# Java代码执行流程/Java虚拟机工作原理？

Java 代码执行流程主要划分为以下5个步骤：编辑源代码、编译生成class文件、加载class文件、运行class文件、垃圾回收。

1源码编写

编辑源代码，并命名为Student.java。

2编译

输入javac Student.java/使用Java源码编译器编译。将源代码文件编译生成student.class字节码文件。字节码文件存放了这个类的字段、方法、父类、实现的接口等各种信息。

3类加载

通过类加载器将.class二进制数据读入内存中。即读取student.class文件中数据并存储在方法区中，并建立一个Class对象，作为运行时方法Student类各种数据的接口。

一个类文件加载到方法区，一些符号引用被解析（静态解析：如class文件的常量池被加载到方法区运行时常量池，各种其他静态存储结构被加载为方法区运行时数据结构等等）为直接引用或等到运行时分派（动态绑定），经过一系列加载过程后，程序可以通过Class对象访问方法区各种类型数据。

4JVM执行

JVM执行.class文件。执行引擎找到main()入口方法，即在栈里创建一个栈帧，逐行执行方法中的字节码指令。操作完成后方法返回给调用方，栈帧出栈。

当前正在运行的方法的栈帧位于栈顶。若当前方法返回，则当前方法对应的栈帧出栈；当前方法的方法体中若是调用了其他方法，则为被调用方法创建栈帧，并将其压入栈顶。

5垃圾回收

内存空间通过JVM垃圾回收机制GC

# Java内存结构 / 运行时数据区域？

Java内存结构描述的是Java程序执行过程中, 由JVM管理的不同的数据区域。包括以下5部分：

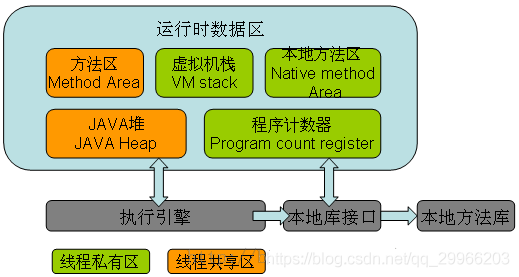
线程共享：堆内存（heap）、方法区（method）

线程私有：程序计数器、栈内存（stack）、本地方法栈（java中JNI调用）

1. 堆内存（线程共享）：JVM所管理的内存中最大一块。唯一目的就是存放实例对象，几乎所有的对象实例都在这里分配。Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。异常状态 OutOfMemoryError
2. 方法区（线程共享）：方法区是被所有线程共享的区域。用于存放类的所有信息（字段、方法、构造函数等）、静态变量、常量。异常状态 OutOfMemoryError

包含运行时常量池：存放编译器生成的各种字面量和符号引用

1. （虚拟机）栈内存（线程私有）：一个线程对应一个栈，生命周期与线程相同。描述的是java方法执行的内存模型：每个方法执行时会创建一个栈帧，用于存放局部变量、操作数栈、方法出口等信息。每一个方法从调用直至完成的过程，对应着一个栈帧在虚拟机中入栈到出栈的过程。异常状态 OutOfMemoryError StackOverflowError
2. 本地方法栈（线程私有）：与虚拟机栈作用相似，区别在于本地方法栈用于支持Native方法执行，存储了每个Native方法调用的状态。
3. 程序计数器（线程私有）：可看做当前线程所执行的字节码的行号指示器。指向方法区方法字节码（下一个指令的地址），并由执行引擎读取并执行下一指令。



Java中的常量池，实际上分为两种形态：静态常量池和运行时常量池。

所谓静态常量池，即\*.class文件中的常量池，这种常量池主要用于存放两大类常量：字面量(Literal)和符号引用量(Symbolic References)

