1. Java对象在内存中 三部分 头部信息， 实例数据， 对齐补充
2. 句柄访问，直接指针 1.句柄访问是 定义一个指针表 一个包含类型的实例地址，一个包含类型的类型地址 2.直接指针是 直接就能访问实例地址，然后在获取类型信息
3. 方法区 类名， 访问修饰符，常量池，字段描述，方法描述
4. 不可达意味着该对象可以被回收
5. 复制删除 标记整理

JVM模型

1. 程序计数器

   指向程序执行的字节码地址，如果正在执行的是Native方法，程序计数器为空，不会发生OutofMemoryError

2. 虚拟机栈
    1. 局部变量表
        1. 存放编译可知的基本数据类型以及对象的引用
        2. double 和 long 占据两个 局部变量空间
    2. 方法出口
    3. 动态链接
    4. 操作栈
3. 本地方法栈

   执行Native方法的栈

4. 堆区
    1. 线程共享
    2. 存放对象实体
    3. 分区(方便回收对象)
5. 方法区
    1. class类信息
    2. 静态变量
    3. 常量
6. 运行时常量

   方法区的一部分

垃圾回收器

1. 如何判断一个对象该被回收了
    1. 引用计数器(无法解决对象之间相互引用的问题)
    2. GCRoot搜索
    3. 从GCRoots开始向下搜索，搜索过的路径称为引用链，当一个对象跟 GCRoots没有任何引用链时，也就是说该对象不可达，九二一被回收了
2. 可以作为GCRoots的对象

    1. 虚拟机栈(本地变量表中)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. 本地方法栈中引用的对象
3. 几种引用的区别
    1. 强引用 Object o = new Object(),只要强引用还在，就不会被回收
    2. 软引用 如果系统内存不够时，会发生回收
    3. 弱引用 只能存活到下一次CG前
    4. 虚引用 没啥用
4. 不可达的对象会被标记两次
    1. 发现没有引用链和GCRoots相连，被标记
    2. 被标记的对象会放到执行Finalize()的队列中，可能会被引用而不会被回收，但是只能被拯救一次
5. 无效的类
    1. 该类的所有实例都被回收
    2. 该类的ClassLoader被回收
    3. 该类的Class对象没有被引用，没有任何地方通过反射生产该类
6. 垃圾回收算法
    1. 标记清除(1.内存碎片2.标记和清除的效率都不高)
    2. 复制算法
    3. 标记整理(将所有的对象都移到一端)
7. 垃圾回收器
    1. Serial/Serial Old收集器
        1. 单线程
        2. 新生代
        3. 暂停所有工作线程 Stop the World
    2. ParNew 收集器
        1. Serial的多线程版本
        2. 只能和CMS收集器配合
        3. 仍然需要暂停所有工作线程STW
    3. Parallel Scavenge
        1. 并行多线程收集器(多条收集线程利用多核同时执行)
        2. 适合于运算量比较大的，没有太多交互的任务
        3. 为什么叫吞吐量收集器，像CMS停顿时间短，但是可能停顿发生的很频繁，所以吞吐量就小了，这个收集器停顿总时长短，也就是说可能一次发生的停顿时间长，但是停顿的次数少。所以适合不用频繁停顿的。
        4. 不能与CMS配合，只能和Parallel Old配合
    4. Parallel Old Serial Old都是标记整理算法， Serial ParNew Paralle Scavenge都是暂停复制算法
    5. CMS收集器
        1. 适用于B/S服务器上的一种，因为系统响应短，用户体验好
        2. 过程
            1. 初始标记 STW(GCRoots直接关联的对象)
            2. 并发标记（判断对象是否可达）
            3. 再次标记STW(新生代可能发生了MinorGC，一些对象不可达)
            4. 并发清除

3. 缺点
    1. 对CPU敏感，暂用一部分CPU资源
    2. 如果内存超过一定限额，会触发Full GC
    3. 会产生很多碎片 标记-清除算法
6. G1收集器
    1. 标记-整理算法
    2. 将堆区划分成大小相等的区域进行管理
8. 内存分配策略
    1. 首先在新生代Eden区分配，如果内存不够触发一次Minor GC
    2. 否则分配到老年代
    3. 大对象直接到老年代
    4. 对象晋升老年区会判断内存大小，如果不够触发Full GC

Class对象

9. 1. 首先在新生代Eden区分配，如果内存不够触发一次Minor GC
    2. 否则分配到老年代
    3. 大对象直接到老年代
    4. 对象晋升老年区会判断内存大小，如果不够触发Full GC
10.

1. Class文件结构
    1. MagicNumber魔数，用来确定是不是.class文件
    2. Class文件的版本号，用来判断Jdk是否支持
    3. 常量池
        1. 字面量，文本字符串，被声明为final的常量
        2. 符号引用
            1. 类和接口的名称
            2. 字段名称和描述符
            3. 方法名称和描述符
    4. 访问标志
        - 是否final
        - 是否public，否则是private
        - 是否是接口
        - 是否可用invokespecial字节码指令
        - 是否是abstact
        - 是否是注解
        - 是否是枚举
    5. 类索引、父类索引和接口索引集合
    6. 字段表集合(除了方法内声明的变量)
    7. 方法表集合
    8. 属性表集合
2. 初始话的时机
    1. 非final的static变量

2. final static String s = new String("s");
3. 反射
4. new一个对象
5. 调用静态方法

3. 类加载过程
    1. 加载
        1. 根据类命获取这个类的二进制字节流
        2. 将静态存储结构转化为方法区运行时数据结构
        3. 在堆中生成一个代表这个类的java.lang.Class对象，作为方法区数据访问入口
    2. 验证
        1. 文件格式验证
        2. 元数据验证
            1. 是否有父类
            2. 父类是否时final的
            3. 字段，方法是否和父类冲突
        3. 字节码验证
        4. 符合引用验证
    3. 准备
        1. 对static类变量分配内存并设置默认值
        2. 都在方法区中分配
        3. 如果是static final 设置为指定的值
    4. 解析 将符号引用替换为直接引用
        1. 类或接口解析
        2. 字段解析
        3. 方法解析
        4. 接口方法解析
    5. 初始化
        1. 父类static变量
        2. 父类static代码块
        3. 子类static变量
        4. 子类static代码块
        5. 父类普通代码块
        6. 父类构造器
        7. 子类普通代码块
        8. 子类构造器
4. 双亲委派模型
    1. Bootstrap 类加载器
    2. Extension 类加载器
    3. Application 类加载器
    4. 定义

        除了顶级父类，都要把类加载的任务交给父类加载器处理，父类加载器处理不了，自己在处理