JVM

Java代码是怎么运行起来的

首先我们需要用javac命令对Java文件进行编译得到class字节码文件，这些字节码会交给JVM来负责运行。

JVM采用类加载器把编译好的class字节码文件加载到JVM中，JVM会给予自己的字节码执行引擎来执行加载到内存里的我们写好的那些类。

类的加载过程

加载→验证→准备→解析→初始化→使用→卸载

当使用类实例化对象的时候，必须把这个类的字节码文件中的类**加载**到内存里。

在**验证**阶段，根据Java虚拟机规范来校验字节码文件是否符合指定规范。

准备阶段给这个类分配一定内存空间，来一个默认的初始值。

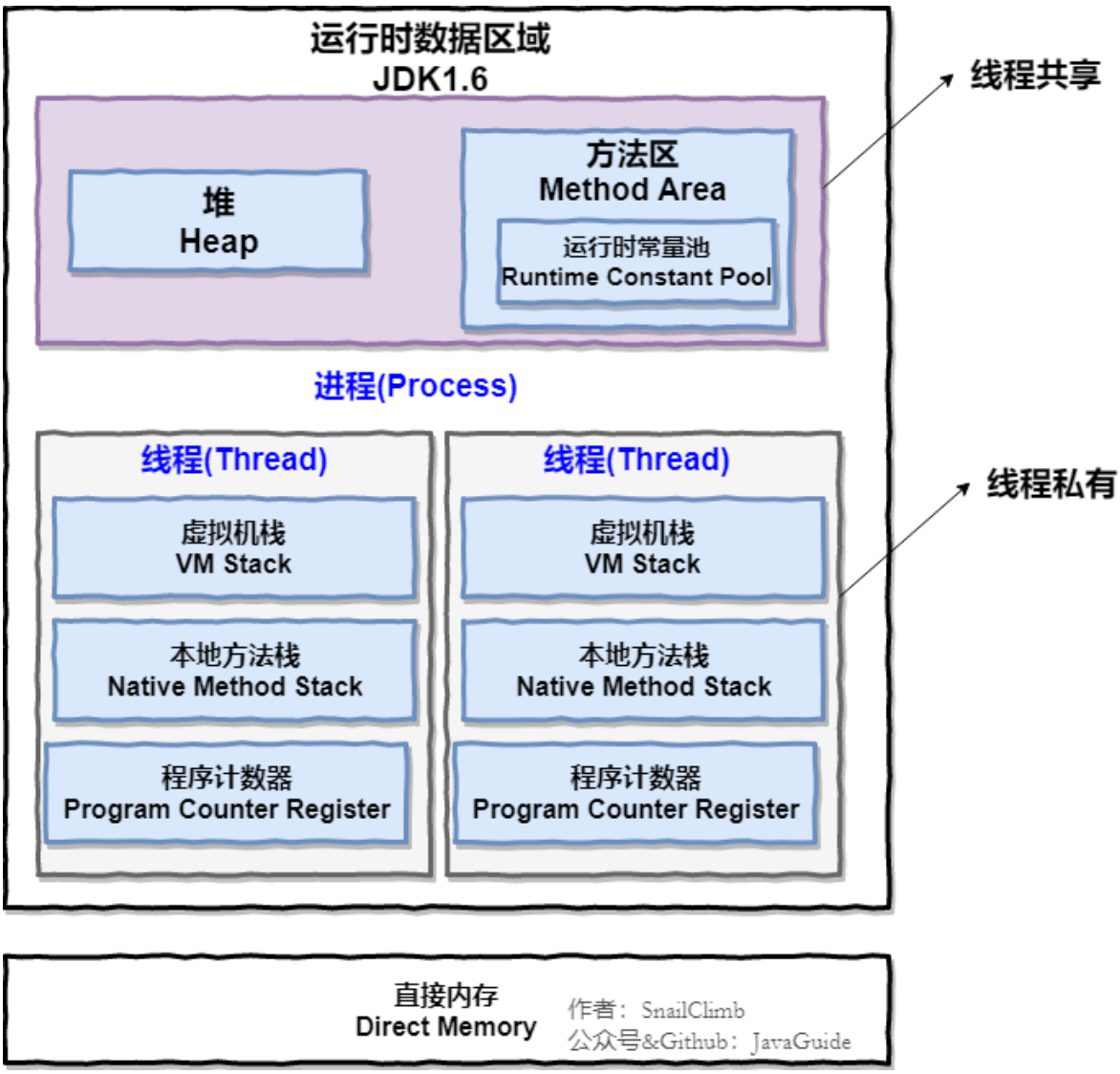
解析阶段做的事情是把符号引用替换为直接引用。

初始化阶段，给类的成员变量赋值

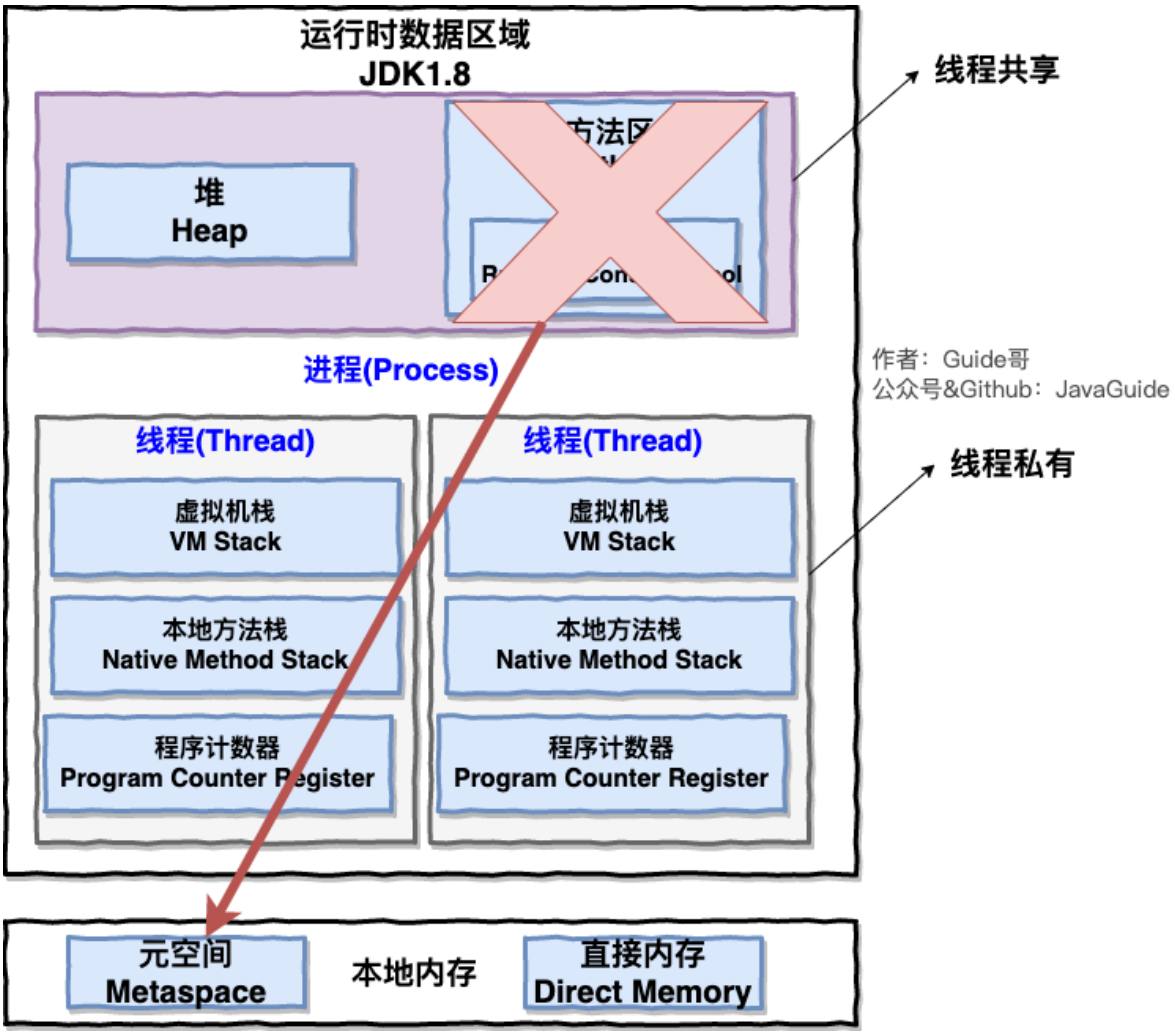
完成以上步骤这个类初始化可以用了

另外类的加载机制遵循 ① **全盘委托**，一个类的和他依赖和调用的类都由一个加载器加载 ②**双亲委托**，加载一个类时总是由父类先去加载，找不到再由子类加载。③缓存机制，所有加载过的类都会被缓存起来，加载一个类时先到缓存里找，如果没有再去加载class文件。

