1. 内存模型以及分区，需要详细到每个区放什么。

JVM 分为堆区和栈区，还有方法区，初始化的对象放在堆里面，引用放在栈里面，class 类信息常量池（static 常量和 static 变量）等放在方法区

new:

- 方法区：主要是存储类信息，常量池（static 常量和 static 变量），编译后的代码（字节码）等数据
- 堆：初始化的对象，成员变量 （那种非 static 的变量），所有的对象实例和数组都要在堆上分配
- 栈：栈的结构是栈帧组成的，调用一个方法就压入一帧，帧上面存储局部变量表，操作数栈，方法出口等信息，局部变量表存放的是 8 大基础类型加上一个应用类型，所以还是一个指向地址的指针
- 本地方法栈：主要为 Native 方法服务
- 程序计数器：记录当前线程执行的行号

2. 堆里面的分区：Eden，survival （from+ to），老年代，各自的特点。

堆里面分为新生代和老生代（java8 取消了永久代，采用了 Metaspace），新生代包含 Eden+Survivor 区，survivor 区里面分为 from 和 to 区，内存回收时，如果用的是复制算法，从 from 复制到 to，当经过一次或者多次 GC 之后，存活下来的对象会被移动到老年区，当 JVM 内存不够用的时候，会触发 Full GC，清理 JVM 老年区

当新生区满了之后会触发 YGC,先把存活的对象放到其中一个 Survice 区，然后进行垃圾清理。因为如果仅仅清理需要删除的对象，这样会导致内存碎片，因此一般会把 Eden 进行完全的清理，然后整理内存。那么下次 GC 的时候，就会使用下一个 Survive，这样循环使用。如果有特别大的对象，新生代放不下，就会使用老年代的担保，直接放到老年代里面。因为 JVM 认为，一般大对象的存活时间一般比较久远。

3. 对象创建方法，对象的内存分配，对象的访问定位。

new 一个对象

4. GC 的两种判定方法：

引用计数法：指的是如果某个地方引用了这个对象就+1，如果失效了就-1，当为 0 就会回收但是 JVM 没有用这种方式，因为无法判定相互循环引用（A 引用 B,B 引用 A）的情况

引用链法： 通过一种 GC ROOT 的对象（方法区中静态变量引用的对象等-static 变量）来判断，如果有一条链能够到达 GC ROOT 就说明，不能到达 GC ROOT 就说明可以回收

5. SafePoint 是什么

比如 GC 的时候必须要等到 Java 线程都进入到 safepoint 的时候 VMThread 才能开始执行 GC，

1. 循环的末尾 (防止大循环的时候一直不进入 safepoint，而其他线程在等待它进入 safepoint)
2. 方法返回前
3. 调用方法的 call 之后
4. 抛出异常的位置

6. GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？

先标记，标记完毕之后再清除，效率不高，会产生碎片

复制算法：分为 8：1 的 Eden 区和 survivor 区，就是上面谈到的 YGC
标记整理：标记完毕之后，让所有存活的对象向一端移动

7. GC 收集器有哪些？CMS 收集器与 G1 收集器的特点。

并行收集器：串行收集器使用一个单独的线程进行收集，GC 时服务有停顿时间
串行收集器：次要回收中使用多线程来执行
CMS 收集器是基于**“标记—清除”**算法实现的，经过多次标记才会被清除
G1 从**整体来看是基于“标记—整理”**算法实现的收集器，从**局部（两个 Region 之间）上来看是基于“复制”**算法实现的

8. Minor GC 与 Full GC 分别在什么时候发生？

新生代内存不够用时候发生 MGC 也叫 YGC，JVM 内存不够的时候发生 FGC

9. 几种常用的内存调试工具：jmap、jstack、jconsole、jhat

jstack 可以看当前栈的情况，jmap 查看内存，jhat 进行 dump 堆的信息
mat（eclipse 的也要了解一下）

10. 类加载的几个过程：

加载、验证、准备、解析、初始化。然后是使用和卸载了
通过全限定名来加载生成 class 对象到内存中，然后进行验证这个 class 文件，包括文件格式校验、元数据验证，字节码校验等。准备是对这个对象分配内存。解析是将符号引用转化为直接引用（指针引用），初始化就是开始执行构造器的代码

11.JVM 内存分哪几个区，每个区的作用是什么?

java 虚拟机主要分为以下一个区:

方法区：

1. 有时候也成为**永久代**，在该区内很少发生垃圾回收，但是并不代表不发生 GC，在这里进行的 GC 主要是对方法区里的常量池和对类型的卸载

2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。

3. 该区域是被线程共享的。

4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

虚拟机栈:

1. 虚拟机栈也就是我们平常所称的**栈内存**,它为 java 方法服务，每个方法在执行的时候都

会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

2. 虚拟机栈是线程私有的，它的生命周期与线程相同。

3. 局部变量表里存储的是基本数据类型、returnAddress 类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定

4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式

5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。

堆

java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

程序计数器

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个 java 虚拟机规范没有规定任何 OOM 情况的区域。

12.如和判断一个对象是否存活?(或者 GC 对象的判定方法)

判断一个对象是否存活有两种方法:

1. **引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A,B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2.**可达性算法**(引用链法)

该算法的思想是：从一个被称为 **GC Roots** 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。

在 java 中可以作为 GC Roots 的对象有以下几种:

- 虚拟机栈中引用的对象
- 方法区类静态属性引用的对象
- 方法区常量池引用的对象
- 本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比**不一定会被回收**。当一个对象不可达 GC Root 时，这个对象并

不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记

如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法

或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行 finalize()方法，那么这个对象将会放在一个称为 F-Queue 的对队列中，虚拟机会触发一个 Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize()执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行第二次被标记，这时，该对象将被移除”即将回收”集合，等待回收。

13.简述 java 垃圾回收机制?

在 java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

14.java 中垃圾收集的方法有哪些?

1. **标记-清除:**

   这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作。

2. **复制算法:**

   为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，

内存的代价太高，每次基本上都要浪费一般的内存。

于是将该算法进行了改进，内存区域不再是按照 1：1 去划分，而是将内存划分为 8:1:1 三部分，较大那份内存交 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。(java 堆又分为新生代和老年代)

3. **标记-整理**

   该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

4. **分代收集**

   现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用**复制**算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用**标记-整理** 或者 **标记-清除。**

15.java 内存模型

java 内存模型(JMM)是线程间通信的控制机制.JMM 定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java 内存模型的抽象示意图如下：

从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1. 首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。

2. 然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

16.java 类加载过程?

java 类加载需要经历一下 7 个过程：

加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。

2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。

3. 在内存中生成该类的 Class 对象，作为该类的数据访问入口。

验证

验证的目的是为了确保 Class 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.

2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。

3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。

4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

```
    public static int value=123;//在准备阶段 value 初始值为 0 。在初始化阶段才会变为 123 。
```

```
1
2
```

- 1
- 2

解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

17. 简述 java 类加载机制?

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的 java 类型。

18. 类加载器双亲委派模型机制？

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

19.什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。

2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

20.简述 java 内存分配与回收策率以及 Minor GC 和 Major GC

1. 对象优先在堆的 Eden 区分配。

2. 大对象直接进入老年代.

3. 长期存活的对象将直接进入老年代.

   当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC.Minor Gc 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC,但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。