**1.jvm内存布局**

1.程序计数器：当前线程正在执行的字节码的行号指示器，线程私有，唯一一个没有规定任何内存溢出错误的情况的区域。

2.Java虚拟机栈：线程私有，描述Java方法执行的内存模型，每个方法运行时都会创建一个栈帧，存放局部变量表、操作数栈、动态链接、方法出口等信息，每个方法的运行到结束对应一个栈帧的入栈和出栈。会有StackOverFlowError异常（申请的栈深度大于虚拟机所允许深度）和OutOfMemoryError异常（线程无法申请到足够内存）。

3.本地方法栈：功能与Java虚拟机栈相同，不过是为Native方法服务。

4.java堆：线程共享，存放实例对象和数组对象，申请空间不足抛出OutOfMemoryError异常。

5.方法区：线程共享，存储已被虚拟机加载的类的类信息、常量、静态变量、编译后的代码；运行时常量池存放class文件中描述的符号引用和直接引用，具有动态性。方法空间不足时抛出OutOfMemoryError异常。

6.直接内存：JVM规范之外的，NIO类引入了一种基于通道和缓冲区的I/O方式，可使用Native函数库直接分配内存，通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作，避免了在Java堆和Native堆中来回复制数据。

**2.垃圾回收算法与垃圾回收器**

垃圾收集算法：

1.标记-清除算法：将所有需要回收的对象先进行标记，标记结束后对标记的对象进行回收，效率低，会造成大量碎片。

2.复制算法：将内存分为两块大小相等的空间，每次只用其中一块，若一块内存用完了，就将这块内存中活着的对象复制到另一快内存中，将已使用的进行清除。不会产生碎片，但是会浪费一定的内存空间。堆的年轻代使用此算法，因为年轻代对象多为生存周期比较短的对象。年轻代将空间分为一个Eden和两个survivor，每次只使用Eden加一个survivor，回收时，将Eden和survivor中存活的对象复制到另一个survivor上，最后清理Eden和survivor。当Eden与survivor存活对象大于另一个survivor空间大小则需要老年代来担保。

3.标记-整理算法：标记阶段与标记-清除算法相同，标记完成后将所有存活对象向一端移动，然后清除掉端边界外对象。

4.分代收集算法：根据对象存活周期分为将内存分为新生代与老年代，新生代采取复制算法，老年代采用标记清除或标记整理算法。

垃圾回收器：

1.Serial收集器：单线程，垃圾回收时需要停下所有的线程工作。

2.ParNew收集器：Serial的多线程版本。

3.Parallel Scavenge收集器：年轻代，多线程并行收集。设计目标是实现一个可控的吞吐量（cpu运行代码时间/cpu消耗的总时间）。

4.Serial Old收集器：Serial老年代版本。

5.CMS：目标是获得最短回收停顿时间，基于标记清除算法，整个过程四个步骤：初始标记（标记GCRoot直接关联对象，速度很快）、并发标记（从GCRoot向下标记）、重新标记（并发标记过程中发生变化的对象）、并发清除（清除老年代垃圾）。初始标记和重新标记需要停顿所有用户线程。缺点：无法处理浮动垃圾、有空间碎片的产生、对CPU敏感。

6.G1收集器：唯一一个可同时用于老年代与新生代的收集器。采用标记整理算法，将堆分为不同大小星等的Region，G1追踪每个region的垃圾堆积的价值大小，然后有一个优先列表，优先回收价值最大的region，避免在整个堆中进行安全区域的垃圾收集，能建立可预测的停顿时间模型。整个过程四个步骤：初始标记、并发标记、最终标记（并发标记阶段发生变化的对象的变化记录写入线程remembered set log，同时与remembered set合并）、筛选回收（对每个region回收价值和成本拍寻，得到一个最好的回收方案并回收）。

**3.垃圾回收对象时程序的逻辑是否可以继续执行**

不同回收器不同：Serial、ParNew会暂停用户所有线程工作；CMS、G1会在某一阶段暂停用户线程。

**4.内存分配策略**

1.对象优先在Eden分配：若Eden无空间，Java虚拟机发起一次Minor GC。

2.大对象直接进入老年代：大对象指需要大量连续内存空间的对象（如长数组、长字符串）

3.长期存活的对象进入老年代：每个对象有一个对象年龄计数器，age=15晋升为老年代。age+1的两个情况：对象在Eden出生并经过一次Minor GC存活且被survivor容纳；在survivor区经历过一次minor GC。