字面量相当于Java语言层面常量的概念，如文本字符串，声明为final的常量值等。

符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：

（1）类和接口的全限定名

（2）字段名称和描述符

（3）方法名称和描述符

而运行时常量池，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指方法区中的运行时常量池。

常量池是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中。（jdk1.7后从运行时常量池移到了堆中）

# 对象的创建、内存布局 和 访问定位

略

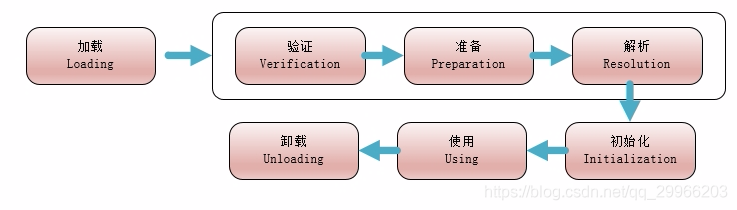
# 类加载机制

把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型。

在Java语言里，类型的加载、连接和初始化过程都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的灵活性，Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点来实现的。

# 类的生命周期

加载，验证，准备，解析，初始化，使用和卸载。其中验证，准备，解析3个部分统称为连接。  
这7个阶段发生顺序如下图：



加载，验证，准备，初始化，卸载这5个阶段的顺序是确定的，而解析阶段则不一定：它在某些情况下可以在初始化完成后在开始，这是为了支持Java语言的运行时绑定。  
其中加载，验证，准备，解析及初始化是属于类加载机制中的步骤。注意此处的加载不等同于类加载。

# 触发类加载的条件

①.遇到new,getstatic,putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候，读取或设置一个类的静态字段的时候（被final修饰，已在编译期把结果放入常量池的静态字段除外），以及调用一个类的静态方法的时候。

②.使用java.lang.reflect包的方法对类进行反射调用的时候。

③.当初始化一个类的时候，发现其父类还没有进行过初始化，则需要先出发父类的初始化。

④.当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

⑤.当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic,REF\_putStatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出发初始化。

# 类加载的具体过程

略

# 类加载器分类

不同的类加载器负责加载不同的类。主要分为两类。

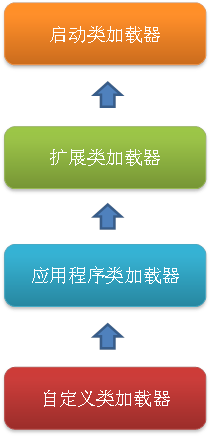
1启动类加载器（Bootstrap ClassLoader）： 由C++语言实现（针对HotSpot）,负责将存放在\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中，即负责加载Java的核心类。

2其他类加载器： 由Java语言实现，继承自抽象类ClassLoader。如：

扩展类加载器（Extension ClassLoader）： 负责加载\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库，即负责加载Java扩展的核心类之外的类。

应用程序类加载器（Application ClassLoader）： 负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器，通过ClassLoader.getSystemClassLoader()方法直接获取。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

以上2大类，3小类类加载器基本上负责了所有Java类的加载。下面我们来具体了解上述几个类加载器实现类加载过程时相互配合协作的流程。



# 双亲委派模型

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

这样的好处是不同层次的类加载器具有不同优先级，比如所有Java对象的超级父类java.lang.Object，位于rt.jar，无论哪个类加载器加载该类，最终都是由启动类加载器进行加载，保证安全。即使用户自己编写一个java.lang.Object类并放入程序中，虽能正常编译，但不会被加载运行，保证不会出现混乱。

# 垃圾回收机制

如何判断对象是否会被垃圾回收机制处理掉？

1. ****引用计数法****

* 原理  
  假设有一个对象A，任何一个对象对A的引用，那么对象A的引用计数器+1，当引用失败时，对象A的引用计数器-1，如果对象A的计数器的值为0，说明A没有引用，可以被回收。
* 特点  
  无法解决循环引用问题。

1. ****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，可达的对象都是存活的。当所有的引用节点寻找完毕之后，剩余的不可达节点则被认为是没有被引用到的节点，即无用的节点（GC Root 不可达对象），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：

1. 虚拟机栈中引用的对象（栈帧中的本地变量表）
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI（Native方法）引用的对象
5. ****finalize —— 对象死亡（被回收）前的最后一次挣扎/自救计划****