JVM的内存划分


运行时数据区域
JDK1.6
堆
Heap
方法区
Method Area
运行时常量池
Runtime Constant Pool
线程共享
进程(Process)
线程(Thread)
虚拟机栈
VM Stack
本地方法栈
Native Method Stack
程序计数器
Program Counter Register
线程(Thread)
虚拟机栈
VM Stack
本地方法栈
Native Method Stack
程序计数器
Program Counter Register
线程私有
直接内存
Direct Memory
作者：SnailClimb
公众号&Github：JavaGuide

在JDK1.8之前，同一个进程中的多个线程，共享这个进程的堆和方法区，运行时常量池在方法区里面。每个线程都有各自的虚拟机栈、本地方法栈和程序计数器。



到了JDK1.8，方法区被彻底移除了，取而代之的是元空间，元空间使用的是直接内存。

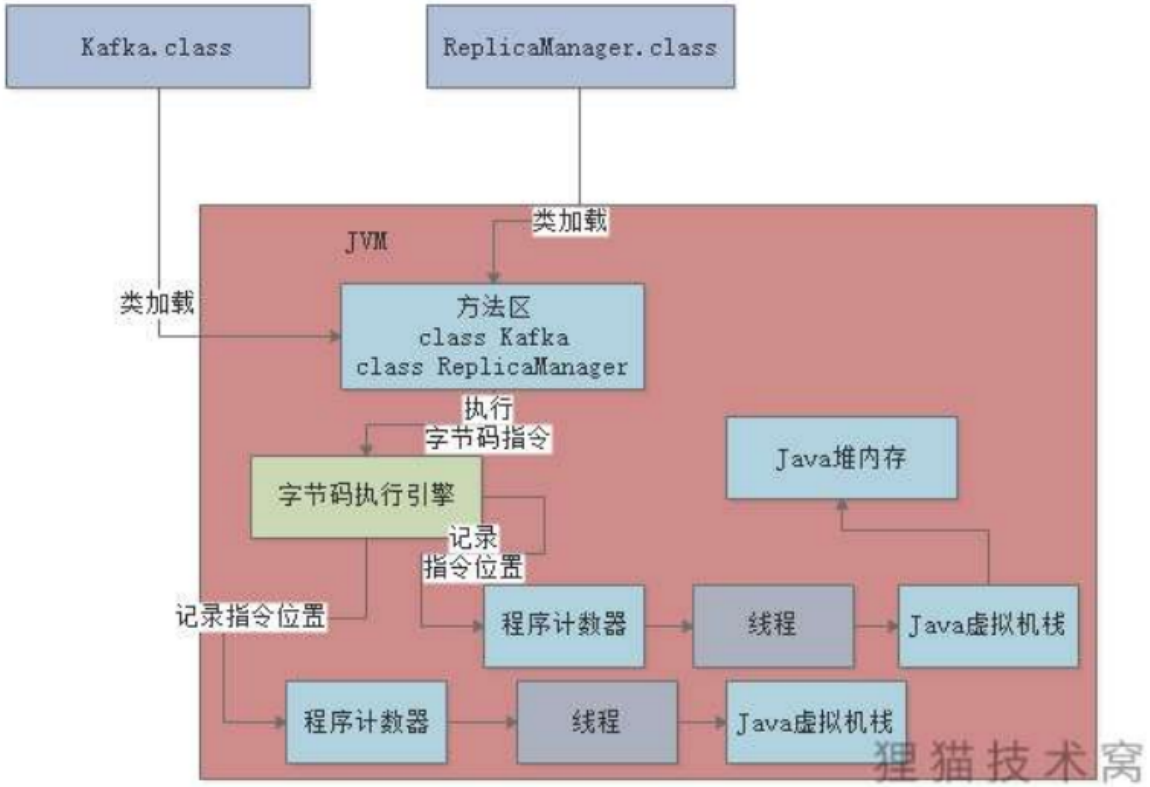
方法区存放我们自己写的各种类的信息

程序计数器是用来记录当前执行的字节码指令的位置，也就是记录目前执行到了哪一条字节码指令

虚拟机栈用来保存每个方法内的局部变量等数据。调用执行任何方法时，都会给方法创建栈帧然后入栈

堆内存存放代码中创建的各种对象

当使用native方法调用本地操作系统的一些方法，就会有线程对应的**本地方法栈**，里面存放了各种native方法的局部变量表之类的信息



串联起来

当一个类被JVM加载到内存后，会存放在方法区。

字节码文件对应着各种字节码指令，JVM会使用自己的字节码执行引擎去执行代码编译出来的代码指令。

JVM是支持多线程的，因此写好的代码会开启多个线程并发执行不同的代码指令，所以每个线程都会有自己一个程序计数器，专门记录当前这个线程目前执行到了字节码指令的位置。

当线程执行一个方法的时候，就会在自己的Java虚拟机栈用来存放自己执行的方法的局部变量，对这个方法调用创建对应的栈帧。

代码运行过程中创建的各种对象，都是放在堆内存中的