4.空间分配担保：

1.在Minor GC之前，先检查老年代最大可用连续空间是否大于新生代所有空间总和，成立则此次GC安全

2.不成立，查看是否允许担保失败设置为true，不允许则进行Full GC

3.允许，看老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小，不成立则Full GC

4.成立，则进行Minor GC

**5.Java中的引用**

1.强引用：new这类引用，只要强引用在，对象永远不会被回收。

2.软引用：描述有用但非必需的对象，在内存溢出之前，会把这些对象列入回收范围内进行第二次垃圾回收。

3.弱引用：描述非必需对象，只存活到下一次垃圾回收前。

4.虚引用：不会对生存时间造成影响，不能通过虚引用获得对象实例，只是在被虚引用的对象被回收时受到一个系统通知。

**6.简述minor gc和full gc**

Minor GC：从新生代回收内存，关键是Eden区内存不足，造成不足的原因是Java对象大部分是朝生夕死(java局部对象)，而死掉的对象就需要在合适的时机被JVM回收

Major GC：从老年代回收内存，一般比Minor GC慢10倍以上。

Full GC：对整个堆来说的，出现Full GC通常伴随至少一次Minor GC，但非绝对。Full GC被触发的时候：老年代内存不足；持久代内存不足；统计得到的Minor GC晋升到老年代平均大小大于老年代空间。

**7.java虚拟机new一个对象的创建过程**

1.在常量池中查看是否有new的参数对应的类的符号引用，并检查这个符号引用对应的类是否被加载、解析、初始化

2.加载后，为新对象分配内存空间，对象多需要的内存大小在类被加载之后就被确定（堆内分配内存：指针碰撞、空闲列表）。

3.将分配的空间初始化为零值。

4.对对象头进行必要设置（实例是哪个类的实例、类的元信息数据、GC分代年龄等）。

5.执行<init>方法，按照程序的值初始化。

**8.java中的类加载机制**

Java虚拟机中类加载过程：加载、验证、准备、解析、初始化。

1.加载：通过一个类的全限名来获取定义此类的二进制字节流；将这个字节流代表的静态存储结构转换为方法区的的动态存储结构；在内存中生成一个代表此类的java.lang.Class对象，作为方法区中这个类的访问入口。

2.验证：验证class文件中的字节流是否符合Java虚拟机规范，包括文件格式、元数据等。

3.准备：为类变量分配内存并设置类变量初始值，分配内存在方法区。

4.解析：将常量池中符号引用替换为直接引用的过程；符号引用与虚拟机实现的内存布局无关，是使用一组符号来描述所引用的目标。class文件中不会保存各个方法的最终布局信息，所以这些符号引用不经过转化是无法得到真正的内存入口地址；直接引用与虚拟机实现的内存布局有关，可以是直接指向目标的指针，偏移量或指向目标的句柄。此过程主要是静态链接，方法主要为静态方法和私有方法。

5.初始化：真正执行类中定义的Java代码。初始化执行类的<clinit>方法，该方法由编译器自动收集类中所有类变量的赋值动作和静态语句块的语句合并产生，且保证子类的clinit调用之前会先执行父类的clinit方法，clinit可以不存在（如没有类变量和静态语句块）。

**9.双亲委派模型**

java中类加载器主要用于实现类的加载，Java中的类和类加载器一起唯一确定类在JVM中的一致性。

系统提供的类加载器：启动类加载器、扩展类加载器、应用程序类加载器。

1.启动类加载器：用C++实现，是JVM的一部分，其他加载器使用Java实现，独立于JVM。主要负责加载<JAVA\_HOME>\lib目录下的类库或被-Xbootclasspath参数指定的路径中的类库，应用程序不能使用该类加载器。

2.扩展类加载器：负责加载<JAVA\_HOME>/lib/ext目录下或者类系统变量java.ext.dirs指定路径下的类库，开发者课直接使用。

3.应用程序类加载器：主要负责加载classpath下的类库，若应用程序没有自定义类加载器，默认使用此加载器

双亲委派模型要求除了启动类加载器，其他类加载器都有自己的父类加载器，使用组合关系来实现复用父类加载器。过程：若一个类加载器收到类加载请求，会把此请求委派给父类加载器去完成，每层都是如此，因此所有的加载请求最后都会传到启动类加载器；只有当父类加载器反馈不能加载，才会把此请求交给子类完成。