即使在可达性分析算法中不可达的对象，也并非是“非死不可”，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。

第一次标记：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记；

第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。即：当对象没有被覆盖finalize()或者finalize()方法已经被虚拟机调用过了，虚拟机将这两种情况都是为没有必要执行。

第二次标记：当一个对象被判断为有必要执行finalize()方法，那么这个对象会被放置到F-Queue队列中，并且稍后JVM自动建立一个低优先级的Finalizer线程执行它，这里“执行”是虚拟机会触发这个方法，但不会承诺等待它运行结束（万一这个方法运行缓慢或者死循环，F-Queue队列其他对象岂不是永久等待）。finalize()是对象逃脱死亡的最后一次机会。稍后GC会对F-Queue进行第二次小规模标记。如果对象能在finalize()方法中重新与引用链上任何一个方法建立关联（例如把自己this关键字赋值给某个类变量或者对象的成员变量）。那么第二次标记时，将会移出即将回收的集合。否则，这个对象就会被回收了。

# 说说Java种的4种引用以及用法？

1. 强引用  
   被强引用关联的对象不会被垃圾回收器回收。使用 new 一个新对象的方式来创建强引用。
2. 软引用  
   用来描述一些还有用但并非必须的对象。被软引用关联的对象只有在内存不够的情况下才会被回收。使用 SoftReference 类来创建软引用。
3. 弱引用  
   用来描述无用对象的，被弱引用关联的对象只要被垃圾回收器扫描到，无论内存是否足够，就一定会回收，即被弱引用关联的对象只能生存到下一次垃圾收集发生之前。使用 WeakReference 类来创建弱引用。
4. 虚引用  
   也叫幽灵引用或幻影引用，是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。它的唯一作用是能在这个对象被收集器回收时收到一个系统通知。使用 PhantomReference 来创建虚引用。

# GC回收算法有哪些？优缺点？

1. ****标记清除法****

* 原理  
  标记清除算法，是将垃圾回收分为2个阶段，分别是标记和清除。  
  标记：从根节点开始标记引用的对象。清除：未被标记引用的对象是垃圾对象，可以被清理。
* 特点  
  标记清除算法解决了循环引用问题（没有从root节点引用的对象都会被回收）  
  但通过清除算法清理出来的内存，碎片化较为严重。因为被回收的对象可能存在于内存的各个角落，所以清理后的空闲内存不连贯。

1. ****标记压缩法****

* 原理  
  标记压缩算法是标记清除算法的基础上，做了优化改进的算法。和标记清除算法一样，也是从根节点开始，对对象的引用进行标记，在清理阶段，并不是简单的清理未标记对象，而是将存活的对象压缩到内存的一段，然后清理边界以外的垃圾，从而解决了碎片化的问题。
* 特点  
  解决了标记清除算法的碎片化问题。  
  但是标记压缩算法多了一步，对象移动内存位置的步骤，其效率有一定影响。

1. ****复制算法****

原理  
复制算法的核心就是，将原有的内存空间一分为二，每次只用其中的一块，在垃圾回收时，将正在使用的对象复制到另一个内存空间中，然后将该内存空间清空，交换两个内存的角色，完成垃圾的回收。

