1. JVM 内存的分类

程序计数器:当前线程执行的代码的行号
虚拟机栈:
本地方法栈:
Java 堆:
方法区:
运行时常量池(属于方法区):
直接内存:

2. GC 标记对象的死活

A. 引用计数法

给对象添加一个引用计数器，没当被引用的时候，计数器的值就加一。引用失效的时候减一，当计数器的值为 0 的时候就表示改对象可以被 GC 回收了。

弊端：A-》B，B-》A，那么 AB 将永远不会被回收了。也就是引用有环的情况。

B. 根搜索算法 GC Roots Tracing

通过一个叫 GC Roots 的对象作为起点，从这些结点开始向下搜索，搜索所走过的路径称为引用链，当一个对象没有与任何的引用链相连的时候则改对象就可以被 GC 回收回收了。GC Roots 包括：java 虚拟机栈中引用的对象，本地方法栈中引用的对象，方法区中常量引用的对象，方法区中静态属性引用的对象。

当一个对象没有出现在引用链的时候，对象的生命就被判了缓刑：如果对象没有重写 finalize()方法或者 finalize()方法已经被执行了一次了，这时候对象在 GC 来到时候一定会被回收了，否则会进入一个 F-Queue 队列，执行 finalize()方法。

3. GC 回收算法

A. 标记-清除法：标记出没有用的对象，然后一个一个回收掉。两个问题：效率和空间问题。产生很多的内存碎片，怡然不可以使用。
B. 复制算法： 按照容量，划分二个大小相等的内存区域，当一块用完的时候将存活的对象复制过来。
C. 标记-整理法：将存活的对象向前移动，连续起来，然后将端以外的内存区域全部清空。
D. 划代收集算法：一般分为新生代和老年代，新生代一般使用复制算法，老年代使用“标记-清除”或者“标记-整理”。

Java 堆内存的分配策略：

1. Young 区

又分为 Eden 区，survivior 1 和 survivior 2。
一般小型的对象都会在 Eden 区上分配内存，当内存不够时候发生 minor GC(年轻代的 GC)，将存活的对象拷贝到 survivor 1 区域，(此时 2 是空的)，然后清空 Eden
在清理 survivor1 的时候将存活的拷贝到 2 区域，长时间存活的可以晋升到 old 区。

这里是停止-复制算法。

2.Old 区一般使用标记清除或者标记整理的方法进行垃圾回收。

4. Class 的加载过程
一个 class 在内存中要经过七个阶段:加载,验证,准备,解析,初始化,使用.卸载

类加载过程:加载.验证.准备.解析.初始化
这个三个阶段称为连接过程

1.加载过程:
1) 通过一个全类限定名称来获取定义此类的二进制字节流.
2) 将这个字节流所代表的静态数据结构转化为方法区的运行时数据结构
3) 在 java 堆生成一个代表这个类的 java.lang.class 对象，作为方法区这些数据访问的入口。

2.验证过程：
确保二进制字节流中包含的信息符合当前虚拟机的要求，并且不会危害到虚拟机自身的安全。
文件格式验证，元数据验证，字节码验证，符号引用验证，

3.准备过程：
为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。
注意两点：1. 类变量指的是 static 类型的变量，实例变量分配在堆内存，不在这初始化。
如果仅仅是一个 static 变量，那么将会被初始化为 0，要是 final static，则被初始化为指定的数值。 -==》 书上原话是这样的，通常情况下将会被初始化为 0。 Static 直到被初始化后才会被初始化为指定的数值。

4.解析阶段：
虚拟机将常量池内的符号引用替换成直接引用的过程
符号引用-以符号来描述引用的目标
直接引用-可以是指向目标的指针，相对偏移量，或者能间接定位到目标的句柄

5.初始化阶段
初始化阶段就是执行类构造器<clinit>方法的过程。
该方法是编译器自动收集类中的多有类变量的赋值动作和静态语句块中的语句合并而成。编译器收集的顺序就是语句在源文件中出现的顺序，静态语句块只能访问到他之前的变量，定义在他之前的变量只能赋值不能访问。
父类的《clinit》方法在子类的之前调用完毕

