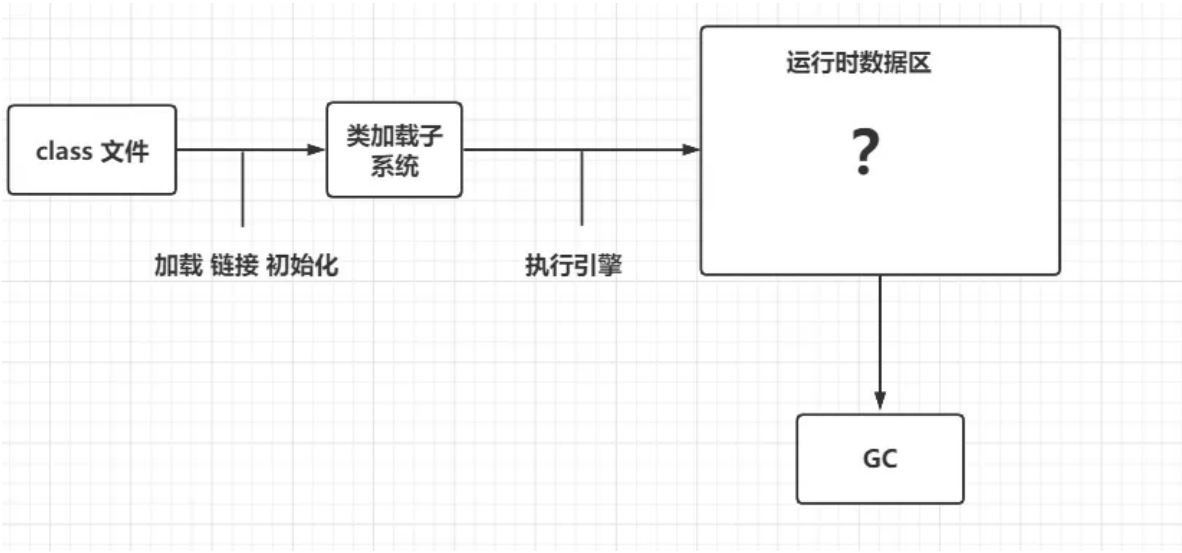
JVM 学习笔记篇

概述

当我们通过前面的：**类的加载-> 验证 -> 准备 -> 解析 -> 初始化** 这几个阶段完成后，就会用到执行引擎对我们的类进行使用，同时执行引擎将会使用到我们运行时数据区，如下：



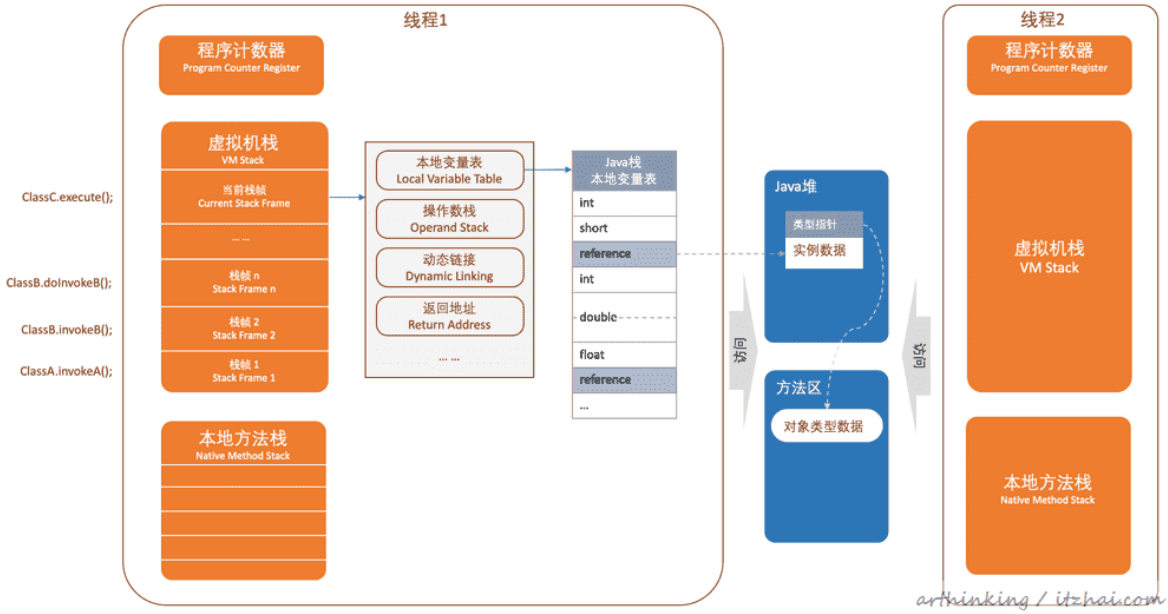
内存是非常重要的系统资源，是硬盘和 CPU 的中间仓库及桥梁，承载着操作系统和应用程序的实时运行 JVM 内存布局规定了 Java 在运行过程中内存申请、分配、管理的策略，保证了 JVM 的高效稳定运行。不同的 JVM 对于内存的划分方式和管理机制存在着部分差异。结合 JVM 虚拟机规范，来探讨一下经典的 JVM 内存布局。