大家脑子里一定要有一个会动的图，你的代码在运行的时候，起码有一个main线程会去执行所有的代码，当然也可能是你启动的别的线程。

然后线程执行时必须通过自己的程序计数器来记录执行到哪一个代码指令了

另外线程在执行方法时，为每个方法都得创建一个栈帧放入自己的Java虚拟机栈里去，里面有方法的局部变量。

最后就是代码运行过程中创建的各种对象，都是放在Java堆内存里的。

JVM分代模型：年轻代和老年代

因为根据用户写代码方式的不同，采用不同的方式来创建和使用对象，对象的生存周期是不同的。

因此JVM将Java堆内存划分为了两个区域，一个是年轻代，一个是老年代

年轻代存放的对象时创建和使用完之后立马就要回收的对象

老年代存放的对象时创建之后需要一直长期存在的对象

划分为年轻代和老年代也是为了方便针对性使用不同的垃圾回收算法

还有一个永久代其实就是方法区，存放一些类的信息

方法区的垃圾回收

以下几种情况下，方法区里的类会被回收：

- 首先该类的所有实例对象都已经从Java堆内存被回收
- 其次加载这个类的ClassLoader已经被回收
- 最后对该类的Class对象没有任何引用

对象在JVM内存中如何分配？如何流转的？

大部分正常的对象都是优先在新生代分配内存的。

当新生代预先分配的内存空间几乎被占满了无法分配新对象了，就会触发一次新生代内存空间的垃圾回收。也称之为"Minor GC"或者"Young GC"，把新生代里那些没有人引用的垃圾对象都给回收掉

当一个对象在多次垃圾回收之后依然存活，就会被转移到堆内存的老年代中去。

对象什么情况下进入老年代

在默认设置下，当对象年龄达到了15岁，也就是躲过了15次GC的时候，它就会转移到老年代里去。

还有另一个动态对象年龄判断规则，加入当前放对象的Survivor区域中，一批对象的总大小大于Survivor区域内存的五成，那么此时大于等于这批对象年龄的对象都可以直接进入老年代。

我们可以设置参数，当要创建一个大于这个大小的对象，就直接把这个大对象放到老年代里去。之所以这么做就是为了避免新生代出现屡次躲过多次GC的大对象，要把它在两个Survivor区域来回复制多次才能进入老年代，这样会非常耗时间。

