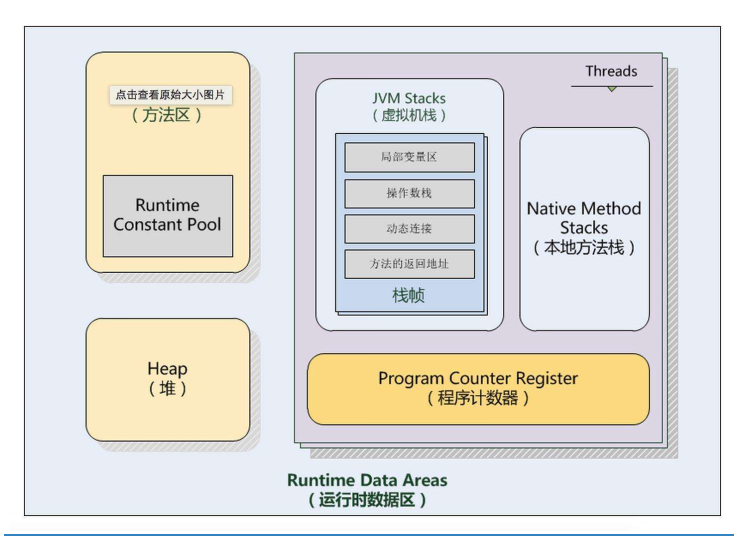
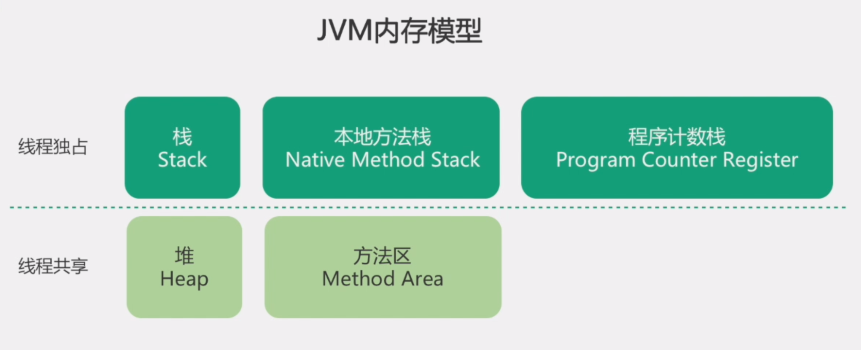
JVM

1. JVM内存模型
   1. JVM内存模型是什么？

jvm内存模型：jvm的内存分区





* 1. 堆内存分区



eden:survivor1:survivor2 =8:1:1 默认分配方式，

Eden区 —— 新对象或者生命周期很短的对象会存储在这个区域中，这个区的大小可以通过-XX:NewSize和-XX:MaxNewSize参数来调整。新生代GC（垃圾回收器）会清理这一区域。Survivor区 —— 那些历经了Eden区的垃圾回收仍能存活下来的依旧存在引用的对象会待在这个区域。这个区的大小可以由JVM参数-XX:SurvivorRatio来进行调节。

方法区也被称为非堆区域（在HotSpot JVM的实现当中）它被分为两个主要的子区域持久代 —— 这个区域会存储包括类定义，结构，字段，方法（数据及代码）以及常量在内的类相关数据。它可以通过-XX:PermSize及 -XX:MaxPermSize来进行调节。如果它的空间用完了，会导致java.lang.OutOfMemoryError: PermGen space的异常。代码缓存——这个缓存区域是用来存储编译后的代码。编译后的代码就是本地代码（硬件相关的），它是由JIT（Just In Time)编译器生成的，这个编译器是Oracle HotSpot JVM所特有的。

元空间 —— 快速入门它是本地堆内存中的一部分它可以通过-XX:MetaspaceSize和-XX:MaxMetaspaceSize来进行调整当到达XX:MetaspaceSize所指定的阈值后会开始进行清理该区域如果本地空间的内存用尽了会收到java.lang.OutOfMemoryError: Metadata space的错误信息。和持久代相关的JVM参数-XX:PermSize及-XX:MaxPermSize将会被忽略掉，并且在启动的时候给出警告信息。充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致元空间 —— 内存分配模型绝大多数的类元数据的空间都从本地内存中分配用来描述类元数据的类也被删除了，分元数据分配了多个虚拟内存空间给每个类加载器分配一个内存块的列表，只进行线性分配。块的大小取决于类加载器的类型， sun/反射/代理对应的类加载器的块会小一些。不会单独回收某个类，如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉。这样减少了碎片，并节省GC扫描和压缩的时间。元空间 —— 调优使用-XX:MaxMetaspaceSize参数可以设置元空间的最大值，默认是没有上限的，也就是说你的系统内存上限是多少它就是多少。使用-XX:MetaspaceSize选项指定的是元空间的初始大小，如果没有指定的话，元空间会根据应用程序运行时的需要动态地调整大小。 一旦类元数据的使用量达到了“MaxMetaspaceSize”指定的值，对于无用的类和类加载器，垃圾收集此时会触发。为了控制这种垃圾收集的频率和延迟，合适的监控和调整Metaspace非常有必要。过于频繁的Metaspace垃圾收集是类和类加载器发生内存泄露的征兆，同时也说明你的应用程序内存大小不合适，需要调整。

* 1. JVM内存模型分区详情

vm内存模型主要指运行时的数据区，包括5个部分。【实际面试题】

栈也叫方法栈，是线程私有的，线程在执行每个方法时都会同时创建一个栈帧，用来存储局部变量表、操作栈、动态链接、方法出口等信息。调用方法时执行入栈，方法返回时执行出栈。