备注：我们通过磁盘或者网络 IO 得到的数据，都需要先加载到内存中，然后 CPU 从内存中获取数据进行读取，也就是说内存充当了 CPU 和磁盘之间的桥梁

第2章：内存区域与内存溢出异常

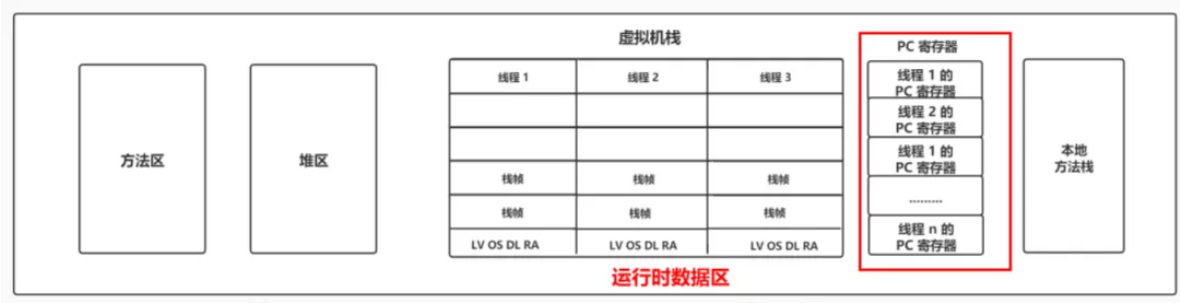
2.1 运行时数据区域

- 程序计数器（私有）
- 虚拟机栈（私有）
- 本地方法栈（私有）
- 方法区
- 堆



2.1.1 程序计数器（PC Register）

JVM 中的`程序计数寄存器`（`Program Counter Register`）中，Register 的命名源于 CPU 的寄存器，寄存器存储指令相关的现场信息。CPU 只有把数据装载到寄存器才能够运行。这里，并非是广义上所指的物理寄存器，或许将其翻译为 PC计数器（或指令计数器）会更加贴切（也称为程序钩子），并且也不容易引起一些不必要的误会。JVM 中的 PC 寄存器是对物理PC寄存器的一种抽象模拟。



程序计数器是一块很小的内存空间，几乎可以忽略不记。也是运行速度最快的存储区域。在 JVM 规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的 Java 方法的 JVM 指令地址；或者，如果是在执行 native 方法，则是未指定值（undefned）。它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。它是唯一一个在 Java 虚拟机规范中没有规定任何 outotMemoryError 情况的区域。

1.当前线程所执行的字节码行号指示器。

2.每个线程都有一个自己的PC计数器。

3.线程私有的，生命周期与线程相同，随JVM启动而生，JVM关闭而死。

4.线程执行Java方法时，记录其正在执行的虚拟机字节码指令地址。

5.线程执行Native方法时，计数器记录为空(Undefined)。

6.唯一在Java虚拟机规范中没有规定任何OutOfMemoryError情况区域。

作用

程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在Java虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，**它是程序控制的指示器，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖这个计数器来完成。**

一句话概括： 用来存储指向下一条指令的地址，也即将要执行的指令代码。由执行引擎读取下一条指令。

问题

（1）使用PC寄存器存储字节码指令地址有什么用呢?

答：因为 CPU 需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行。JVM的字节码解释器就需要通过改变 PC 寄存器 的值来明确下一条应该执行什么样的字节码指令。

（2）PC寄存器为什么被设定为私有的?

答：我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU 会不停地做任务切换，这样必然导致经常中断或恢复，如何保证分毫无差呢？`为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个PC寄存器`，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。由于 CPU 时间片轮限制，众多线程在并发执行过程中，任何一个确定的时刻，一个处理器或者多核处理器中的一个内核，只会执行某个线程中的一条指令。这样必然导致经常中断或恢复，如何保证分毫无差呢？每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器在各个线程之间互不影响。

（3）何为时间片?

答：CPU 时间片即 CPU 分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片。在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。但在微观上：由于只有一个 CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

2.2.2 Java 虚拟机栈

Java虚拟机栈占有的内存空间也就是我们平常所说的“栈内存”，并且也是线程私有的，生命周期与线程相同。虚拟机栈描述的是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，用于存储局部变量表（基本数据类型，对象的引用和returnAddress类型）、操作数栈、动态链接、方法出口等信息。**

局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

每个方法被调用直至执行完成的过程，就对应着一个栈帧从虚拟机栈中从入栈到出栈的过程。对于Java虚拟机栈，有两种以尝情况：

```
1. 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverFlowError异常。配置栈内存：-Xss
2. 如果虚拟机栈在动态扩展时，无法申请到足够的内存，就会抛出OutOfMemoryError异常。配置堆内存：-Xmx
```

参见《常见的JVM内存溢出异常》

2.2.3 本地方法栈

本地方法栈和虚拟机栈所发挥的作用非常相似，它们之间的区别主要是：虚拟机栈是为虚拟机执行的Java方法（即字节码）服务的，而本地方法栈则为虚拟机使用到的Native方法服务。

与虚拟机栈类似，本地方法栈也会抛出StackOverFlowError和OutOfMemoryError异常。

2.2.4 Java 堆

Java堆是Java虚拟机所管理的内存中最大的一块。Java堆在主内存中，是被所有线程共享的一块内存区域，其随着JVM的创建而创建，堆内存的唯一目的是存放对象实例和数组。同时Java堆也是GC管理的主要区域。

Java堆在物理上不需要连续的内存，只要逻辑上连续即可。如果堆中没有内存完成实例分配，并且也无法再扩展时，将会抛出OutOfMemoryError异常。

2.2.5 方法区

方法区是所有线程共享的一块内存区域。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。方法区也有一个别名叫Non-heap（非堆），用来与Java堆区分。对于HotSpot虚拟机来说，方法区又习惯成为“永久代（Permancent Generation）”，但这只是对于HotSpot虚拟机来说的，其他虚拟机的实现上并没有这个概念。相对而言，垃圾收集行为在这个区域比较少出现，但也并非不会来收集，这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载上。

根据Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError 异常。

2.2.6 运行时常量池

运行时常量池属于方法区。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量表，用于存放编译期生成的各种字面常量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放（JDK1.7开始，常量池已经被移到了堆内存中了）。