好处：使得java类伴随他的类加载器有了优先级；保证Java程序运行的稳定性

**10.简述分派**

包括静态分派与动态分派

1.静态分派：发生在编译时期，所有依赖静态类型来定位方法执行版本的分派称为静态分派，典型应用为方法重载。

2.动态分派：在运行期根据实际类型确定方法执行版本的分派过程。典型应用为方法重写，实现是在方法去中建立方法表，若子类中没有重写父类方法，则子类虚方法表中该方法的入口地址与父类指向相同，否则子类方法表中地址会替换为指向子类重写的方法的入口地址。

**11.对象的内存布局**

对象内存布局分为三部分：对象头、实例数据、对齐填充。

对象头包含两部分：

1.存储对象自身运行时数据：哈希码、分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等

2.对象指向它的类元数据指针--类型指针

实例数据：程序代码中所定义的各种类型的字段内容

对齐填充：不是必然存在，仅起到占位符作用（对象大小必须是8子节整数倍）

**12.虚拟机栈中的各个部分**

1.局部变量表：存放方法参数和方法内部定义的局部变量，以变量槽Slot为基本单位，一个Slot可以存放32位以内的数据类型，可重用。

2.操作数栈：先入后出，32位数据类型所占栈容量为1，64为数据类型所占栈容量为2

3.动态链接：常量池中符号引用有一部分在每次运行期间转换为直接引用，这部分称为动态链接。（一部分在类加载阶段或第一次使用时转换为直接引用---静态解析）

4.方法返回地址：方法执行后退出的两种方式：正常完成出口（执行引擎遇到任意一个返回的字节码指令）和异常完成出口（在方法执行过程中遇到异常且此异常未被处理）。两种方式都需要返回到方法被调用的位置程序才能继续执行（正常退出时调用者的PC计数器的值可以作为返回地址且栈帧中很可能保存这个计数器值；异常退出返回地址要通过异常处理器表来确定，栈帧中一般不会保存）。

**13.Java内存模型的happen before原则**

如果两个操作存在happens-before关系，那么前一个操作的结果就会对后面一个操作可见，是定义的两个操作之间的偏序关系，常见的规则：

1.程序顺序规则：一个线程中每个操作，happens-before于该线程中的任意后续操作

2.监视器锁规则：对一个锁的解锁，happens-before于随后这个锁的加锁

3.volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个域的读

4.传递性：若A happens-before B，B happens-before C，则A happens-before C

5.start()规则：如果线程A执行ThreadB.start()，那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

6.join()规则：若线程A 执行ThreadB.join()并成功返回，则线程B的任意操作happens-before于线程A从ThreadB.jion()操作返回成功。

**14.java中方法区存放哪些东西？jvm如何控制方法区的大小以及内存溢出的原因和解决**

方法区大小不是固定的，jvm可根据需要动态调整。方法区主要存放类信息、常量、静态变量、编译后的代码。

控制方法区大小：减少程序中class数量、尽量使用较少的静态变量

修改：-XX:MaxPerSize调大

StackOverflowError异常：线程的方法嵌套调用层次太多，随着Java栈中桢的增多，最终会由于该线程Java栈中所有栈帧总和大于-Xss设置的值而产生此异常。

**15.jvm OutMemory的种类**

1.堆溢出：被缓存的实例对象，大的map，list引用大的对象等

2.栈溢出：栈帧太多

3.方法区溢出：加载很多类会有可能出现，GC不会在主程序运行期对此区域进行清理，可通过设置jvm启动参数解决：-XX:MaxPermSize=256m

**16.jvm如何判断对象是否失效？可达性分析是否可以解决循环引用**

1.引用计数器算法：给对象添加一个引用计数器，当被引用时给计数器加1，引用失效减1，当为0时对象失效。实现简单，判定效率高，无法解决循环引用问题。

2.可达性分析算法：将一系列GC Root作为起始点，从这些节点开始向下搜索，所走过路径称为引用链，若一个对象无引用链，则判断是否执行finalize()方法，若finalize()被覆盖并且没被JVM调用过，则执行此方法，执行后若还无引用链，则对象失效。

可以作为GC Root的对象：

1. 虚拟机栈中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中Native方法引用的对象