本地方法栈与栈类似，也是用来保存线程执行方法时的信息，不同的是，执行java方法使用栈，而执行native方法使用本地方法栈。

程序计数器保存着当前线程所执行的字节码位置，每个线程工作时都有一个独立的计数器。程序计数器为执行java方法服务，执行native方法时，程序计数器为空。

栈、本地方法栈、程序计数器这三个部分都是线程独占的。

堆是jvm管理的内存中最大的一块，堆被所有线程共享，目的是为了存放对象实例，几乎所有的对象实例都在这里分配。当堆内存没有可用的空间时，会抛出OOM异常。根据对象存活的周期不同，jvm把堆内存进行分代管理，由垃圾回收器来进行对象的回收管理。

方法区也是各个线程共享的内存区域，又叫非堆区。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，

jdk1.7中的永久代和1.8中的metaspace都是方法区的一种实现。

面试回答此知识点相关问题时，要答出两个要点：一个是各部分的功能，另一个是哪些线程共享，哪些独占。

* 1. 常量池在jdk版本中的差别

1.6及以前常量池时在永久代中

1.7+常量池在堆空间

* 1. 常见面试题
     1. String.intern在jdk版本的区别

String s = new String("a");

//new String的步骤

1. 如果常量池中没有"a"对象，则在常量池中创建"a"对象，然后再在堆内存中创建"a"对象

2. 如果常量池中有"a"对象，则在堆内存中创建"a"对象

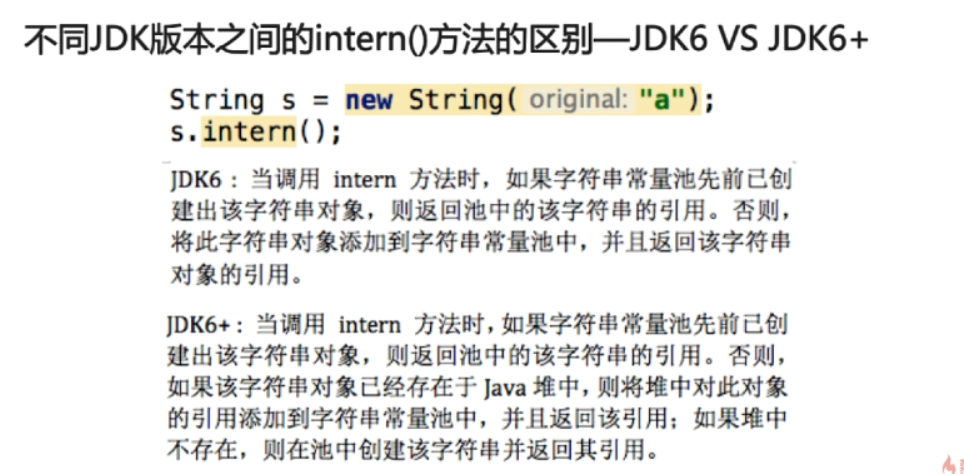
a.intern();

jdk1.6调用intern方法时

1. 如果常量池中已存在该对象，直接返回常量池中对象的引用
2. 如果常量池中不存在该对象，则将该字符串添加到常量池中，并且返回该字符串对象的引用

jdk1.6+调用intern方法时

1. 如果常量池中已存在该对象，直接返回常量池中对象的引用
2. 如果该字符传对象已经存在于Java堆中，则将堆中对此对象的引用添加到字符串常量池中，并且返回该引用
3. 如果堆中不存在，则在池中创建该字符串并返回引用



* + 1. 副主题
    2. 副主题
    3. 副主题

1. JMM内存模型
   1. JMM内存模型是什么?

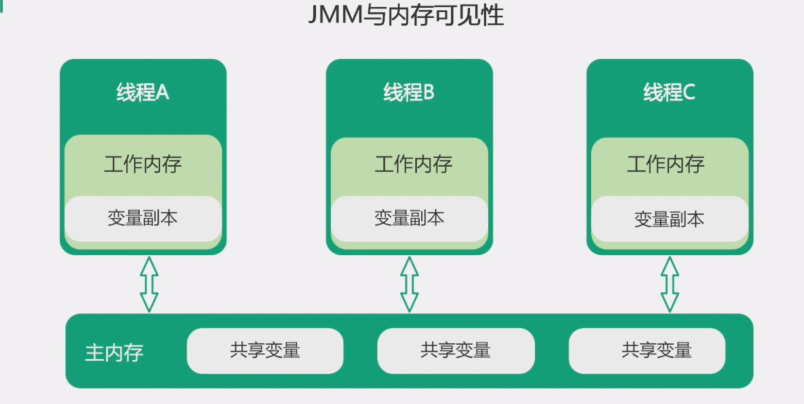
JMM是Java Memory Model的缩写是java内存模型，是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

* 1. JMM特性

1. 原子性

2. 可见性

3. 有序性

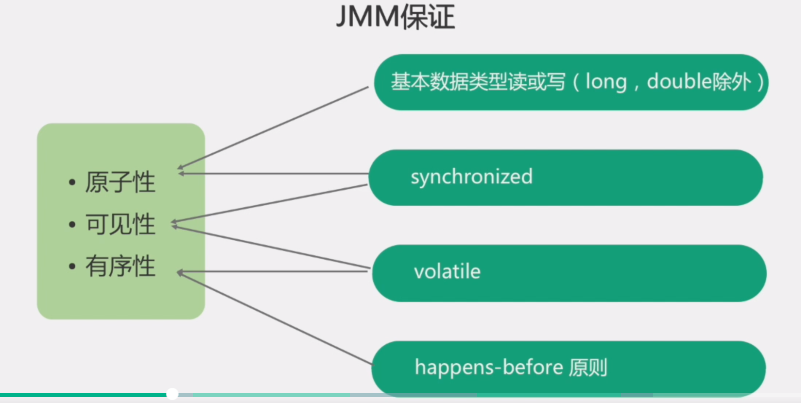


jmm是java内存模型，与刚才讲到的jvm内存模型是两回事，jmm的主要目标是定义程序中变量的访问规则，如图所示，所有的共享变量都存储在主内存中共享。每个线程有自己的工作内存，工作内存中保存的是主内存中变量的副本，线程对变量的读写等操作必须在自己 的工作内存中进行，而不能直接读写主内存中的变量。