如果内存中的垃圾对象较多，需要复制的对象就较少，这种情况下适合使用该方法并且效率较高，反之，则不适合。

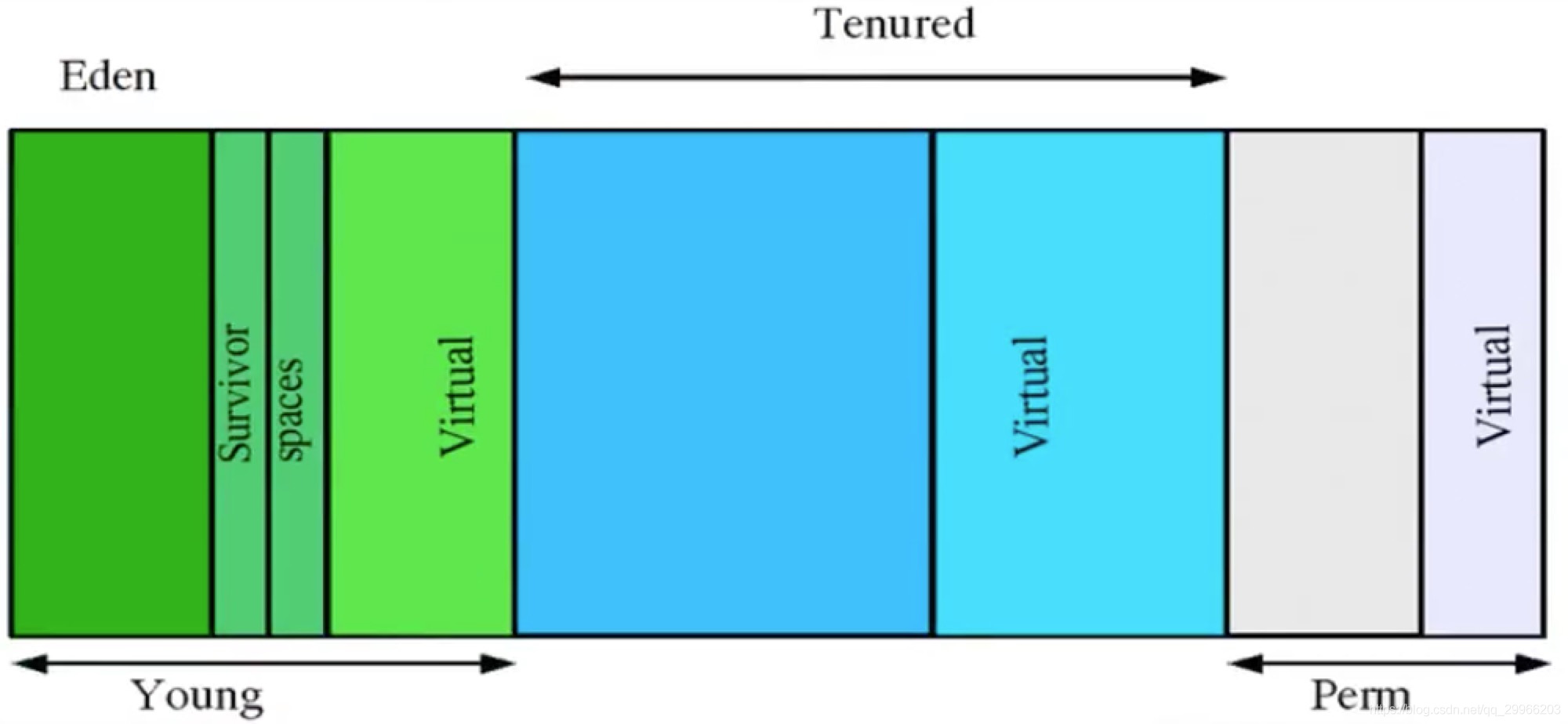
* 特点  
  适用于垃圾对象较多的情况，且清理后，内存无碎片。但不适用于垃圾对象少的情况（如老年代内存），且分配的2块内存空间，在同一个时刻，只能使用一半，内存使用率较低。

# GC回收机制？

（1）Java垃圾回收

程序的运行必然需要申请内存资源，无效的对象资源如果不及时处理就会一直占有内存资源，最终导致内存溢出。为了让程序员更专注于代码的实现，而不用过多考虑内存释放的问题（内存的释放由系统自动识别完成）。Java语言中有了自动的垃圾回收机制，即GC回收机制。

1. ****JVM堆内存结构（分代模型）****  
   ****（2.1）jdk1.7 堆内存模型****



Young 年轻代

年轻代被划分为三部分，Eden区和两个大小严格相同的Survivor区，其中，Survivor区间中，某一时刻只有其中一个（Survivor\_From）是被使用的，另一个（Survivor\_To）留作垃圾收集时赋值对象使用。几乎所有新生成的对象首先是放在Eden区间。