如果新生代在Minor GC之后剩余存活对象太多了，没法放入另外一块Survivor，这时就必须得把这些对象直接转移到老年代去。

老年代对于这个策略是有**空间分配担保规则**的：

第一种可能，Minor GC过后，剩余的存活对象大小小于Survivor区大小，直接放入Survivor区即可

第二种可能，Minor GC过后，剩余存活对象大小大于Survivor区大小，小于老年代可用内存大小，就直接进入老年代。

第三种可能，Minor GC过后，剩余存活对象大小同时大于Survivor区大小和老年代可用内存大小。此时老年代放不下这些存活对象，就会触发一次Full GC

要是Full GC过后，老年代还是没有足够的空间存放Minor GC过后的剩余存活对象，那么此时就会导致OOM内存溢出。

什么情况下JVM内存中的一个对象会被垃圾回收

JVM中使用可达性分析算法来判定对象是否可以被回收，对每个对象都会分析有谁在引用他，一层层往上判断，看是否有GC Roots。

在JVM规范中，方法的**局部变量和类的静态变量都是GC Roots**，被他们引用的对象都不会被回收。

跟JVM内存相关的集合核心参数

-Xms:Java堆内存大小

-Xmx:Java堆内存最大大小

-Xmn:Java堆内存中新生代大小，扣除新生代剩下就是老年代内存大小

-XX:PermSize:永久代大小

-XX:MaxPermSize:永久代最大大小

-Xss:每个线程的栈内存大小

Java对象不同的引用类型：强引用、弱引用、软引用和虚引用

强引用的类型，垃圾回收的时候绝不会去回收这个对象

软引用在正常情况下不会被垃圾回收，自由当内存不足了，才会被回收

弱引用无论内存够不够都会被垃圾回收

JVM中有哪些垃圾回收算法，每个算法各自的优劣是什么?

标记-清除算法

算法分为"标记"和"清除"两个阶段：首先标记出所有需要回收的对象，在标记完成之后统一回收掉所有被标记的对象。标记-清除算法的缺点有两个：效率问题，标记和清除效率都不高。其次，标记清除之后会产生**大量的不连续的内存碎片**，空间碎片太多会导致当程序需要**为较大对象分配内存时无法找到足够的连续内存**而不得不提前触发另一次垃圾收集动作。

复制算法

将可用内存按容量分成大小相等的两块，每次只使用其中一块，当这块内存使用完了，就将还存活的对象复制到另一块内存上去，然后把使用过的内存空间一次清理掉。这样只需要移动堆顶指针，按顺序分配内存即可。复制算法的缺点显而易见，可使用的内存降为原来一半。

标记-整理算法

标记-整理算法在标记-清除算法基础上做了改进，标记阶段是相同的标记出所有需要回收的对象，让所有存活的对象都向一端移动，在移动过程中清理掉可回收的对象，这个过程叫做整理。复制算法在对象存活率高的情况下就要执行较多的复制操作，效率将会变低，而在**对象存活率高的情况下**使用标记-整理算法效率会大大提高。

分代收集算法

根据内存中对象的存活周期不同，将内存划分为几块，java的虚拟机中一般把内存划分为新生代和年老代，当新创建对象时一般在新生代中分配内存空间，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创建。

新生代如何进行垃圾回收

新生代使用复制算法进行垃圾回收，但是并不是简单的将内存划分为两半

真正的复制算法会做出如下优化，把新生代内存空间划分为三块：1个Eden区，2个Survivor区。前者占八成，后者各占一成。

复制算法是这样子运行的：

刚开始对象都是分配在Eden区，如果Eden区快满了，就会触发垃圾回收。此时就会把Eden区中存活对象一次性转移到空着的Survivor区。接着Eden区就会被清空，然后再次分配新对象到Eden区里。如果下次Eden区满，就会再次触发Minor GC，把Eden区和放着上次Minor GC存活对象的Survivor区内的存活对象，转移到另一块Survivor区去。接着新对象继续分配在Eden区和另外那块开始被使用的Survivor区，然后始终保持一块Survivor区是空着的，就这样一直循环使用这三块内存空间。

这么做的最大好处是，只有一成的内存空间是被闲置的，九成的内存都被使用上了。

年轻代和老年代分别适合什么样的垃圾回收算法?