在多线程进行数据交互时，例如线程a给一个共享变量赋值后，由线程b来读取这个值，a修改完变量是修改在自己的工作区内存中，b是不可见的，只有从a的工作区写回到主内存，b再从主内存读取到自己的工作区才能进行进一步的操作。由于指令重排序的存在，这个写-读的顺序有可能被打乱。

因此jmm需要提供原子性、可见性、有序性的保证。

主要介绍下jmm如何保证原子性、可见性，有序性。



jmm保证对除long和double外的基础数据类型的读写操作是原子性的。另外关键字Synchronized也可以提供原子性保证。Synchronized的原子性是通过java的两个高级的字节码指令monitorenter和monitorexit来保证的。

jmm可见性的保证，一个是通过Synchronized，另外一个就是volatile。volatile强制变量的赋值会同步刷新回主内存，强制变量的读取会从主内存重新加载，保证不同的线程总是能够看到该变量的最新值。

jmm对有序性的保证，主要通过volatile和一系列happens-before原则。volatile的另一个作用就是阻止指令重排序，这样就可以保证变量读写的有序性。

happens-before原则包括一系列规则，如

程序顺序原则，即一个线程内必须保证语义串行性；

锁规则，即对同一个锁的解锁一定发生在再次加锁之前；

此外还包括happens-before原则的传递性、线程启动、中断、终止规则等。

1. 执行模式
   1. 解释执行: -Xint
   2. 编译模式: -Xcomp
   3. 混合模式(默认): -Xmixed

JVM默认为混合模式

C:\Users\Administrator>java -version

java version "1.8.0\_191"

Java(TM) SE Runtime Environment (build 1.8.0\_191-b12)

Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)

1. 常见面试题
   1. JVM三大性能调优参数-Xms、-Xmx、-Xss的含义

-Xms: 初始化堆内存大小

-Xmx: 最大堆内存大小

-Xss: 规定了每个线程虚拟机栈(堆栈)的大小默认为256k，此配置会影响此进程中并发线程数的大小

我们

* 1. java内存模型中堆和栈的区别

内存分配策略

1. 静态存储：编译时确定每个数据目标在y运行时的存储空间需求
2. 栈式存储：数据区需求在编译时未知，运行时模块入口前确定
3. 堆式存储：编译时h或运行时模块入口都无法确定，动态分配

联系: 引用对象、数组时，栈里定义变量保存堆中目标的首地址

管理方式：

管理方式：栈自动释放、堆需要GC

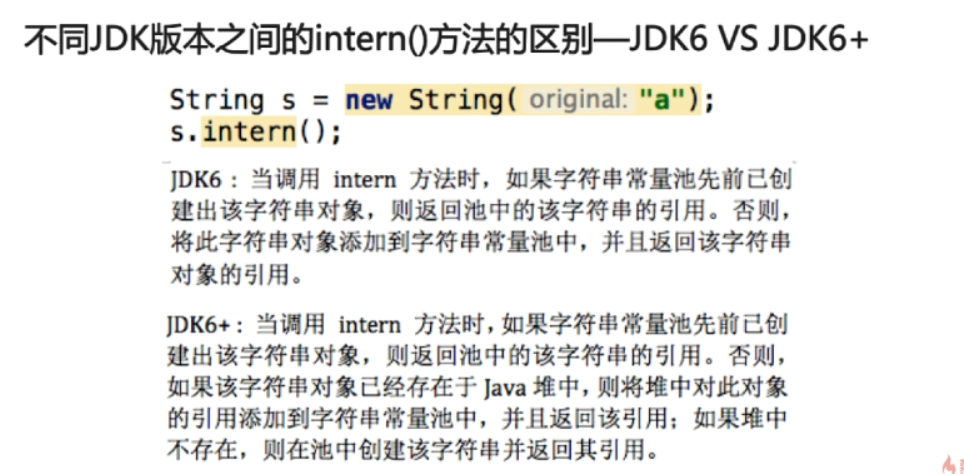
空间大小：栈空间很小、堆空间很大

碎片相关：栈产生的碎片远小于堆

分配方式：栈支持静态分配和动态分配、而堆仅支持动态分配

效率问题：栈的效率比堆高

* 1. intern()方法在jdk1.6与1.6+的区别



**public static void** main(**String**[] args) {  
 **String** s1 = **new** String("a");  
 s1.intern();  
 **String** s2 = "a";  
  
 **System**.***out***.println(s1==s2);*//false* **String** s3 = **new** String("a")+**new** String("a");  
 s3.intern();  
 **String** s4 = "aa";  
 **System**.***out***.println(s3==s4);*//true*}

在jdk1.6+版本,返回false、true

在jdk1.6版本，返回false、false

1. 常见的调优
   1. 堆内存分配

1. -Xms 与-Xmx一般设置成一样大小，防止扩容导致内存抖动，影响程序运行稳定性

* 1. Metaspace调优

https://blog.csdn.net/bolg\_hero/article/details/78189621

1. 性能分析工具
   1. 辅助命令JPS

jps -help

`

[root@~]# jps -help

usage: jps [-help]

jps [-q] [-mlvV] [<hostid>]

Definitions:

<hostid>: <hostname>[:<port>]

`

jps -V：默认

jps -v：输出传递给jvm参数

jps -l：显示全路径类名

jps -q：只显示PID

jps -m：显示简单类名和传递给main方法的cans

* 1. MAT
  2. JMC
  3. JSTACK
  4. JSTAT

1. jvm参数
   1. 标准参数（-）

所有的JVM实现都必须实现这些参数的功能，而且向后兼容

标准参数都是以 - 开头

常见的标准参数有:

java -version

java -help

java -showversion

通过java -?或者java -help查看jvm标准参数

* 1. 非标准参数（-X）

默认jvm实现这些参数的功能，但是并不保证所有jvm实现都满足，且不保证向后兼容

常见的非标准参数(-X)

