一、为什么学虚拟机

学习 Java 虚拟机能深入地理解 Java 这门语言

学习虚拟机是为线上排查问题打下基础

# Java语言的前世今生

JDK 与 JRE

JDK 与 J2SE

J2SE 与 Java SE

Java 语言的历史

# Java虚拟机的历史

# 到底什么是Java虚拟机

# 从源代码到机器码

在 JVM 中有三个非常重要的编译器，它们分别是：前端编译器、JIT 编译器、AOT 编译器。

前端编译器，最常见的就是我们的 javac 编译器，其将 Java 源代码编译为 Java 字节码文件。JIT 即时编译器，最常见的是 HotSpot 虚拟机中的 Client Compiler 和 Server Compiler，其将 Java 字节码编译为本地机器代码。而 AOT 编译器则能将源代码直接编译为本地机器码。这三种编译器的编译速度和编译质量如下：

编译速度上，解释执行 > AOT 编译器 > JIT 编译器。

编译质量上，JIT 编译器 > AOT 编译器 > 解释执行。

而在 JVM 中，通过这几种不同方式的配合，使得 JVM 的编译质量和运行速度达到最优的状态。

# 六、字节码与文件结构

# IMG_256

# Java虚拟机内存结构

Java 虚拟机的内存结构是学习虚拟机所必须掌握的地方，其中以 Java 堆的内存模型最为重要，因为线上问题很多时候都是 Java 堆出现问题。因此掌握 Java 堆的划分以及常用参数的调整最为关键。

除了上述所说的六大部分之外，其实在 Java 中还有直接内存、栈帧等数据结构。但因为直接内存、栈帧的使用场景还比较少，所以这里并不做介绍，以免让初学者一时间混淆。

学到这里，一个 Java 文件就加载到内存中了，并且 Java 类信息就会存储在我们的方法区中。如果创建对象，那么对象数据就会存放在 Java 堆中。如果调用方法，就会用到 PC 寄存器、Java 虚拟机栈、本地方法栈等结构。

# JVM类加载机制

分析一个类的执行顺序大概可以按照如下步骤：

\*\*确定类变量的初始值。\*\*在类加载的准备阶段，JVM 会为类变量初始化零值，这时候类变量会有一个初始的零值。如果是被 final 修饰的类变量，则直接会被初始成用户想要的值。

\*\*初始化入口方法。\*\*当进入类加载的初始化阶段后，JVM 会寻找整个 main 方法入口，从而初始化 main 方法所在的整个类。当需要对一个类进行初始化时，会首先初始化类构造器（），之后初始化对象构造器（）。

\*\*初始化类构造器。\*\*JVM 会按顺序收集 类变量的赋值语句、静态代码块，最终组成类构造器由 JVM 执行。

\*\*初始化对象构造器。\*\*JVM 会按照收集成员变量的赋值语句、普通代码块，最后收集构造方法，将它们组成对象构造器，最终由 JVM 执行。

如果在初始化 main 方法所在类的时候遇到了其他类的初始化，那么就先加载对应的类，加载完成之后返回。如此反复循环，最终返回 main 方法所在类。

# 垃圾回收机制

现今的 Java 虚拟机判断垃圾对象使用的是：GC Root Tracing 算法。其大概的过程是这样：从 GC Root 出发，所有可达的对象都是存活的对象，而所有不可达的对象都是垃圾。

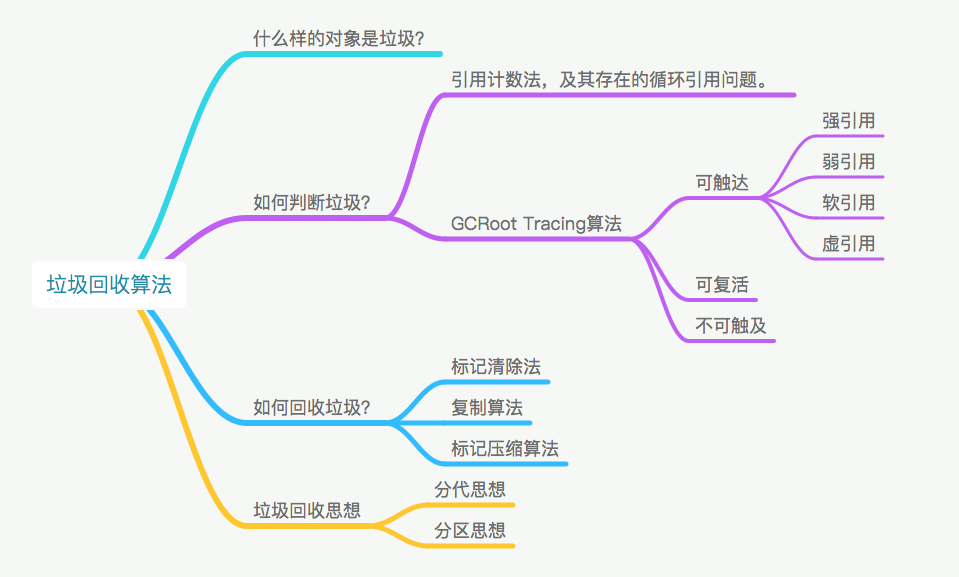
可以看到这里最重要的就是 GC Root 这个集合了，其实 GC Root 就是一组活跃引用的集合。但是这个集合又与一般的对象集合不太一样，这些集合是经过特意筛选出来的，通常包括：