年轻代使用的是复制算法

老年代使用的是标记整理算法，首先标记出老年代当前存活的对象，接着让这些存活对象在内存里移动，把存活对象尽量都挪到一边去，让存活对象紧凑靠在一起，避免垃圾回收过后出现过多的内存碎片。再一次性把垃圾对象都回收掉。

老年代的垃圾回收算法速度至少比新生代的慢十倍，如果系统频繁出现老年代的Full GC，会导致系统性能被严重影响，出现频繁卡顿的情况。

触发新生代GC的时机

当新生代的Eden区满了的时候，就会触发年轻代GC。采用复制算法来回收新生代垃圾。

触发老年代GC的时机

1. 老年代可用内存小于新生代全部对象大小，如果没有开启空间担保参数，会直接触发Full GC，所以一般空间担保参数都会打开
2. 老年代可用内存小于历次新生代GC后进入老年代的平均对象大小，此时会提前Full GC。
3. 新生代Minor GC后的存活对象大于Survivor，并且老年代内存不足，会Full GC。

JVM中都有哪些常见的垃圾回收器，各自的特点是什么?

Serial和Serial Old垃圾回收器，分别用来回收新生代和老年代的垃圾对象。

工作原理是单线程运行，垃圾回收的时候会停止我们自己写系统的其他工作线程。

ParNew和CMS垃圾回收器，分别用来回收新生代和老年代的垃圾对象。

他们都是多线程并发机制，性能更好，现在一般是线上生产系统的标配组合。

G1垃圾回收器统一收集新生代和老年代。

CMS收集器流程

CMS在执行一次垃圾回收的过程一共分为4个阶段：

- 初始标记
  - 标记出所有GC Roots直接引用的对象，需要暂停一切工作线程，速度很快
- 并发标记
  - 系统线程可以随意创建各种新对象或者让部分存活对象失去引用。
  - 垃圾回收线程尽可能对已有的对象进行GC Roots追踪
  - 花费时间最长，但是并发不影响系统
- 重新标记
  - 让系统程序停下来，重新标记第二阶段新创建的一些对象或者是去引用变成垃圾的对象，速度很快
- 并发清理
  - 让系统程序随意运行，清理之前标记为垃圾的对象

CMS垃圾回收期间的一些细节问题

1. 并发回收垃圾导致CPU资源紧张

并发标记需要对GC Roots进行深入追踪，但是老年代里存活对象很多。这个过程会追踪大量对象，耗时较高。

并发清理需要把垃圾对象从各种随机的内存位置清理掉，也比较耗时

2. Concurrent Mode Failure问题

并发清理阶段，CMS回收之前标记好的垃圾对象。但是这个阶段系统一直在运行，可能会随着系统运行让一些对象进入老年代，同时还变成垃圾对象，这种垃圾对象是“浮动垃圾”。

因为它虽然成为了垃圾，但是CMS只能回收之前标记出来的垃圾对象，不会回收它们，需要等到下一次GC的时候才会回收它们。

所以为了保证在CMS垃圾回收期间还有一定的内存空间让一些对象可以进入老年代，一般会预留一些空间。”-XX:CMSInitiatingOccupancyFaction"参数可以设置老年代占用多少比例的时候触发CMS垃圾回收，JKD1.6默认的值是92%。

当CMS垃圾回收期间，系统程序要放入老年代的对象大于可用内存，这时候会发生Concurrent Mode Failure，也就是说并发垃圾回收失败了。此时会自动用“Serial Old”垃圾回收器代替CMS，也就是直接强行把系统程序STW，重新进行长时间的GC Roots追踪，标记出全部垃圾对象，不允许新的对象产生。然后一次性把垃圾对象都回收掉，结束了再恢复系统线程。

所以在生产实践中，自动触发CMS垃圾回收的比例需要合理优化，避免Concurrent Mode Failure问题。

3. 内存碎片问题

CMS采用“标记-清理”算法会导致大量内存碎片产生。如果内存碎片太多会导致后续对象进入老年代找不到可用的连续内存空间，并触发Full GC。

CMS有一个参数是"-XX:+UseCMSCompactAtFullCollection"默认打开的。意思是Full GC之后要再次进行STW，停止工作线程，然后进行碎片整理。就是把存活对象挪到一起。

G1垃圾回收器的工作原理

G1垃圾回收器可以同时回收新生代和老年代对象，最大的特点是把Java堆内存拆分为多个大小相等的Region

G1最大的一个特点是可以让用户设置一个垃圾回收的预期停顿时间。

G1可以做到对垃圾回收导致的系统停顿可控：G1必须要追踪每个Region里的回收价值。也就是说必须搞清楚每个Region里的对象有多少是垃圾，如果对这个Region进行垃圾回收，需要耗费多长时间，可以回收多少垃圾。