-Xint:解释模式

-Xcomp:编译模式

-Xmixed:混合模式(默认)

大坑:

-Xms和-Xmx是-XX参数不是-X参数

-Xms 等价于: -XX:InitialHeapSize

-Xmx 等价于: -XX:MaxHeapSize

* 1. 非Stable参数（-XX）

此类参数各个jvm实现会有所不同，将来可能会随时取消，需要慎重使用

* + 1. Boolean类型参数

DisableExplicitGC

>禁止调用System.gc()；但jvm的gc仍然有效

-XX:-DisableExplicitGC：关闭功能

-XX:+DisableExplicitGC：开启功能

#### 其他参数

-XX:-CITime 打印消耗在JIT编译的时间

-XX:-ExtendedDTraceProbes 开启solaris特有的dtrace探针

-XX:-HeapDumpOnOutOfMemoryError 当首次遭遇OOM时导出此时堆中相关信息

-XX:-PrintClassHistogram 遇到Ctrl-Break后打印类实例的柱状信息，与jmap -histo功能相同

-XX:-PrintConcurrentLocks 遇到Ctrl-Break后打印并发锁的相关信息，与jstack -l功能相同

-XX:-PrintCommandLineFlags 打印在命令行中出现过的标记

-XX:-PrintCompilation 当一个方法被编译时打印相关信息

-XX:-PrintGC 每次GC时打印相关信息

-XX:-PrintGC Details 每次GC时打印详细信息

-XX:-PrintGCTimeStamps 打印每次GC的时间戳

-XX:-TraceClassLoading 跟踪类的加载信息

-XX:-TraceClassLoadingPreorder 跟踪被引用到的所有类的加载信息

-XX:-TraceClassResolution 跟踪常量池

-XX:-TraceClassUnloading 跟踪类的卸载信息

-XX:-TraceLoaderConstraints 跟踪类加载器约束的相关信息

* + 1. KV键值对类型

公式: -XX:属性名key=属性值value

-XX:MetaspaceSize=128M #元数据空间大小

-XX:MaxTenuringThreshold=15 #年轻代中的对象经历过多少次GC后升级到老年代

-Xms 等价于: -XX:InitialHeapSize # 堆内存初始化大小

-Xmx 等价于: -XX:MaxHeapSize # 堆内存最大内存空间大小

* + 1. jinfo查看当前运行程序的配置

1. 使用jps查看java进程

2. jinfo -flag 参数名 pid

3. jinfo -flag MaxTenuringThreshold PID

4. jinfo -flags PID #查看所有-XX参数所有的参数和值

jifno -flags PID详细信息解读

Non-default VM flags表示默认的配置

Command line:表示手动配置的参数

`

[root@iZ8vbesbzj30ty24cvijhvZ ~]# jinfo -flags 12582

Attaching to process ID 12582, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 25.161-b12

Non-default VM flags: -XX:+AlwaysPreTouch -XX:CICompilerCount=2 -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=16777216 -XX:G1ReservePercent=25 -XX:GCLogFileSize=31457280 -XX:InitialHeapSize=218103808 -XX:InitiatingHeapOccupancyPercent=30 -XX:MarkStackSize=4194304 -XX:MaxDirectMemorySize=16106127360 -XX:MaxHeapSize=218103808 -XX:MaxNewSize=100663296 -XX:MinHeapDeltaBytes=16777216 -XX:NewSize=104857600 -XX:NumberOfGCLogFiles=5 -XX:-OmitStackTraceInFastThrow -XX:+PrintAdaptiveSizePolicy -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:SoftRefLRUPolicyMSPerMB=0 -XX:-UseBiasedLocking -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC -XX:+UseGCLogFileRotation -XX:-UseLargePages

Command line: -Xms200m -Xmx200m -Xmn100m -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerMB=0 -verbose:gc -Xloggc:/dev/shm/mq\_gc\_%p.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintAdaptiveSizePolicy -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSize=30m -XX:-OmitStackTraceInFastThrow -XX:+AlwaysPreTouch -XX:MaxDirectMemorySize=15g -XX:-UseLargePages -XX:-UseBiasedLocking -Djava.ext.dirs=/jre/lib/ext:/opt/rocketmq-all-4.4.0/distribution/target/apache-rocketmq/bin/../lib

`

1. GC
   1. 什么是GC?

GC(Garbage Collection)：即垃圾回收器，诞生于1960年MIT的Lisp语言，主要是用来回收，释放垃圾占用的空间。

java GC泛指java的垃圾回收机制，该机制是java与C/C++的主要区别之一，我们在日常写java代码的时候，一般都不需要编写内存回收或者垃圾清理的代码，也不需要像C/C++那样做类似delete/free的操作。

* 1. 为什么需要学习GC?

对象的内存分配在java虚拟机的自动内存分配机制下，一般不容易出现内存泄漏问题。但是写代码难免会遇到一些特殊情况，比如OOM神马的。。尽管虚拟机内存的动态分配与内存回收技术很成熟，可万一出现了这样那样的内存溢出问题，那么将难以定位错误的原因所在。

学习GC的角度

1. 哪些内存要回收
2. 什么时候回收
3. .怎么回收
   1. 如何判断JVM对象是否可回收

1. 引用计数法

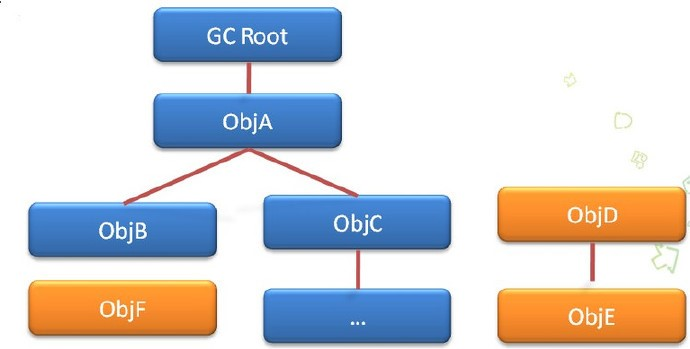
引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