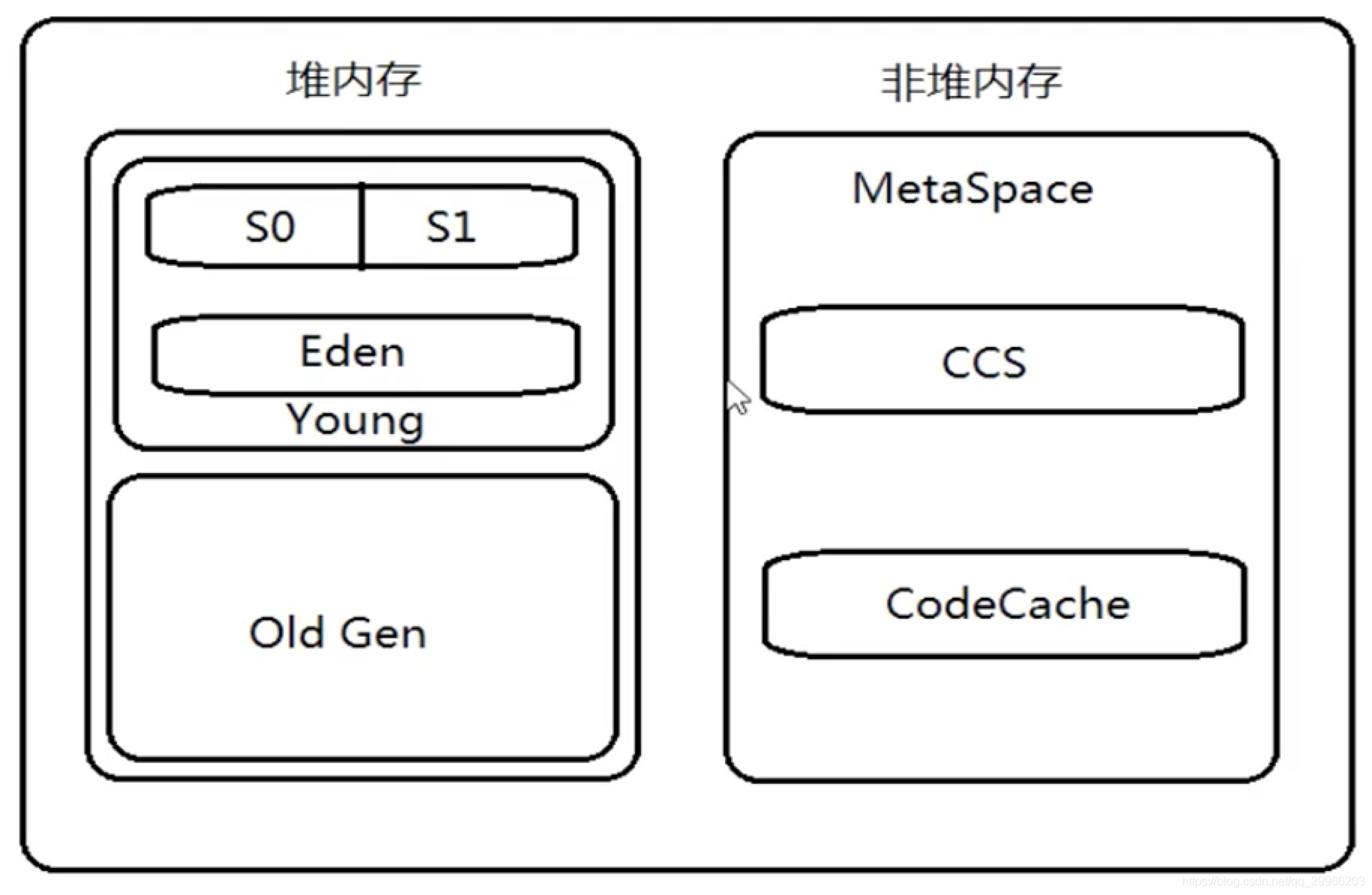
Tenured 老年代

Tenured区主要保存生命周期长的对象，一般是一些存活时间长的对象（如较大的对象直接分配老年代），或当一些对象在Young区复制转移一定的次数以后，对象就会被转移到Tenured区。因此老年代对象存活时间比较长，存活率高。