也就是说，这部分内容，在编译时只是放入到了常量池信息中，到了加载时，才会放到运行时常量池中去。运行时常量池县归于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区的运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用的比较多的是String类的intern()方

法。

当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常，常量池属于方法区，同样可能抛出OutOfMemoryError异常。

Java内存区域模型总结

内存区域	线程私有	主要作用	溢出异常
程序计数器	是	记录当前线程执行的位置	无异常
虚拟机栈	是	存储局部变量表（基本数据类型，**对象的引用**和returnAddress类型）、操作数栈、动态链接、方法出口等信息（java方法）	StackOverflowError和OutOfMemoryError
本地方法栈	是	和虚拟机栈相似，区别本地方法栈为虚拟机使用到的Native方法服务	StackOverflowError和OutOfMemoryError
堆	否	存放对象实例和数组	OutOfMemoryError
方法区	否	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	OutOfMemoryError

2.2 直接内存（Direct Memory）

直接内存不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域。但是这部分也被频繁地使用，而且也可能导致OutOfMemoryError 异常出现。

直接内存是在Java堆外的、直接向系统申请的内存区间。

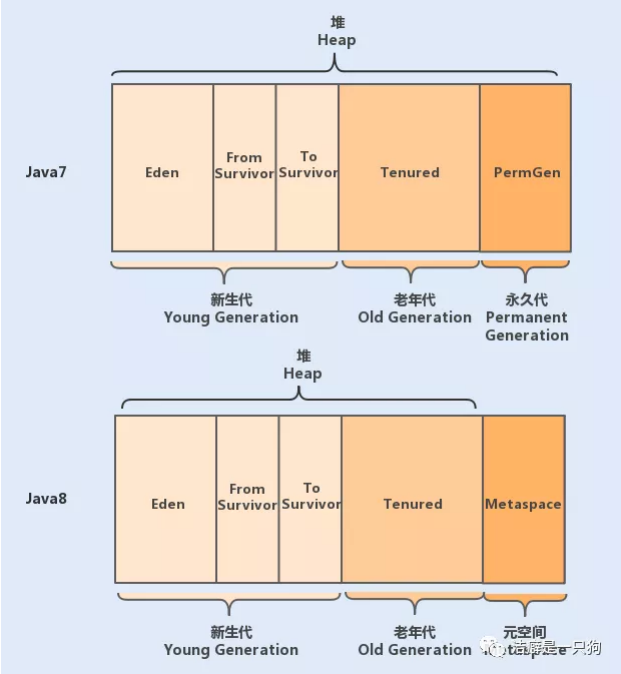
来源于NIO，通过存在堆中的DirectByteBuffer操作Native内存。

通常，访问直接内存的速度会优于Java堆，即读写性能高：

- 因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存。
- Java的NIO库允许Java程序使用直接内存，用于数据缓冲区。

3 JVM 参数

3.1 JVM内存模型



程序计数器

程序计数器是一块很小的内存空间，主要记录了线程的字节码资质，如分支，循环，跳转，异常，线程恢复等会依赖程序计数器

比如，在多线程中，当线程的数量多于CPU的数量的时候,线程之间就会发生CPU抢占,比如一个线程的时间片用完了，或者其他原因此线程的CPU资源被提前抢夺，那么这个线程退出就需要一个程序计数器记录下一次被唤醒的指令,且是唯一一个没有规定任何OutOfMemoryError情况的区域.

方法区

很多开发者都习惯将方法区成为永久代，但是这两者并不等价。

HotSpot虚拟机使用永久代实现方法区，但是在其他虚拟机中，例如Oracle的JRockit,IBM的J9就不存在永久代的说法,可以说，HotSpot虚拟机中，设计人员使用永久代实现JVM内存模型的方法区。

方法区是用来存储加载类的相关信息,包括类信息，运行时常量，字符串常量池，类信息包括类的版本，字段，方法，接口和父类信息

JVM在执行某个类的时候,必须经过加载，连接，初始化，而连接又分为验证，准备，解析三个阶段，在加载类的时候，JVM会先加载class文件，而在class文件中除了有类的版本，字段，方法和接口等描述信息，还有一项信息就是常量池，为常量池存放的是字面量和符号引用

- 字面量就是字符串，基本类型的常量(final修饰的变量)
- 符号应用则包括类和方法的权限定名，字段的名称和描述以及方法的名称和描述符

同时类加载到内存之后，JVM将class文件的常量池放到了运行时常量池，在解析阶段，JVM会把符号引用替换为直接引用

例如，类中的一个字符串常量在class文件中，存放在class文件常量池中，在类加载完之后，JVM把这个字符串常量放到了运行时常量池，而在解析阶段,会指定该字符串对象的索引值,运行时常量是共享的，所以class文件中常量池多个相同的字符串在运行时常量池只有一份.

运行池常量是方法区的一部分,运行时常量相对于class中的常量池有一个另外的特性，就是具备动态性，java语言并不要求常量一定在编译期才会产生，运行期间也可以将新的常量放入池中，比如我们使用String.intern().

方法区和堆一样是线程共享的，因此如两个线程同时访问方法区的同一个类信息，而这个类还没有装入JVM,那么只有一个线程允许加载他，另外一个等待

同时在java7中已经把静态变量和运行池常量放到了堆中，其他部分存储在JVM的非堆内存中,但是在java8版本，使用元空间替代了永久代，除了静态变量和运行时常量还放在堆中，其余在方法区的信息都迁移到了元空间，而元空间是本地内存.

为什么用元空间替换永久代

1. 移除永久代是为了以后可以融合HotSpot JVM与JRockit VM做准备，因为JRockit没有永久代
2. 永久代内存经常溢出,爆出异常java.lang.OutOfMemoryError: PermGen,这是因为java7指定永久代的大小是8M,而每次FULL GC的回收率偏低,不是很好，并且永久代的大小也依赖很多因素，如JVM加载的class总数，常量池的大小和方法的大小.