优点：引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0。

2. 可达性分析算法

可达性分析算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点，无用的节点将会被判定为是可回收的对象。



在Java语言中，可作为GC Roots的对象包括下面几种：

  a) 虚拟机栈中引用的对象（栈帧中的本地变量表）；

  b) 方法区中类静态属性引用的对象；

  c) 方法区中常量引用的对象；

  d) 本地方法栈中JNI（Native方法）引用的对象。

* 1. 哪些内存需要回收?

java内存模型中分为五大区域已经有所了解。我们知道程序计数器、虚拟机栈、本地方法栈，由线程而生，随线程而灭，其中栈中的栈帧随着方法的进入顺序的执行的入栈和出栈的操作，一个栈帧需要分配多少内存取决于具体的虚拟机实现并且在编译期间即确定下来【忽略JIT编译器做的优化，基本当成编译期间可知】，当方法或线程执行完毕后，内存就随着回收，因此无需关心。

而Java堆、方法区则不一样。方法区存放着类加载信息，但是一个接口中多个实现类需要的内存可能不太一样，一个方法中多个分支需要的内存也可能不一样【只有在运行期间才可知道这个方法创建了哪些对象没需要多少内存】，这部分内存的分配和回收都是动态的，gc关注的也正是这部分的内存。

Java堆是GC回收的“重点区域”。堆中基本存放着所有对象实例，gc进行回收前，第一件事就是确认哪些对象存活，哪些死去[即不可能再被引用]

* 1. 堆的回收区域

为了高效的回收，jvm将堆分为三个区域

1.新生代（Young Generation）NewSize和MaxNewSize分别可以控制年轻代的初始大小和最大的大小

2.老年代（Old Generation）

3.永久代（Permanent Generation）【1.8以后采用元空间，就不在堆中了】



* 1. 方法区GC

猿们，方法区存储内容是否需要回收的判断可就不一样咯。方法区主要回收的内容有：废弃常量和无用的类。对于废弃常量也可通过引用的可达性来判断，但是对于无用的类则需要同时满足下面3个条件：

该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例；

加载该类的ClassLoader已经被回收；

该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

* 1. GC 为什么分代回收

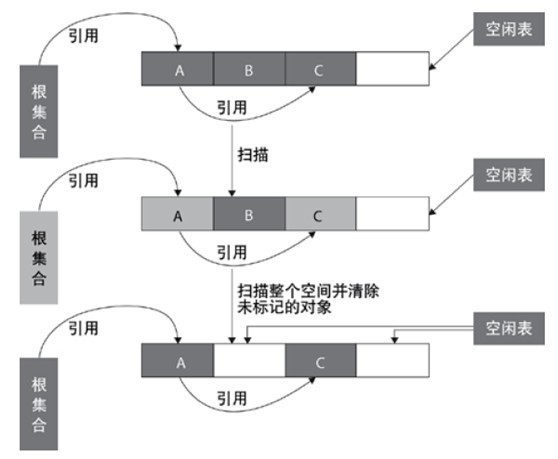
堆内存是虚拟机管理的内存中最大的一块，也是垃圾回收最频繁的一块区域。当然了，我们写的代码中new的对象等都存在堆内存中。

如果说堆内存没有区域划分，所有新创建的对象和生命周期很长的对象放在一个区域，随着对象越来越多触发了JVM的垃圾回收机制，而每次回收都要遍历所有的对象，这个时间成本是难以想象的，严重影响GC效率。

而有了内存分代，新创建的对象会在新生代中分配内存，而经过多次回收仍然存活下的对象或者有特殊情况的对象会移至老年代，类信息、静态变量等信息存放在永久代里。新生代中的对象存活时间短，只需要在新生代区域中频繁进行GC，而老年代中对象存活时间长，回收的频率相对就低很多了，永久代则很少进行垃圾回收。所以呢，给堆内存分代是为了提高对象内存分配和垃圾回收的效率。

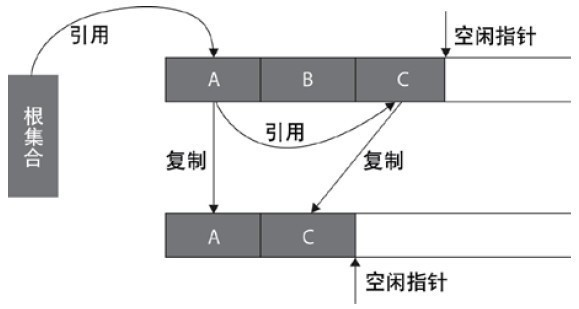
* 1. GC算法
     1. 标记-清除算法

标记-清除算法采用从根集合（GC Roots）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，如下图所示。标记-清除算法不需要进行对象的移动，只需对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。



* + 1. 复制算法

  复制算法的提出是为了克服句柄的开销和解决内存碎片的问题。它开始时把堆分成 一个对象 面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于copying算法的垃圾 收集就从根集合（GC Roots）中扫描活动对象，并将每个 活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。



* + 1. 标记-整理算法
    2. JAVA引用
       1. 强引用
       2. 软引用
       3. 弱引用
       4. 虚引用
    3. 分代收集算法
       1. 年轻代（Young Generation）的回收算法

a) 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

b) 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。

c) 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收。

d) 新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)。

* + - 1. 年老代（Old Generation）的回收算法

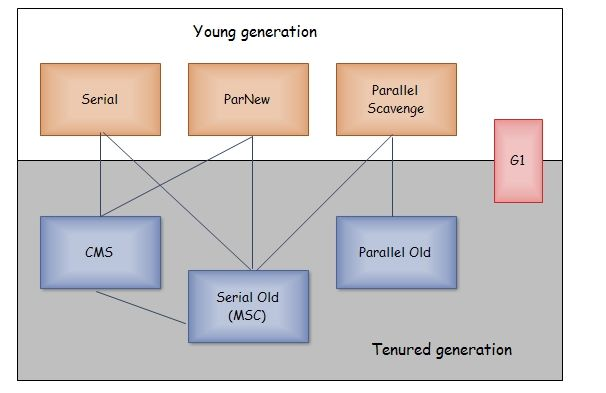
a) 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

b) 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

* + - 1. 持久代（Permanent Generation）的回收算法

用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代也称方法区，jdk1.7叫持久代也叫永久代jdk1.8叫元数据空间Metaspace

