JVM是Java的运行时虚拟机，所有的Java程序都是在JVM沙箱中运行，每个Java程序就是一个独立的JVM进程。

谈到Java程序是如何运行的，首先需要理解的肯定是JVM是如何运行的，什么是JVM；要理解我们编写的Java程序，运行起来以后到底是什么样子，本质上就是弄清楚JVM是什么样子。

Java程序的代码是什么样的

Java诞生之初最大的卖点就是编写的代码跨平台可移植性，实现这种可移植性，是因为Java通过平台特定的虚拟机，运行中间的字节码，而不是直接编译成本地二进制代码实现，中间字节码也就是java文件编译后生成的.class文件，Jar包的话，实际上只是一系列.class文件的集合。

编写Java程序，首先需要一个入口点，在运行的时候通过指定MainClass来指定入口点，代码层面主类必须实现一个静态的main函数，运行时虚拟机会从MainClass.main开始执行指令，其他的逻辑只是import和函数调用了。

SDK自带的javac命令，负责将我们编程的Java代码，也就是.java文件，编译成平台无关的字节码；字节码可以在任何操作系统平台上，通过平台对应的JVM执行；JVM执行的时候，运行字节码，根据自己的平台特性，将字节码转换成平台相关的二进制码运行。

javac编译器运行的过程大致分为：词法分析（Token流）、语法分析（语法树）、语义分析（注解语法树），还有代码生成器，根据注解语法树，生成字节码，

语义分析阶段，编译器会做一些操作，将人类友好的代码，做一些处理，转换成更符合机器执行机制的代码，例如全局变量，魔法变量，依赖注入，注解这些魔法机制。大致分为以下步骤：

给类添加默认构造函数

处理注解

检查语义的合法性并进行逻辑判断

数据流分析

对语法树进行语义分析(变量自动转换并去掉语法糖)

JVM是什么

JVM = 类加载器 classloader + 执行引擎 execution engine + 运行时数据区域 runtime data area

JVM就是运行编译好字节码的虚拟机，不同的操作系统和平台上，虚拟机将平台无关的字节码，编译成特定平台的指令去执行。我觉得，JVM首先是一个独立运行在操作系统上的进程。执行java命令运行程序的时候，会启动一个进程，每个独立的程序就运行在一个独立的JVM进程里。JVM负责执行字节码，从而实现程序要完成的所有功能。

JVM主要由三部分组成：类加载器、运行时数据区和执行引擎。类加载器加载编译好的.class文件，将所有类结构和方法变量放入运行时数据区，初始化之后，将程序的执行交给执行引擎；JIT编译器，负责将字节码编译成平台特定的二进制码，调用本地接口库。垃圾回收器作为执行引擎的一部分，负责维护运行时数据区中可变的应用程序内存空间。

类加载器（ClassLoader）

类加载器将类加载到内存，并管理类的生命周期，知道将类从内存中卸载结束生命周期。

系统提供了三种类加载器，分别用于不同类的加载：

启动类加载器（Bootstrap ClassLoader），该加载器会将<JAVA\_HOME>lib目录下能被虚拟机识别的类加载到内存中，也就是系统类

扩展类加载器（Extension ClassLoader），该加载器会将<JAVA\_HOME>libext目录下的类库加载到内存

应用程序类加载器（Application ClassLoader），该加载器负责加载用户路径上所指定的类库。

运行时数据区（Runtime Data Area）

运行时数据区，是JVM运行时，在内存中分配的空间。

运行时数据区，被分为五个不同的结构：

Java虚拟机栈（Java Stacks）: 也叫栈内存，主管Java程序的运行，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束该栈就Over，生命周期和线程一致，是线程私有的。

本地方法栈（Native Method Memory）: 登记的native方法，执行引擎执行时加载

程序寄存器（PC Registers）: 当前线程所执行字节码的指针，存储每个线程下一步要执行的字节码JVM指令。

Java堆（Heap Memory）: 应用的对象和数据都是存在这个区域，这块区域也是线程共享的，也是gc 主要的回收区，一个 JVM 实例只存在一个堆类存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，以方便执行器执行。

方法区（Method Area）: 所有定义的方法的信息都保存在该区域，此区域属于共享区间。静态变量+常量+类信息+运行时常量池存在方法区中，实例变量存在堆内存中。

其中的程序寄存器、Java虚拟机栈是按照线程分配的，每个线程都有自己私有的独立空间。

运行的方法和运行期数据，以栈帧的形式存储在运行时JVM虚拟机栈中，栈帧中保存了本地变量，包括输入输出参数和本地变量；保存类文件和方法等帧数据，还记录了出栈入栈操作。每一个方法被调用直至执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

堆在JVM是所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁。

执行引擎（Execution Engine）

执行引擎由三个模块组成，分别是执行引擎，JIT Compiler和Garbage Collector，执行引擎的核心是Jit Compiler，执行字节码或者本地方法；垃圾回收器，则是一系列线程，负责管理分代堆内存。

三个模块分别是运行时计算和运行时内存的管理，负责执行运行时指令的是执行引擎，通过程序寄存器和虚拟机栈中的栈帧出入栈实现方法和指令的执行。GC则负责堆内存的管理，因为GC的时候需要停止指令的执行，消耗资源，所以采用分代方式管理对象收集。JIT则是把字节码编译成本地二进制代码，并调用本地库执行。