虚拟机栈

java虚拟机是线程私有的内存空间，他和java线程一起创建，和销毁，当一个线程创建的时候，会在虚拟机栈中申请一个线程栈，存放方法的局部变量，操作数栈,动态链接方法和返回地址等信息，并参与方法的调用和返回，每一个方法的调用伴随着入栈，调出伴随着出栈

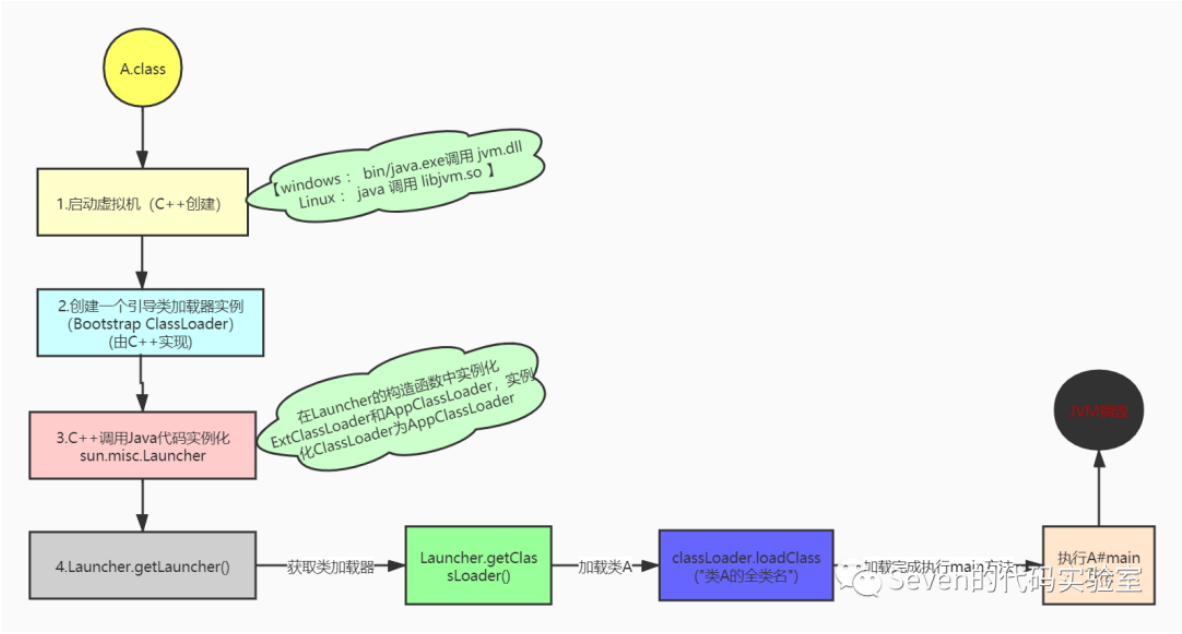
本地方法栈

本地方法栈和虚拟机栈功能类似，但是他是管理本地方法的调用，java虚拟机用于管理java函数的调用，且本地方法栈是有c语言实现，而虚拟机栈是有java实现

3.2 类加载器、沙箱安全机制、双亲委派机制

Java执行代码的大致流程

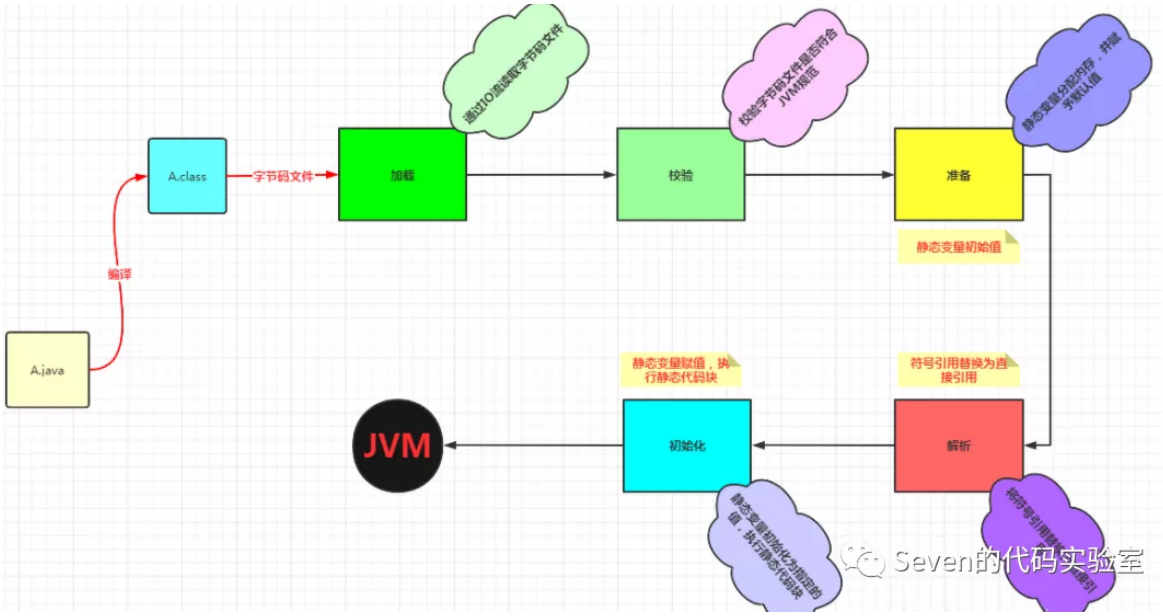
JVM执行Java代码大致有8个步骤:



- 1.启动虚拟机（C++创建）
- 2.创建一个引导类加载器实例（BootstrapClassLoader）
- 3.C++调用Java代码创建JVM启动器，创建sun.misc.Launcher实例（该类由引导类加载器加载创建其他的类加载器）
- 4.sun.misc.Launcher.getLauncher() 获取运行类自己的加载器ClassLoader --> AppClassLoader
- 5.获取到ClassLoader后调用loadClass(“A”)方法加载运行的类A
- 6.加载完成执行A类的main方法
- 7.程序运行结束
- 8.JVM销毁

类加载的步骤

loadClass是类加载中最核心的方法，在后面讲双亲委派的时候会细讲，现在先讲一下类加载中最核心的五个步骤：



- 1.**加载**：通过IO流读取类的字节码文件
- 2.**验证**：校验加载进来的字节码文件是否符合JVM规范
- 3.**准备**：给静态变量分配内存并赋予默认值

- 4.**解析**：将符号引用替换为直接引用
- 5.**初始化**：将静态变量初始化为指定的值，同时执行静态代码块的代码

类加载器和双亲委派机制

类加载器

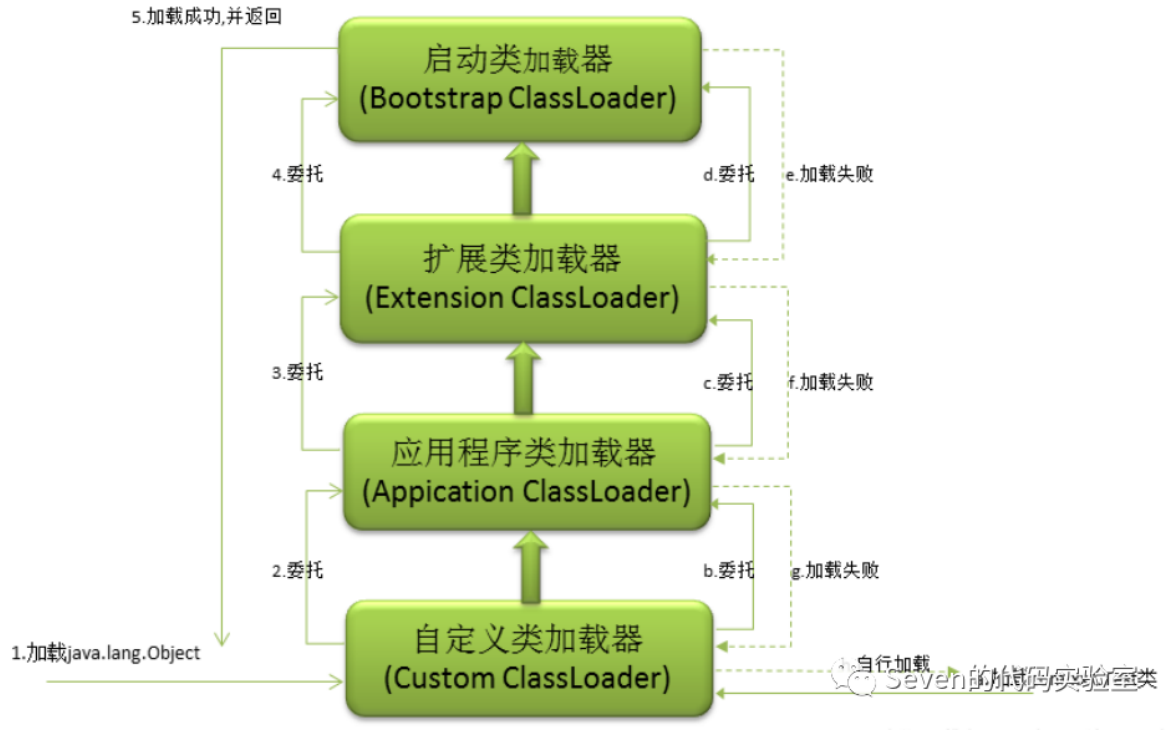
首先看看类加载器有哪几种？

- 1.**BootstrapClassLoader**（引导类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等）

  这个是最顶层的类加载器。

- 2.**ExtClassLoader**（扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包）。
- 3.**AppClassLoader**（应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载我们应用中自己写的那些类）。
- 4.**自定义加载器**：负责加载用户自定义路径下的类包。

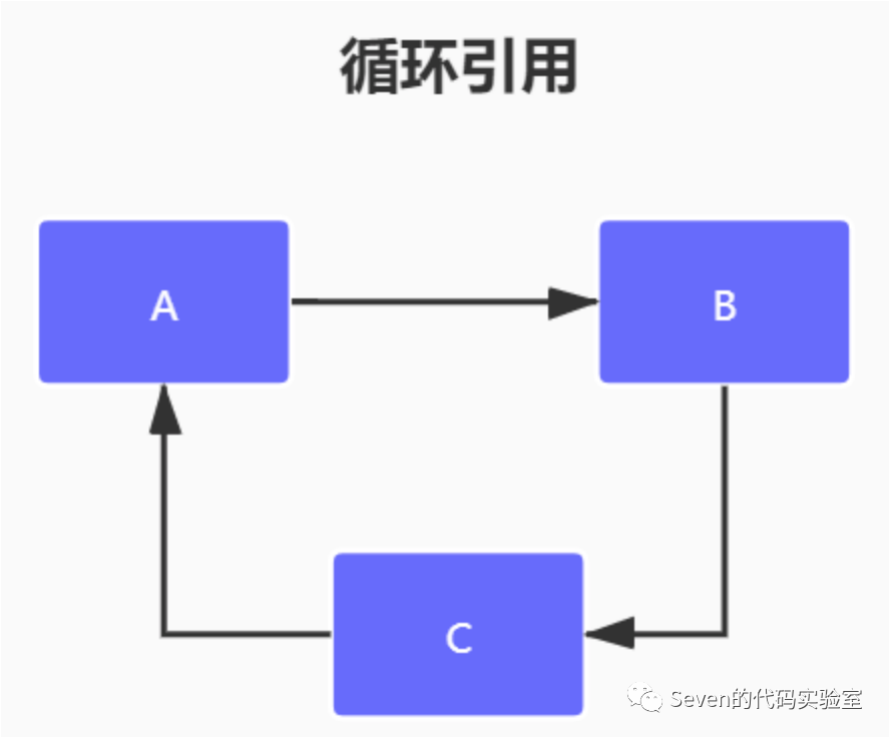
  这个是我们自己定义的类加载器，可以根据实际的需求定制自己类加载器去加载对应的class文件。比如可以定制一个实现热部署的类加载器。



