相等的region，每个region有自己的分代标志(eden, survivor, old)，在逻辑上不相邻。



region大小为1M~32M(2^n)之间，目标是分配不超过2048个region。可以通过 -XX:G1HeapRegionSize=n来配置region大小，但不是最终值，最终大小由jvm决定。

图中“H”为大对象（humongous object，H-obj），大小大于等于region一半的对象。

H-obj直接分配到老年代。

在global concurrent marking阶段的cleanup 和 full GC阶段回收。

在分配H-obj之前先检查是否超过 initiating heap occupancy percent，超过则启动global concurrent marking，为了提早回收，防止 evacuation failures 和 full GC(采用serial方式)。

**G1的垃圾回收机制：**

G1 GC模式：

G1提供了两种GC模式：young GC和mixed GC。二者都是完全STOP THE WORLD的。

**Young GC：**

当eden区内存不足时，会触发young GC。选定所有的年轻代region构成CSet（collection set）。

通过控制年轻代region个数来控制GC时间开销。所以当-XX:MaxGCPauseMillis（默认 200ms）设置的时间过小，年轻代的region个数会很少，导致young GC的频率大大增 加。

**RSet（Remembered Set）：**

在逻辑上，每个region都有一个RSet，记录了其它region中的对象对本region的引用 （记录了”谁引用我”）。

在进行young GC时，可以通过RSet知道该对象是否被老年代的对象（old->young）引 用，而不用扫描整个老年代。大大减少了GC的工作量。

进行mixed GC时，有Rset记录的（young->old）（old->old）的引用，因此不用扫描整 个老年代。减少了mixed GC的工作量。

**Mixed GC：**

分为两个阶段：

全局并发标记（global concurrent marking）

拷贝存活对象（evacuation）

-XX: G1HeapWastePercent：

当老年代垃圾占比达到此参数时，会触发Mixed GC。默认值10%。

选定所有年轻代region和部分收集收益高的老年代region。

通过global concurrent marking统计得出收集收益高的老年代Region，构成收集集合 CSet以此来控制时间开销。

**Pause Prediction Model（停顿预测模型）：**

G1根据这个模型统计计算出来的历史数据来预测本次收集需要选择的Region数量，从 而尽量满足用户设定的目标停顿时间。

**global concurrent marking：**

与CMS类似，但是是为Mixed GC服务的，不是一次GC的必须过程。

1.初始标记（Initial Mark，STW）：标记从GC root开始可达的对象。借用young GC的暂 停，因此没有额外的暂停阶段。

2.并发标记（Concurrent Marking）：从GC Root开始对heap中的对象标记，标记线程与 应用程序线程并行执行，并且收集各个Region的存活对象信息。此阶段可以被young GC中断，young GC完成后继续执行。

3.最终标记（Remark，STW）：标记那些在并发标记阶段发生变化的对象。

4.清除（Cleanup）：收集没有存活对象的Region。

- XX:InitiatingHeapOccupancyPercent：

当此Java堆（整个堆）占用率达到该阈值时，触发global concurrent marking。默认值 为45%。

SATB：

在global concurrent marking的并发标记阶段。

假设正在对A进行标记，已经将A中的 成员变量B标记完成，此时B对B1的引用修 改为B2，那么B2就有可能被漏标，而被 回收。

为了确保不出现漏标对象的情况，SATB通过write barrier 将引用都记录下来，认为引 用记录中的对象都是活对象。

缺点：这种做法会让部分可回收的对象躲过此次GC，导致浮动垃圾（float garbage）。

**G1的相关参数：**

\*开头的设置在Java HotSpot VM, build 23不可用。

\*-XX: G1HeapWastePercent：

当老年代垃圾占比达到此参数时，会触发Mixed GC。默认值10%。

- XX:InitiatingHeapOccupancyPercent：

当此Java堆（整个堆）占用率达到该阈值时，触发global concurrent marking。默认值 为45%。

\*-XX: G1MixedGCLiveThresholdPercent：

old generation region中的存活对象的占比，只有在此参数之下，才会被选入CSet。此设 置替换了-XX:G1OldCSetRegionLiveThresholdPercent。默认值65%。

\*-XX: G1MixedGCCountTarget：

一次global concurrent marking之后，最多执行Mixed GC的次数。默认8次。

\*-XX: G1OldCSetRegionThresholdPercent：

一次Mixed GC中能被选入CSet的最多老年代region个数。默认为10%（java堆的10%）。

-XX:G1HeapRegionSize：

设置Region大小，并非最终值。1~32M（2^n）。

\*- XX:MaxGCPauseMillis：

GC收集目标耗时。默认200ms。不宜太小（如50ms以下）。

\*-XX:G1NewSizePercent：

新生代最小值，默认值5%。

\*- XX:G1MaxNewSizePercent：

新生代最大值，默认值60%。

- -XX:ParallelGCThreads：

STW期间，并行GC线程数。最多为8。

- -XX:ConcGCThreads：

并发标记阶段，并行执行的线程数。设置为ParallelGCThreads的1/4。

\*-XX:G1ReservePercent：

设置保留空闲内存的百分比，以减少空间溢出(Evacuation Failure)的风险。默认是10%。

**G1触发FULL GC：**

1.分配大对象时，老年代空间不够。

2.mixed GC的速度无法跟上对象新增的速度，导致老年代空间不够。

3.代码调用Sytem.gc()。

FULL GC是采用serial old GC来对整堆进行一次GC。会导致长时间的STW，应尽量避免 这种情况的出现。

**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。