Perm 永久代

Perm代主要保存class,method,field对象。

****（2.2）jdk1.8 堆内存模型****



由上图可以看出，jdk 1.8 的内存模型由2部分组成，年轻代+老年代。

年轻代：Eden + 2\*Survivor

年老代：Old

在jdk 1.8 中变化最大的Perm区，用Metaspace（元数据空间）进行了替换。其中，Metaspace所占用的内存空间不是在虚拟机内部，而是在本地内存空间中。（防止由于永久代内存经常不够用或发生内存泄露，改用本地内存空间）

1. ****GC回收算法（分代算法）****

分代收集，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，可以将不同生命周期的对象分代，不同的代采取不同的回收算法进行垃圾回收（GC），以便提高回收效率。

Young 年轻代（复制算法）

由于年轻代垃圾对象较多，复制的对象较少，因此复制算法效率较高。年轻代划分为一块较大的Eden区和两块较小的Survivor区（Survivor\_From、Survivor\_To）

1. 几乎所有新生成的对象首先是放在Eden区间。当Eden区间内存不足时，会发起一次GC。
2. 在GC开始的时候，对象只会存在于Eden区和Survivor\_From区，Survivor\_To区是空的。
3. 紧接着进行GC，Eden区所有存活对象都会复制到Survivor\_To区，Survivor\_From区中仍存活的对象根据他们的年龄决定去向，若年龄达到年龄阀值（MaxTenuringThreshold）的对象会移动到老年代，没有到达阀值的对象会复制到Survivor\_To区。
4. 清空Eden区和Survivor\_From区。然后交换Survivor\_To与Survivor\_From区。使存活的对象均位于Survivor\_From区域，Survivor\_To区为空。
5. GC会一直重复上述过程，直到Survivor\_To区被填满（Survivor\_To区不足以存放Eden与Survivor\_From区中存活的对象时），此时所有存活的对象移动到老年代。

* Tenured 老年代（标记-清除/压缩法）  
  由于老年代垃圾对象较少（对象存活率较高），因此适合使用标记-清除/压缩法。  
  当老年代内存满时触发GC。

# GC是什么时候触发的？

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Minor GC和Full GC。

1. Minor GC

这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为年轻代对象存活时间很短，所以Minor GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

触发条件：

一般情况下，当新对象生成，会在年轻代Eden上分配，若Eden区域已满，申请空间失败时，就会触发Minor GC：对Eden区域进行GC，在回收时，将 Eden 和 Survivor\_From 中还存活着的对象全部复制到 Survivor\_To 上，然后清理 Eden 和 Survivor\_From，最后将Survivor\_To中对象移动到 Survivor\_From。

1. Full GC

对整个堆进行整理，包括回收年轻代、老年代和持久代。Full GC因为需要对整个堆进行回收，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数。

触发条件：

年老代（Tenured）被写满；

持久代（Perm）被写满；

System.gc()被显示调用；

上一次GC之后Heap的各域分配策略动态变化；

# 垃圾收集器？

* 单线程与多线程：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
* 串行与并行：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；并行指的是垃圾收集器和用户程序同时执行。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

Serial 垃圾收集器

串行垃圾收集器，是指使用单线程进行垃圾回收。垃圾回收时，只有一个线程在工作，并且java应用中的所有线程都要暂停，等待垃圾回收的完成。这种现象称之为STW（Stop-The-World）

对于交互性交强的应用而言，这种垃圾收集器是不能够接受的。一般在javaweb应用中是不会采用该收集器的。在G1的FULL GC采用Serial GC进行回收。

Parallel 垃圾收集器

并行垃圾收集器在串行垃圾收集器的基础上做了改进，将单线程改为了多线程进行垃圾回收，这样可以缩短垃圾回收的时间。并行垃圾收集器在收集的过程中也会暂停应用程序，与串行垃圾收集器是一样的，只是并行执行，速度更快，暂停时间更短。分为：