G1会追踪每个Region中可以回收的对象大小和预估时间，在垃圾回收的时候，尽量把垃圾回收对系统造成的影响控制在指定的时间范围内，同时在有限的时间回收尽可能多的垃圾对象。

回收过程：

- 初始标记
  - 需要停止其他线程，仅仅标记一下GC Roots直接能引用的对象，速度很快
- 并发标记
  - 允许系统程序运行，同时GC Roots开始追踪所有存活的对象
  - JVM会对并发标记阶段对对象做出的一些修改记录起来
- 最终标记
  - 系统程序停止运行，根据并发阶段记录那些对象修改，最终标记哪些对象时存活和垃圾的。
- 混合回收
  - 计算老年代中每个Region中的存活对象数量，存活对象的占比还有执行垃圾回收的预期性能和效率
  - 接着停止系统程序，全力以赴尽快进行垃圾回收。会选择部分Region进行回收，因为必须让垃圾回收的停顿时间控制在我们指定的范围内。

当进行混合回收的时候，无论年轻代还是老年代都基于复制算法进行回收，都要把各个Region的存活对象拷贝到别的Region里去。如果拷贝的过程中发现没有空闲的Region可以承载自己存活对象了，就会触发一次失败。一旦失败，立马就回切换为停止系统程序，然后采用单线程进行标记、清理和压缩整理，空闲出一批Region，这个过程是非常慢的。

JVM性能优化在优化什么?

系统真正最大的问题是内存分配、参数设置不合理，导致对象频繁进入老年代，频繁触发老年代GC，导致系统频繁的每隔几分钟就要卡死几秒钟

什么是Young GC和Full GC?

1. Minor GC/Young GC

两个名词是等价的。在年轻代的Eden内存区域被占满之后，就需要触发年轻代GC。

Minor GC和Young GC都是针对年轻代的GC。

2. Full GC和Old GC

老年代GC为了避免歧义，用Old GC来指代

3. Full GC

对JVM进行一次整体的垃圾回收，把各个内存区域的垃圾都回收掉

4. Major GC

现在用的很少，要问清楚是说Old GC还是Full GC。

5. Mixed GC

G1中特有的概念。在G1中，一旦老年代占据堆内存的45%，就要触发Mixed GC，对年轻代和老年代都会进行回收。

什么是内存溢出？在那些区域会发生内存溢出？

元空间用来存放类信息，会发生OOM

每个线程的虚拟机栈内存是固定的，也会溢出

在堆内存上不断创建对象，也会发生内存溢出

内存溢出

元空间溢出

一旦元空间满了，就会触发Full GC，连带着回收元空间里的类。但是回收条件非常苛刻，所以未必能回收掉里面很多的类。当JVM还在不断加载类放到元空间后，就会发生内存溢出。

但是一般情况下元空间不会发生内存溢出，发生内存溢出一般都是因为两个原因

第一种原因，工程师不懂JVM直接使用默认参数。默认的才几十MB，对于一些稍微大型的系统，不仅自身有很多类，还依赖了很多外部的jar包有很多类，就很容易超出内存。

第二种原因，很多人写系统用cglib之类的技术动态生成一些类，如果没有控制好导致生成的类过多，就会把元空间塞满导致内存溢出。

栈溢出

线程调用一个方法，都会将本次方法调用的栈压入虚拟机栈，这个栈帧是有方法的局部变量的

但是一般栈内存都是足够进行一定深度的方法调用，除非不断地调用方法压栈，比如方法不断调用自己。

除非代码里写bug，一般不会发生栈溢出

堆溢出

有限的内存中放了过多的对象，而且大多数都是存活的，此时即使GC过后还是大部分都存活。所以要继续放入更多的对象已经不可能了，此时只能引发内存溢出问题。

如何解决内存溢出

元空间内存溢出

先分析GC日志，然后再让JVM自动dump出内存快照，最后用MAT分析一下这份内存快照，从内存快照里找到内存溢出的原因

栈空间内存溢出

把所有的异常都写入本地日志文件，当系统崩溃了，只要去日志定位一下异常信息就知道了

堆空间内存溢出

检查日志文件，拿到JVM导出的内存快照，用MAT分析内存快照即可