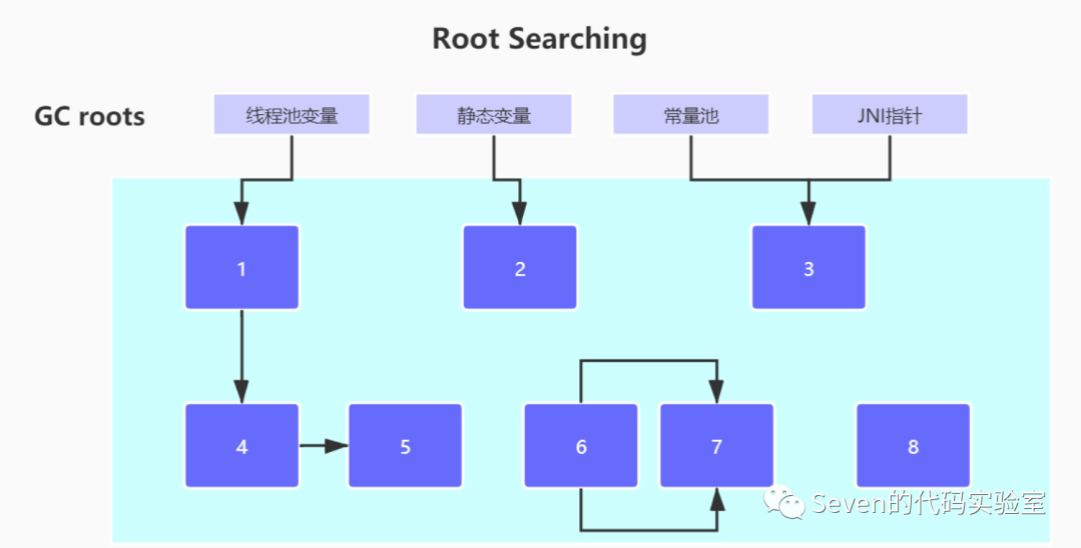
双亲委派定义：当我们需要加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类（注意这里说的父类并不是传统意义上说的继承的父类，他们之间没有继承关系，所以应该说上层类加载器更准确一些）。

3.3 GC垃圾识别算法

- 引用计数法（reference count）— 存在循环引用的问题



- 根可达算法（Root Searching）



根可达算法：是从根上对象开始搜索的算法。

根对象：当一个程序启动的时候需要用到的对象叫根对象。

根可达算法意思就是从根上开始搜，Java程序是从一个main方法开始运行的，一个main方法会开启一个线程，这个线程就会有线程栈，里面就会有main栈帧。从main栈帧里面创建出来的对象都是根对象，当然main方法里面调用了别的方法，那别的方法也是我们引用到的，这些都是有引用的对象，但是从main开始的这个栈帧里的对象都叫做根对象

另外一个是静态变量，一个class它有一个静态的变量，class的被load到内存之后就会对静态变量进行初始化，所以静态变量访问到的对象也是根对象。

还有就是常量池，如果你这个class会用到其他class的那些类的对象，那这些对象也是根对象。

最后是JNI，JNI指的是如果你调用了C和C++写的那些本地方法所用的的对象也是根对象。

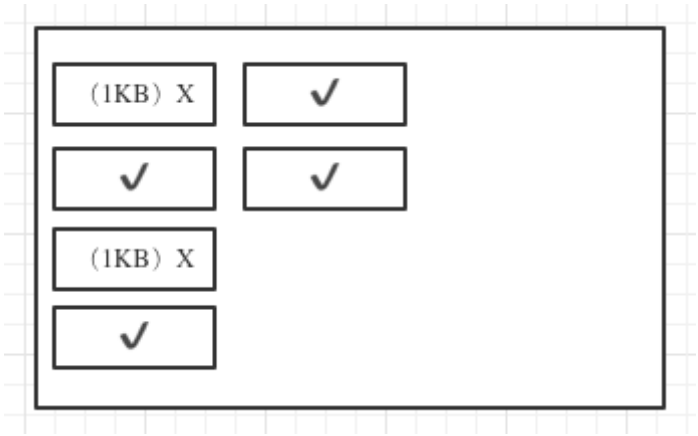
如上图所示，对象一、二、三、四、五均是存在根对象的引用，对象五、六之间是我们上面所提到的循环引用，对象八不存在引用，故对象六、七、八是垃圾。

3.4 哪些是GCRoot

- 线程栈变量：一个main方法开始运行，main线程栈中的变量调用了其他变量，main栈中的方法访问到的对象叫根对象。
- 静态变量：T.class对静态变量初始化时能访问到的对象叫根对象。
- 常量池：如果一个class能够用其他class的对象叫根对象。
- JNI指针：如果调用本地方法运用到的对象叫根对象。
- Java 虚拟机内部的引用，如基本数据类型的Class对象，一些常驻的异常对象，还有系统类加载器。
- 所有被同步锁（synchronized 关键字）持有对象。
- 反映Java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。