* + 1. 垃圾收集器



1. 图中展示了7种不同分代的收集器：

Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1；

2. 而它们所处区域，则表明其是属于新生代收集器还是老年代收集器：

新生代收集器：Serial、ParNew、Parallel Scavenge；

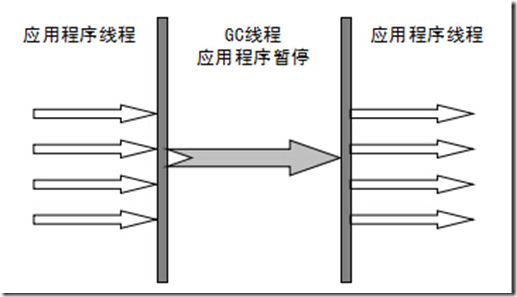
老年代收集器：Serial Old、Parallel Old、CMS；

整堆收集器：G1；

3. 两个收集器间有连线，表明它们可以搭配使用：

Serial/Serial Old、Serial/CMS、ParNew/Serial Old、ParNew/CMS、Parallel Scavenge/Serial Old、Parallel Scavenge/Parallel Old、G1；

* + - 1. Serial垃圾收集器



Serial收集器是单线程收集器，是分代收集器。它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束(Stop TheWorld)。

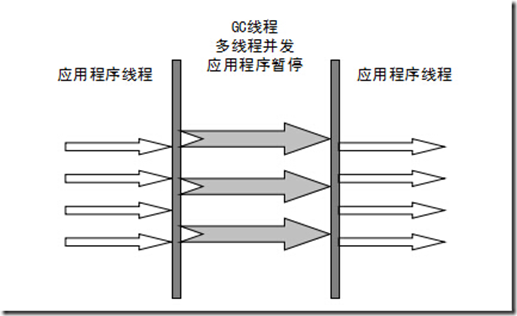
新生代：单线程复制收集算法；

老年代：单线程标记整理算法。

Serial一般在单核的机器上使用，是Java 5非服务端JVM的默认收集器，参数-XX:UseSerialGC设置使用。

优势：对于单CPU环境来说，Serial收集器没有线程交互的开销，专心做垃圾收集可以获得最高的单线程收集。Serial收集器对于在Client模式下的虚拟机是一个很好的选择。

* + - 1. ParNew垃圾收集器



ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

参数控制：-XX:+UseParNewGC ParNew收集器

-XX:ParallelGCThreads 限制线程数量

* + - 1. Parallel垃圾收集器

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

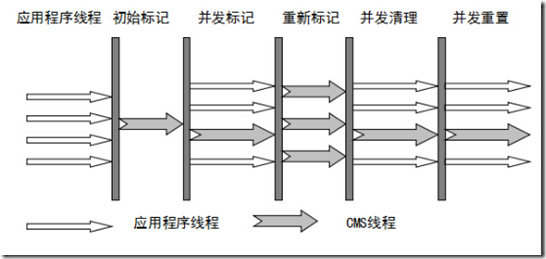
参数控制：-XX:+UseParallelGC 使用Parallel收集器+ 老年代串行

* + - 1. Parallel Old垃圾收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在JDK 1.6中才开始提供

参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

* + - 1. CMS垃圾收集器



CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为4个步骤，包括：

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则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对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用ParNew）

优点:并发收集、低停顿

缺点：产生大量空间碎片、并发阶段会降低吞吐量

参数控制：-XX:+UseConcMarkSweepGC 使用CMS收集器

-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长

-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理

-XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）

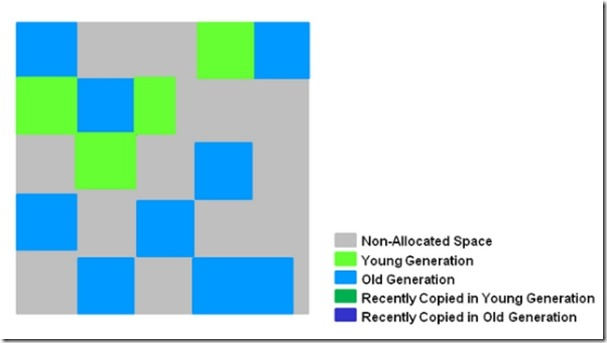
* + - 1. G1垃圾收集器

G1是目前技术发展的最前沿成果之一，HotSpot开发团队赋予它的使命是未来可以替换掉JDK1.5中发布的CMS收集器。与CMS收集器相比G1收集器有以下特点：

1. 空间整合，G1收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次GC。

2. 可预测停顿，这是G1的另一大优势，降低停顿时间是G1和CMS的共同关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为N毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

上面提到的垃圾收集器，收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region的集合。



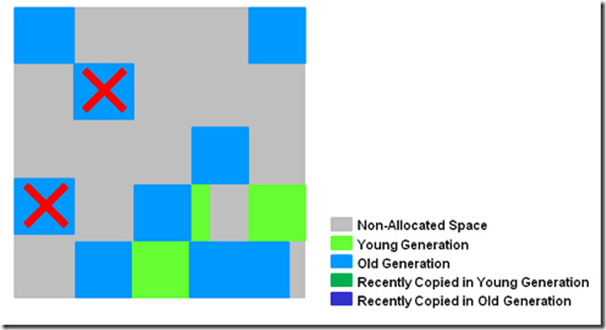
G1的新生代收集跟ParNew类似，当新生代占用达到一定比例的时候，开始出发收集。和CMS类似，G1收集器收集老年代对象会有短暂停顿。