GC垃圾回收机制

Java的内存管理，主要是针对的堆内存，因为堆内存是运行时程序和数据分配的空间；不同于内存的其他区域，加载完程序之后，基本上可以确定需要占用的空间大小；heap memory 空间会在运行时动态的分配，无法预测，可大可小，而且快速变化，管理不慎就容易产生内存溢出，所以由JVM提供了强大的分代内存管理机制。

JVM 使用分代内存管理，来分配运行时的堆内存结构，针对不同的世代，采用不同的垃圾回收算法。

常用垃圾回收算法

引用计数器法（Reference Counting）

标记清除法（Mark-Sweep）

复制算法（Coping）

标记压缩法（Mark-Compact）

分代算法（Generational Collecting）

分区算法（Region）

堆内存的组成

heap 的组成有三区域/世代：分别是新生代（Young Generation）、老生代（Old Generation/tenured）和永久区（Perm）。

新生代堆内存又分成Eden区和两个生存区，其中Eden区和生存区的占比为8:1:1，在清理新生代内存的时候，使用的是复制清除算法，优点是清除以后不会产生碎片；简单的复制算法，将内存分成大小相同的两个区域，每次周期只分配其中的一半，这样空间利用率比较低，只使用了一半的内存。

考虑到新生代内存区的对象都是周期很短的，所以JVM实现了一种优化的复制算法，设置一个较大的Eden区来分配对象内存，Eden区空间不够了触发垃圾回收，将上一个生存区和Eden区中还存活的对象，复制到空闲的生存区。然后清空上述两个区域，这样就不会产生内存碎片。

将清理一定次数（15次）还生存的对象，定期晋升到老生代内存区，如果生存区空间不够了，则马上就会触发晋升机制。将部分对象直接晋升到老生代。

如果晋升之后，发现老生代内存不够，就会触发完整的全局GC，清理老生代和新生代内存，老生代内存清理需要使用标记清除和标记整理两种算法。

GC工作原理

分配内存的时候，首先分配到新生代的Eden区，如果Eden区满了，就会发起一次Minor GC，将Eden和From Survivor生存的对象，拷贝到To Survivor Space，如果清理过程中，to Space的空间占用达到一定阈值，或者有对象经历Minor GC的次数达标，就会将对象移动到老生代内存。如果移动过程中发现，老生代内存的空间已经不够了。这时就需要发起Full GC，先进行一次Minor GC，然后通过CMS进行标记清除算法，清理老生代内存，老生代内存经历标记清除之后，因为会产生内存碎片，还需要采用标记整理算法，将所有内存块往前移动，形成连续的内存空间。

老生代标记清除的优点是不需要额外空间。不同于老生代清除算法，会产生碎片，而且标记算法的成本开销也很大；在新生代清除中，因为考虑到大多数新生代对象生存期都是很短暂的，可以使用一种空间换时间的思路，拿出一部分内存空间不分配，而是作为中转，将每次检查时还生存的对象拷贝到Survivor Space，然后直接清除所有原区域的对象，因为大量对象都是生存周期极短的，所以Survivor Space的空间可以远小于正常分配的空间。

不同于引用计数方法，Java使用一种 GC Roots 的对象作为起点开始检查对象，当一个对象到 GC Roots 没有任何引用链相连时, 即该对象不可达, 也就说明此对象是不可用的。就会在GC的时候收回。

GC清理类型的时候，为了防止程序地址出现异常，需要stop the world，清理线程会停止所有运行线程，直到清理完，这个时候是影响性能的。

垃圾回收器的本质

垃圾回收器在JVM层面，是由一系列不同的组件组成的，每种组件是一个独立线程，分别执行自己的逻辑。

新生代垃圾收集器：

Serial（串行）收集器是最基本、发展历史最悠久的收集器，它是采用复制算法的新生代收集器，。它是一个单线程收集器，只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须暂停其他所有的工作线程，直至Serial收集器收集结束为止（“Stop The World”）。

ParNew收集器就是Serial收集器的多线程版本，它也是一个新生代收集器。除了使用多线程进行垃圾收集外，其余行为包括Serial收集器可用的所有控制参数、收集算法（复制算法）、Stop The World、对象分配规则、回收策略等与Serial收集器完全相同，两者共用了相当多的代码。

Parallel Scavenge收集器也是一个并行的多线程新生代收集器，它也使用复制算法。Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标是达到一个可控制的吞吐量（Throughput）。

老生代垃圾收集器：

Serial Old 是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”（Mark-Compact）算法

Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法

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，优点是：并发收集、低停顿，因此CMS收集器也被称为并发低停顿收集器（Concurrent Low Pause Collector）

面向服务端的G1收集器。

G1收集器是一款面向服务端应用的垃圾收集器。

在使用G1收集器时，Java堆的内存布局和其他收集器有很大的差别，它将这个Java堆分为多个大小相等的独立区域，虽然还保留新生代和老年代的概念，但是新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

GC回收的触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC

Full GC触发条件：

gc()方法的调用

老年代代空间不足

方法区空间不足

CMS GC时出现promotion failed和concurrent mode failure

统计得到的Minor GC晋升到旧生代的平均大小大于老年代的剩余空间

堆中分配很大的对象

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

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

GC Roots

在Java语言中，可以作为GC Roots的对象包括下面几种：

虚拟机栈（栈帧中的本地变量表）中引用的对象；

方法区中类静态属性引用的对象；

方法区中常量引用的对象；

本地方法栈中JNI（即一般说的Native方法）引用的对象；

总结就是，方法运行时，方法中引用的对象；类的静态变量引用的对象；类中常量引用的对象；Native方法中引用的对象。