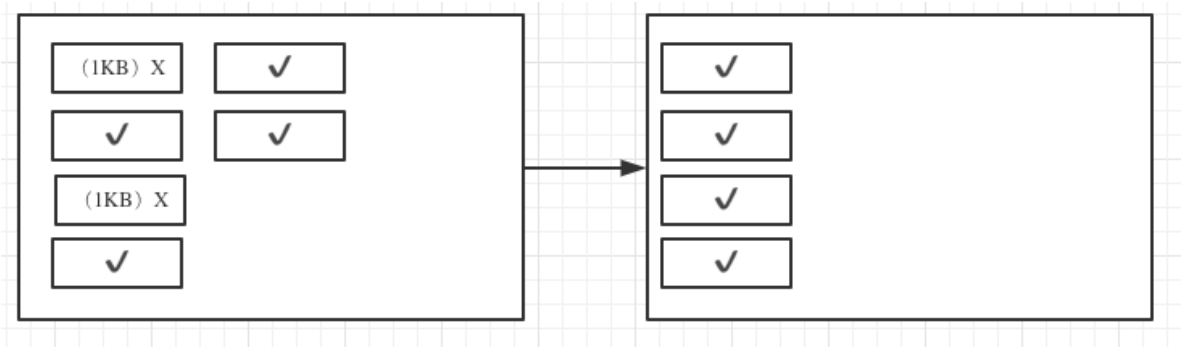
3.5 GC垃圾回收算法

1.标记 - 清除算法； 存在内存碎片的问题，浪费空间



在gc时候，首先扫描时对需要清理的无用对象进行标记，然后将这些对象直接清理。

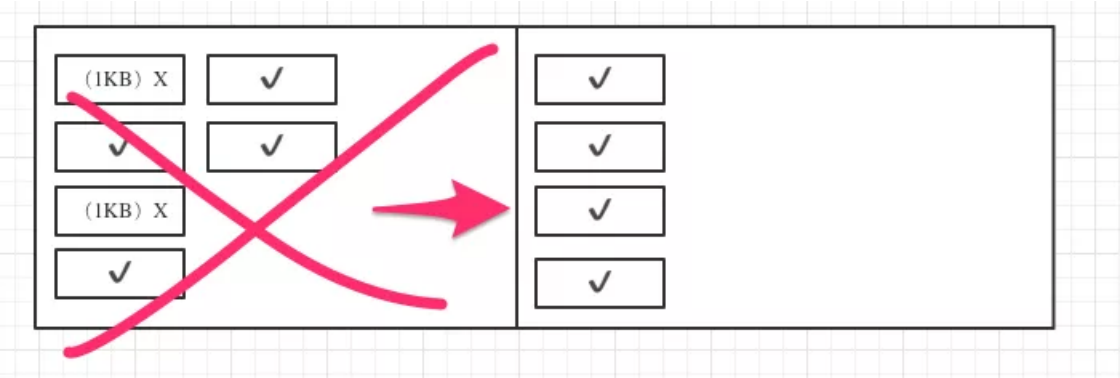
2.标记 - 整理算法



标记 - 整理算法就是在标记 - 清理算法的基础上，多加了一步整理的过程，把空闲的空间进行上移，从而解决了内存碎片的问题。

但是缺点也很明显：每进一次垃圾清除都要频繁地移动存活的对象，效率十分低下。

3. 标记 - 复制算法

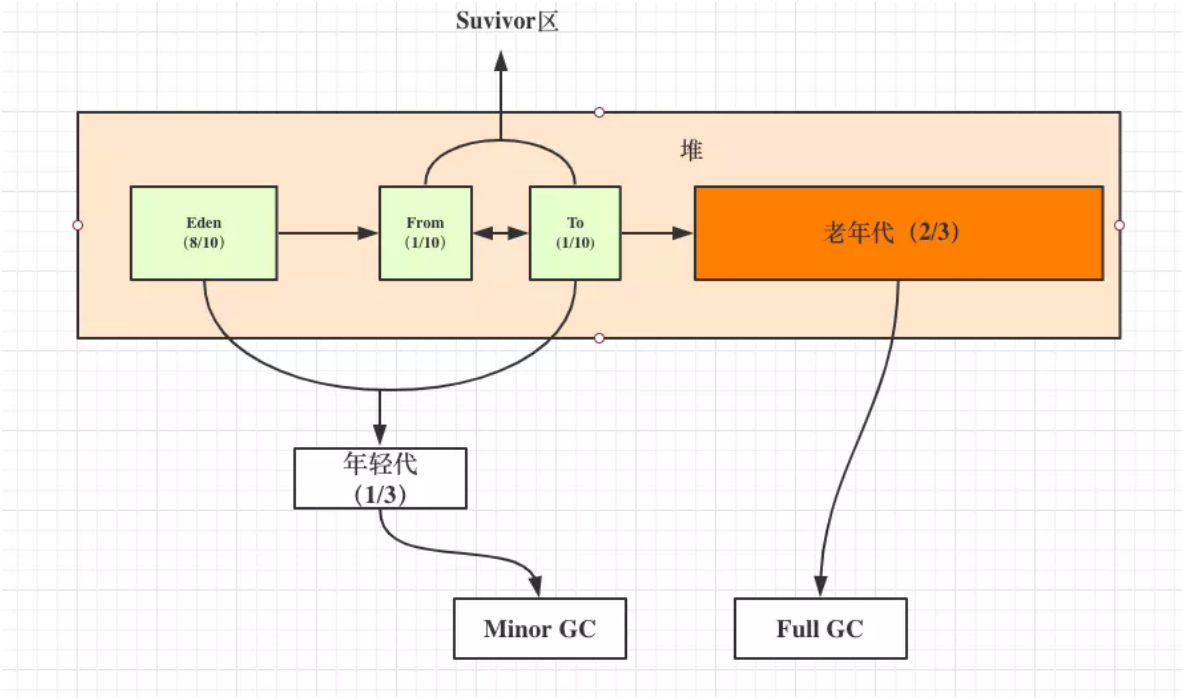


复制算法是将空间一分为二，在清理时，将需要保留的对象复制到第二块区域上，复制的时候直接紧凑排列，然后把原来的一块区域清空。

不过复制算法的缺点也显而易见，本来 JVM 堆假设有 100M 内存，结果由于将空间一分为二，真正能用的变成只有 50M 了！这肯定是不能接受的！另外每次回收也要把存活对象移动到另一半，效率低下。