收集步骤：

1、标记阶段，首先初始标记(Initial-Mark),这个阶段是停顿的(Stop the World Event)，并且会触发一次普通Mintor GC。对应GC log:GC pause (young) (inital-mark)

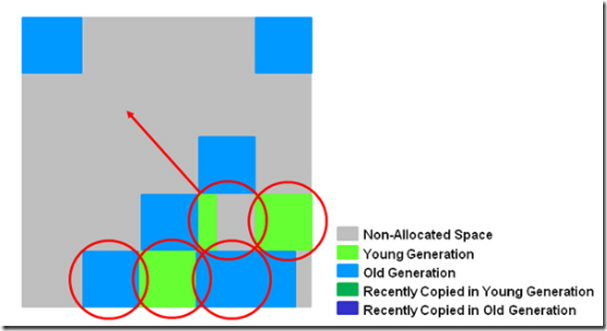
2、Root Region Scanning，程序运行过程中会回收survivor区(存活到老年代)，这一过程必须在young GC之前完成。

3、Concurrent Marking，在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被young GC中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那个这个区域会被立即回收(图中打X)。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。

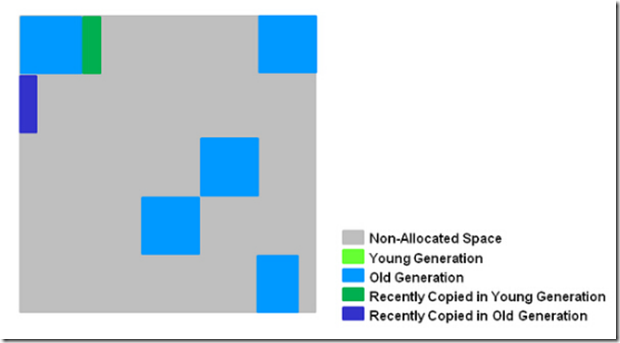


4、Remark, 再标记，会有短暂停顿(STW)。再标记阶段是用来收集 并发标记阶段 产生新的垃圾(并发阶段和应用程序一同运行)；G1中采用了比CMS更快的初始快照算法:snapshot-at-the-beginning (SATB)。

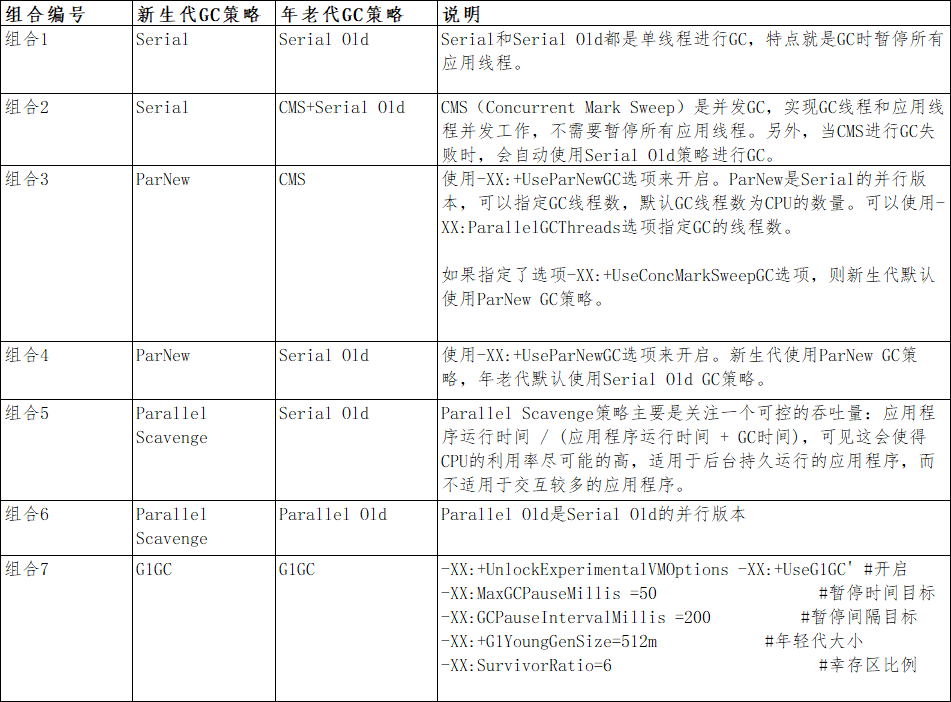
5、Copy/Clean up，多线程清除失活对象，会有STW。G1将回收区域的存活对象拷贝到新区域，清除Remember Sets，并发清空回收区域并把它返回到空闲区域链表中。



6、复制/清除过程后。回收区域的活性对象已经被集中回收到深蓝色和深绿色区域。



* + - 1. 常用的收集器组合



* 1. 什么时候进行GC?

Minor GC ，Full GC 触发条件

* 从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC；
* 对老年代GC称为Major GC；
* 而Full GC是对整个堆来说的；

在最近几个版本的JDK里默认包括了对永生带即方法区的回收（JDK8中无永生带了），出现Full GC的时候经常伴随至少一次的Minor GC,但非绝对的。Major GC的速度一般会比Minor GC慢10倍以上。下边看看有那种情况触发JVM进行Full GC及应对策略。

Minor GC触发条件：

当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）System.gc()方法的调用(编程规范中禁止调用System.gc())

此方法的调用是建议JVM进行Full GC,虽然只是建议而非一定,但很多情况下它会触发 Full GC,从而增加Full GC的频率,也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存，可通过通过-XX:+ DisableExplicitGC来禁止RMI（Java远程方法调用）调用System.gc。

（2）老年代空间不足

旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误： java.lang.OutOfMemoryError: Java heap space 为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