5. 类加载器

Class 和 classloader 一同确立了 class 在虚拟机中的唯一性。

Classloader 有两种：1.启动类加载器，由 c++完成，属于虚拟机的一部分。2.所有其他类加载器，由 java 语言实现，全部继承自抽象类 java.lang.ClassLoader

细分又可分为三类：

启动类加载器

扩展类加载器

应用程序类加载器

还可以有自定义类加载器

以上则称为：双亲委派模型

工作过程如下：如果一个类收到了类加载请求，他首先不会自己尝试去加载这个类，而是把这个加载请求委派给父类加载器去完成，每一层的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求的时候，子类加载器才回去尝试自己加载这个类。

6. 引起类初始化操作的四个行为（有且仅有这四个）：

1.使用 new 关键字创建对象的时候，读取或者设置一个类的自动字段的时候（final 类得不会，因为在准备阶段因为复制）以及调用一个类的静态方法的时候。

2.反射调用的时候

3.子类初始化的时候

4.改类为主类的时候(有 main(string[] args))

7. 静态分派和动态分派

Class 文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址。这个特性给 Java 带来了更强大的动态扩展能力，使得可以在类运行期间才能确定某些目标方法的直接引用，称为动态连接，也有一部分方法的符号引用在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。这在前面的“Java 内存区域与内存溢出”一文中有提到。

静态解析成立的前提是：方法在程序真正执行前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在编译器进行编译时就必须确定下来，这类方法的调用称为解析。

在 Java 语言中，符合“编译器可知，运行期不可变”这个要求的方法主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法都不可能通过继承或别的方式重写出其他的版本，因此它们都适合在类加载阶段进行解析。

Java 虚拟机里共提供了四条方法调用字节指令，分别是：

invokestatic：调用静态方法。

invokespecial：调用实例构造器<init>方法、私有方法和父类方法。

invokevirtual：调用所有的虚方法。

invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器和父类方法四类，它们在类

加载时就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法（还包括 final 方法），与之相反，其他方法就称为虚方法（final 方法除外）。这里要特别说明下 final 方法，虽然调用 final 方法使用的是 invokevirtual 指令，但是由于它无法覆盖，没有其他版本，所以也无需对方发接收者进行多态选择。Java 语言规范中明确说明了 final 方法是一种非虚方法。

解析调用一定是个静态过程，在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用转化为可确定的直接引用，不会延迟到运行期再去完成。而分派调用则可能是静态的也可能是动态的，根据分派依据的宗量数（方法的调用者和方法的参数统称为方法的宗量）又可分为单分派和多分派。两类分派方式两两组合便构成了静态单分派、静态多分派、动态单分派、动态多分派四种分派情况。

静态分派

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派，静态分派的最典型应用就是多态性中的方法重载。静态分派发生在编译阶段，因此确定静态分配的动作实际上不是由虚拟机来执行的。

动态分派

动态分派与多态性的另一个重要体现——方法覆写有着很紧密的关系。向上转型后调用子类覆写的方法便是一个很好地说明动态分派的例子。这种情况很常见，因此这里不再用示例程序进行分析。很显然，在判断执行父类中的方法还是子类中覆盖的方法时，如果用静态类型来判断，那么无论怎么进行向上转型，都只会调用父类中的方法，但实际情况是，根据对父类实例化的子类的不同，调用的是不同子类中覆写的方法，很明显，这里是要根据变量的实际类型来分派方法的执行版本的。而实际类型的确定需要在程序运行时才能确定下来，这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

单分派和多分派

前面给出：方法的接受者（亦即方法的调用者）与方法的参数统称为方法的宗量。但分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。

我们首先来看编译阶段编译器的选择过程，即静态分派过程。这时候选择目标方法的依据有两点：一是方法的接受者（即调用者）的静态类型是 Father 还是 Child，二是方法参数类型是 Eat 还是 Drink。因为是根据两个宗量进行选择，所以 Java 语言的静态分派属于多分派类型。

再来看运行阶段虚拟机的选择，即动态分派过程。由于编译期已经了确定了目标方法的参数类型（编译期根据参数的静态类型进行静态分派），因此唯一可以影响到虚拟机选择的因素只有此方法的接受者的实际类型是 Father 还是 Child。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于单分派类型。