4. 分代算法

分代收集算法整合了以上算法，综合了这些算法的优点，最大程度避免了它们的缺点。与其说它是算法，倒不是说它是一种策略，因为它是把上述几种算法整合在了一起，我们先从下图看看对象的生存规律。



新生代和老年代的默认比例为 1 : 2，新生代又分为 Eden 区， from Survivor 区（简称 S0 ），to Survivor 区(简称 S1)，三者的比例为 8: 1 : 1。

根据新老生代的特点选择最合适的垃圾回收算法，我们把新生代发生的 GC 称为 Young GC（也叫 Minor GC ）,老年代发生的 GC 称为 Old GC（也称为 Full GC ）。

大多数情况下，对象在新生代 Eden区中分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 MinorGC；

Minor GC 非常频繁，一般回收速度也比较快；出现了 Full GC，经常会伴随至少一次的 Minor GC，Full GC 的速度一般会比 Minor GC 慢10倍以上。

3.6 JVM调节参数

3.7 各种垃圾回收器的原理，CMS，G1（Garbage First GC）

指标:

- 内存占用（Footprint）；
- 吞吐量（Throughput）；
- 延迟（Latency）；

串行垃圾回收器：Serial、Serial Old。

并行垃圾回收器：ParNew、Parallel Scavenge（简称：Parallel）、Parallel Old。

并发垃圾回收器：CMS、G1。

G1 是一个并行回收器，它把堆内存分割为很多不相关的区域（Region）物理上不连续，使用不同的 Region 来表示 Eden、Survior 0、Servior 1、老年代。G1 跟踪各个 Region 里面的垃圾堆积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

G1 使用了全新的分区算法，有如下特点

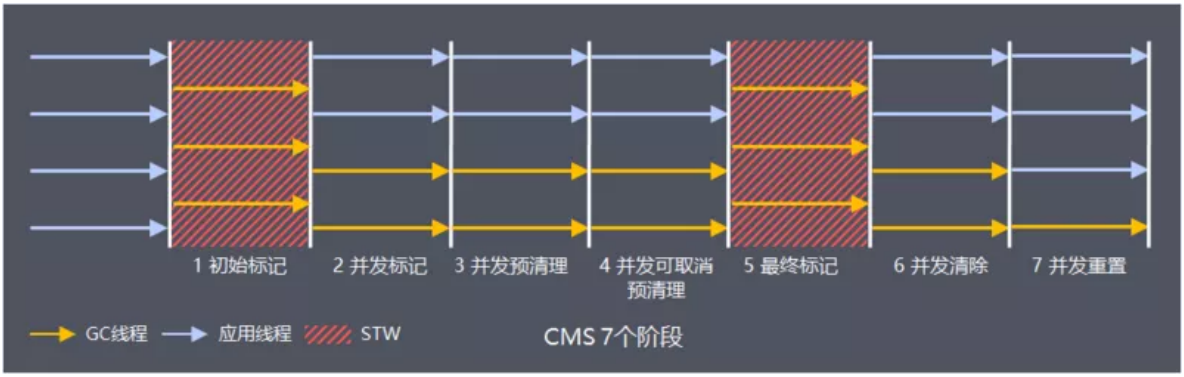
- 并行与并发
- - 并行性，在回收期间，可以有多个 GC 线程同时工作，此时用户线程 STW。
  - 并发性，G1 部分工作可以和应用出现同时执行。
- 分代收集
- - G1 仍然属于分代型垃圾回收器，它还是会区分年轻代、老年代，但从堆的结构上看，它不要求整个代区连续，也不再坚持固定大小和固定数量。
  - 将堆空间分为若干个区域，这些区域中包含逻辑上的年轻代和老年代。它同时兼顾年轻代和老年代的回收。
- 空间整合
- - G1 内存回收是以 Region 作为基本单位的，Region 之间是复制算法。避免内存碎片，有利于程序的长久运行。

    ```
    -XX:+UseG1GC、-XX:G1HeapRegionSize
    ```

CMS

CMS(Concurrent Mark and Swee 并发-标记-清除)，是一款基于并发、使用标记清除算法的垃圾回收算法，只针对老年代进行垃圾回收。CMS收集器工作时，尽可能让GC线程和用户线程并发执行，以达到降低STW时间的目的。

```
-XX:+UseConcMarkSweepGC
```



jdk1.8 默认：UseParallelGC

jdk1.9默认：UseG1GC

3.8 G1（Garbage First GC）的调节参数

G1 GC是启发式算法，会动态调整年轻代的空间大小。目标也就是为了达到接近预期的暂停时间。G1提供了两种GC模式，Young GC和Mixed GC，两种都是Stop The World(STW)的。

Young GC

Young GC主要是对Eden区进行GC，它在Eden空间耗尽时会被触发。在这种情况下，Eden空间的数据移动到Survivor空间中，如果Survivor空间不够，Eden空间的部分数据会直接晋升到老年代空间。Survivor区的数据移动到新的Survivor区中，也有部分数据晋升到老年代空间中。最终Eden空间的数据为空，GC停止工作，应用线程继续执行。

Mixed GC

Mix GC不仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的老年代分区。GC步骤分2步：全局并发标记（global concurrent marking）和 拷贝存活对象（evacuation）。