（3）方法区空间不足

JVM规范中运行时数据区域中的方法区，在HotSpot虚拟机中又被习惯称为永生代或者永生区，Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下也会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：

java.lang.OutOfMemoryError: PermGen space

为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

如果发现统计数据说之前Minor GC的平均晋升大小比目前old gen剩余的空间大，则不会触发Minor GC而是转为触发full GC

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存

小于该对象大小

* 1. 新生代晋身老年代方式
* Eden区满时，进行Minor GC，当Eden和一个Survivor区中依然存活的对象无法放入到Survivor中，则通过分配担保机制提前转移到老年代中。
* 若对象体积太大, 新生代无法容纳这个对象，-XX:PretenureSizeThreshold即对象的大小大于此值, 就会绕过新生代, 直接在老年代分配, 此参数只对Serial及ParNew两款收集器有效。
* 长期存活的对象将进入老年代。

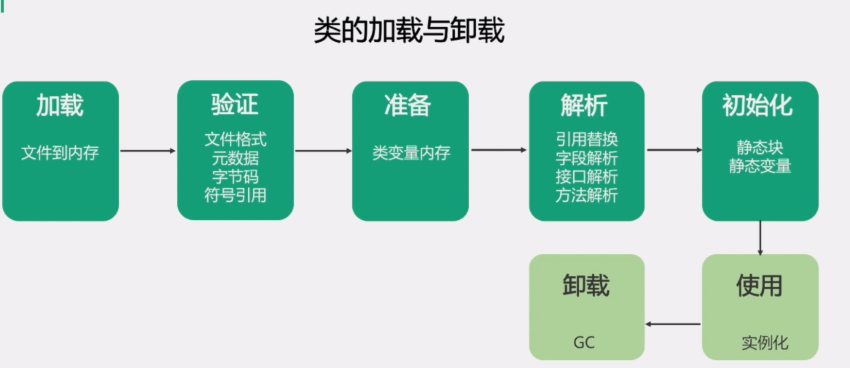
虚拟机对每个对象定义了一个对象年龄（Age）计数器。当年龄增加到一定的临界值时，就会晋升到老年代中，该临界值由参数：-XX:MaxTenuringThreshold来设置。

如果对象在Eden出生并在第一次发生MinorGC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；以后每经历一次Minor GC，该对象还存活的话Age=Age+1。

* 动态对象年龄判定。

虚拟机并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

1. 编译器优化
   1. 公共子表达式的消除
   2. 指令重排
   3. 内联
   4. 逃逸分析
      1. 方法逃逸
      2. 线程逃逸
   5. 栈上分配
   6. 同步消除
2. 类加载
   1. 类加载流程



类的加载指的是将编译好的class类文件中的字节码读入到内存中，将其放在方法区内并创建对应的Class对象。

类的加载分为加载、链接、初始化，其中链接又包括验证、准备、解析三步。看到图中上半部分深绿色，我们逐个分析：

加载是文件到内存的过程。通过类的完全限定名查找此类字节码文件，并利用字节码文件创建一个Class对象

验证是对类文件内容验证。目的在于确保Class文件符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种：文件格式验证，元数据验证，字节码验证，符号引用验证。

准备阶段是进行内存分配。为类变量也就是类中由static修饰的变量分配内存，并且设置初始值，这里要注意，初始值是0或者null，而不是代码中设置的具体值，代码中设置的值是在初始化阶段完成的。另外这里也不包含用final修饰的静态变量，因为final在编译的时候就会分配了。

解析主要是解析字段、接口、方法。主要是将常量池中的符号引用替换为直接引用的过程。直接引用就是直接指向目标的指针、相对偏移量等。

最后是初始化：主要完成静态块执行与静态变量的赋值。这是类加载最后阶段，若被加载类的父类没有初始化，则先对父类进行初始化。

只有对类主动使用时，才会进行初始化，初始化的触发条件包括创建类的实例的时候、访问类的静态方法或者静态变量的时候、Class.forName()反射类的时候、或者某个子类被初始化的时候。

类的生命周期，就是从类的加载到类实例的创建与使用，再到类对象不再被使用时可以被GC卸载回收。这里要注意一点，由java虚拟机自带的三种类加载器加载的类在虚拟机的整个生命周期中是不会被卸载的，只有用户自定义的类加载器所加载的类才可以被卸载。

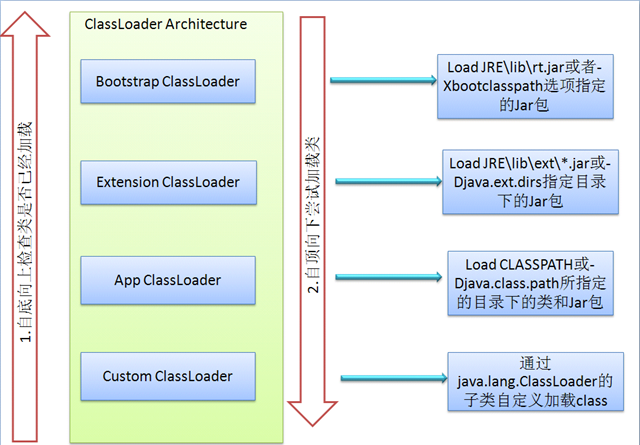
* 1. 双亲委派机制

1. 双亲委派的流程

2. 双亲委派的好处

1. 避免类的重复加载

2. 避免java的核心api被篡改



Bootstrap ClassLoader 最顶层的加载类，主要加载核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。另外需要注意的是可以通过启动jvm时指定-Xbootclasspath和路径来改变Bootstrap ClassLoader的加载目录。比如java -Xbootclasspath/a:path被指定的文件追加到默认的bootstrap路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些jar包是不是存在于这个目录。

Extention ClassLoader 扩展的类加载器，加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。

Appclass Loader也称为SystemAppClass 加载当前应用的classpath的所有类。

* 1. Bootstrap类加载器
  2. ExtClassLoader
  3. AppClassLoader(系统类加载器)
  4. 自定义类加载器
  5. loadClass和forName的区别