* 所有当前被加载的 Java 类
* Java 类的引用类型静态变量
* Java类的运行时常量池里的引用类型常量
* VM的一些静态数据结构里指向GC堆里的对象的引用
* 等等

简单地说，GC Root 就是经过精心挑选的一组活跃引用，这些引用是肯定存活的。那么通过这些引用延伸到的对象，自然也是存活的。

## **如何进行垃圾回收？**

垃圾回收算法简单地说有三种算法：标记清除算法、复制算法、标记压缩算法。



# JVM垃圾回收器

Java 虚拟机的垃圾回收器可以分为四大类别：串行回收器、并行回收器、CMS 回收器、G1 回收器。



# 垃圾回收得类型

## Minor GC

从年轻代空间回收内存被称为 Minor GC，有时候也称之为 Young GC。对于 Minor GC，你需要知道的一些点：

当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 Eden 区满了。所以 Eden 区越小，越频繁执行 Minor GC。

当年轻代中的 Eden 区分配满的时候，年轻代中的部分对象会晋升到老年代，所以 Minor GC 后老年代的占用量通常会有所升高。

质疑常规的认知，所有的 Minor GC 都会触发 Stop-The-World，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的，因为大部分 Eden 区中的对象都能被认为是垃圾，永远也不会被复制到 Survivor 区或者老年代空间。如果情况相反，即 Eden 区大部分新生对象不符合 GC 条件（即他们不被垃圾回收器收集），那么 Minor GC 执行时暂停的时间将会长很多（因为他们要JVM要将他们复制到 Survivor 区或老年代）。

## Major GC

从老年代空间回收内存被称为 Major GC，有时候也称之为 Old GC。

许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的。

Minor GC 作用于年轻代，Major GC 作用于老年代。 分配对象内存时发现内存不够，触发 Minor GC。Minor GC 会将对象移到老年代中，如果此时老年代空间不够，那么触发 Major GC。因此才会说，许多 Major GC 是由 Minor GC 引起的。

## Full GC

Full GC 是清理整个堆空间 —— 包括年轻代、老年代和永久代（如果有的话）。因此 Full GC 可以说是 Minor GC 和 Major GC 的结合。

当准备要触发一次 Minor GC 时，如果发现年轻代的剩余空间比以往晋升的空间小，则不会触发 Minor GC 而是转为触发 Full GC。因为JVM此时认为：之前这么大空间的时候已经发生对象晋升了，那现在剩余空间更小了，那么很大概率上也会发生对象晋升。既然如此，那么我就直接帮你把事情给做了吧，直接来一次 Full GC，整理一下老年代和年轻代的空间。

另外，即在永久代分配空间但已经没有足够空间时，也会触发 Full GC。

## Stop-The-World

Stop-The-World，中文一般翻译为全世界暂停，是指在进行垃圾回收时因为标记或清理的需要，必须让所有执行任务的线程停止执行任务，从而让垃圾回收线程回收垃圾的时间间隔。

在 Stop-The-World 这段时间里，所有非垃圾回收线程都无法工作，都暂停下来。只有等到垃圾回收线程工作完成才可以继续工作。可以看出，Stop-The-World 时间的长短将关系到应用程序的响应时间，因此在 GC 过程中，Stop-The-World 的时间是一个非常重要的指标。

# JVM 关于堆空间的参数配置

| **参数** | **含义** |
| --- | --- |
| -Xms | 初始堆大小 |
| -Xmx | 最大堆空间 |
| -Xmn | 设置新生代大小 |
| -XX:SurvivorRatio | 设置新生代eden空间和from/to空间的比例关系 |
| -XX:PermSize | 方法区初始大小 |
| -XX:MaxPermSize | 方法区最大大小 |
| -XX:MetaspaceSize | 元空间GC阈值（JDK1.8） |
| -XX:MaxMetaspaceSize | 最大元空间大小（JDK1.8） |
| -Xss | 栈大小 |
| -XX:MaxDirectMemorySize | 直接内存大小，默认为最大堆空间 |

# 如何查看 JVM 中已经设置的参数，包括显示参数和隐式参数

-XX:+PrintVMOptions 程序运行时，打印虚拟机接受到的命令行显式参数。

-XX:+PrintCommandLineFlags 打印传递给虚拟机的显式和隐式参数。

-XX:+PrintFlagsFinal 打印所有的系统参数的值

# JVM之类追踪信息

| **参数** | **含义** |
| --- | --- |
| -verbose:class | 跟踪类的加载和卸载 |
| -XX:+TraceClassLoading | 跟踪类的加载 |
| -XX:+TraceClassUnloading | 跟踪类的卸载 |
| -XX:+PrintClassHistogram | 显示类信息柱状图 |

# JVM之GC日志配置

| **参数** | **含义** |
| --- | --- |
| -XX:PrintGC | 打印GC日志 |
| -XX:+PrintGCDetails | 打印详细的GC日志。还会在退出前打印堆的详细信息。 |
| -XX:+PrintHeapAtGC | 每次GC前后打印堆信息。 |
| -XX:+PrintGCTimeStamps | 打印GC发生的时间。 |
| -XX:+PrintGCApplicationConcurrentTime | 打印应用程序的执行时间 |
| -XX:+PrintGCApplicationStoppedTime | 打印应用由于GC而产生的停顿时间 |
| -XX:+PrintReferenceGC | 跟踪软引用、弱引用、虚引用和Finallize队列。 |
| -XLoggc | 将GC日志以文件形式输出。 |