ParNew 垃圾收集器

ParNew收集器是工作在年轻代上的，只是将串行的垃圾收集器改为了并行。ParNew GC可以与CMS GC合作使用

ParallelGC 垃圾收集器

ParallelGC收集器工作机制和ParNewGC收集器一样，不同在于新增了可以对老年代进行多线程垃圾回收，且增加了对吞吐量（=运行用户代码时间 / CPU总消耗时间（运行用户代码时间+垃圾收集时间））的控制。

高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

****CMS 收集器****  
CMS全称Concurrent Mark Sweep，是一款并发的、使用标记-清除算法的垃圾回收器，该回收器是针对老年代垃圾回收的，CMS是HotSpot虚拟机中第一款真正意义上的****并发****收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。

CMS垃圾回收器的执行过程如下：

（1）初始化标记 CMS Initial mark：会导致stw，标记一下GC Roots能直接关联到的对象，速度很快。

（2）并发标记 CMS-concurrent-mark：与用户线程同时运行，进行GC Roots Tracing的过程。

（3）预清理 CMS-concurrent-preclean：与用户线程同时运行，用于修正并发标记。

（4）重新标记 CMS Final Remark：会导致stw，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（产生新的/释放旧的 对象或引用关系）的标记几率（这个阶段的停顿时间一般会比初始标记阶段稍长，但远比并发标记时间短）。

（5）并发清除 CMS-concurrent-sweep：与用户线程同时运行，清除标记对象（采用标记-清除）。

（6）调整堆大小，设置CMS在清理之后进行内存压缩，目的是清理内存中的碎片。

（7）并发重置状态 CMS-concurrent-reset：与用户线程同时运行，等待下次CMS的触发。

年轻代默认使用ParNew收集器，老年代使用CMS收集器。

****G1 收集器（重要）****

****G1垃圾收集器是在jdk1.7中正式使用的全新的垃圾收集器，是一种面向服务端应用的垃圾收集器。oracle官方计划在jdk9中将G1编程默认的收集器，以代替CMS。****

****G1的设计原则就是简化JVM性能调优，开发人员只需要简单的三步便可完成调优：****

****第一步：开启G1垃圾收集器****

****第二步：设置堆的最大内存****

****第三步：设置最大停顿时间****

****G1垃圾收集器的原理是：****

****相比其他收集器而言，最大的区别在于G1垃圾收集器取消了年轻代、老年代的物理划分，取而代之将堆划分为若干个区域（Regin），这些区域包含了有逻辑上的年轻代、老年代区域。****

****每个区域被标记了Eden、Survivor、Old和Humongous，在运行时充当相应的角色。H代表Humongous，用于存放占用空间超过分区容量50%以上的大对象。****

****每个Regin都有一个RememberSet，用来记录该Regin对象的引用对象所在Regin。通过使用Remember Set，在做可达性分析时可以避免全堆扫描。****

****G1取消了新生代、老年代物理空间的划分。这样我们再也不用单独地对每个代的空间进行设置，也不用担心每个代地内存是否足够。****

****G1垃圾收集器提供三种垃圾回收模式：****

****Young GC****

****发生在年轻代的GC算法，一般对象（除了巨型对象）都是在Eden Region中分配内存，当所有eden region被耗尽无法申请内存时，就会触发一次Young GC，采用复制算法，执行完一次Young GC，存活对象会被拷贝到Survivor Region或者晋升到Old Region中。****

****Mixed GC****

****当越来越多地对象晋升到老年代Old Regin时候，为了避免堆内存被耗尽，虚拟机会触发一个混合垃圾回收器，即Mixed GC。该算法除了回收整个Young Regin，还会回收一部分Old Regin。****

****Mixed GC执行过程分为以下几个步骤：****

****（1）初始标记 initial mark：STW，标记GC Roots直接关联对象****

****（2）并发标记 concurrent marking：与应用程序并发执行，在整个堆中查找从初始标记衍生出的存活对象****

****（3）最终标记 Remark：STW，修正在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标记记录****

****（4）清除垃圾 Cleanup：STW，采用复制算法进行垃圾回收，将一部分Regin里的存活对象复制到另一部分Regin中。****

* Full GC  
  如果对象内存分配速度过快，Mixed Gc来不及回收，导致老年代被填满，就会触发一次Full GC，G1的Full GC算法就是单线程执行的Serial GC，会导致异常长时间的暂停时间，需要进行不断的调优，尽可能的避免Full GC

# **面试**

# 对象的生命周期?

1. ****创建阶段(Created)****

****在创建阶段系统通过下面的几个步骤来完成对象的创建过程：****

****（1）为对象分配存储空间****

****（2）开始构造对象****

****（3）从超类到子类对static成员进行初始化****

****（4）超累成员变量按顺序初始化，递归调用超累的构造方法****

****（5）子类成员变量按顺序初始化，子类构造方法调用****

****一旦对象被创建，并被分派给某些变量赋值，这个对象的状态就切换到应用状态****

1. ****应用阶段(In Use)****

****对象至少被一个强引用持有。****

1. ****不可见阶段(Invisible)****

****当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该这些引用仍然是存在着的。****

****简单说就是程序的执行已经超出了该对象的作用域了。****

1. ****不可达阶段(Unreachable)****

****对象处于不可达阶段是指该对象不再被任何强引用所持有。****

****与“不可见阶段”相比，“不可见阶段”是指程序不再持有该对象的任何强引用，这种情况下，该对象仍可能被JVM等系统下的某些已装载的静态变量或线程或JNI等强引用持有着，这些特殊的强引用被称为”GC root”。存在着这些GC root会导致对象的内存泄露情况，无法被回收。****

1. ****收集阶段(Collected)****

****当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，则对象进入了“收集阶段”。如果该对象已经重写了finalize()方法，则会去执行该方法的终端操作。****

1. ****终结阶段(Finalized)****

****当对象执行完finalize()方法后仍然处于不可达状态时，则该对象进入终结阶段。在该阶段是等待垃圾回收器对该对象空间进行回收。****

1. ****对象空间重分配阶段(De-allocated)****

****垃圾回收器对该对象的所占用的内存空间进行回收或者再分配了，则该对象彻底消失了，称之为“对象空间重新分配阶段”。****

# **谈谈**[static](https://so.csdn.net/so/search?q=static&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_29966203/article/details/_blank)**关键字/谈谈static编译运行时的流程，在虚拟机中如何保存的？**

1. ****static 关键字****  
   被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。

****static方法****

****static方法一般称作静态方法，静态方法不依赖于任何对象就可以进行访问。因此静态方法不使用this，且在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为静态方法独立于任何对象实例，非静态成员方法/变量都是必须依赖具体的对象才能够被调用，因此：****

****（1）静态方法仅能调用其他static方法****

****（2）静态方法仅能访问static数据****

****（3）静态方法不能引用this或super****

****static变量****

****静态变量是随着类加载时被完成初始化的，它在内存中仅有一个，且JVM也只会为它分配一次内存，可以直接通过类名来访问它，同时类所有的实例都共享静态变量，所有实例的引用都指向同一个地方，任何一个实例对其的修改都会导致其他实例的变化。但是实例变量则不同，它是伴随着实例的，每创建一个实例就会产生一个实例变量，它与该实例同生共死。****

1. ****static 在内存中存储方式****

****存储在方法区中****

****Java把内存分为栈内存和堆内存，其中栈内存用来存放一些基本类型的变量、数组和对象的引用，堆内存主要存放一些对象实例。在JVM加载一个类的时候，若该类存在static修饰的成员变量和成员方法，则会为这些成员变量和成员方法在固定的位置（方法区：存放class被加载后的类信息、常量、静态变量等）开辟一个固定大小的内存区域（只要这个类被加载，Java虚拟机就能根据类名在运行时数据区的方法区内定找到他们），有了这些“固定”的特性，那么JVM就可以非常方便地访问他们。****