- JAVA核心
  - JUC多线程以及高并发
    - 谈谈你对volatile的理解
    - 谈谈对CAS的理解
      - 比较，符合条件时交换
      - 使用UnSafe类
      - CAS示例代码
      - CAS的缺点
    - 线程不安全的集合类，解决方案 —— 例子
    - 公平锁/非公平锁/可重入锁/递归锁/自旋锁的理解
      - 公平锁 —— 吞吐量不如非公平锁
      - 非公平锁 —— 会造成线程饿死和顺序反转的问题
      - 可重入锁（递归锁）
        - 同一线程中，外层方法获取锁后，内层方法自动获取锁
        - ReentrantLock、synchronize
      - 自旋锁 —— CAS —— 线程不阻塞，代码循环执行尝试修改，好处是减少线程上下文切换，缺点是持续占用CPU资源
      - CountDownLatch
      - CyclicBarrier
      - Semaphore
      - 阻塞队列
      - Callable接口
      - 线程池
        - 为什么用线程池，优势
          - 降低资源消耗 —— 通过复用已创建的线程降低线程创建销毁造成的资源消耗
          - 提高响应速度，任务达到时不需要等待线程创建就能立即执行
          - 提高线程的可管理性，线程是稀缺资源，不能无限创建，使用线程池可以进行统一的分配、调优和监控
        - 线程池如何使用 —— Executors工具类快捷创建
          - Executors.newFixedThreadPool 固定线程数的线程池
          - Executors.newSingleThreadPool 单个线程的线程池
          - Executors.newCachedThreadPool 可动态扩展线程数的线程池
        - 线程池的几个重要参数
          - corePoolSize：线程池中核心常驻线程数
          - maximumPoolSize：线程池能够同时存在的最大线程数
          - keepAliveTime：多余的空闲线程的存活时间，线程池中线程数超过corePoolSize后，空闲时间达到keepAliveTime，多余的空闲线程就会被销毁，直到剩余线程数等于corePoolSize
          - unit：生存时间的时间单位TimeUnit
          - workQueue：任务队列，提交到线程池但还未被执行的任务
          - threadFactory：线程池中创建线程会调用的工厂对象，一般用默认的即可，可以实现自定义工厂，对创建出的线程做自定义参数设置
          - handler：拒绝策略，任务队列满了并且工作线程大于等于线程池最大线程数时，对后续提交的任务采取何种策略。
        - 说说线程池的底层工作原理
          - 1 当创建了线程池后，线程池等待任务被提交，初始池子中没有线程
          - 2.1 如果正在运行的线程数量小于corePoolSize，则线程池会立即创建出线程执行任务
          - 2.2 核心线程全被占用时，则后续提交的任务会被放入workQueue。
          - 2.3 如果队列满了，且正在运行的线程数量小于maximumPoolSize，那么后续提交的任务将被新创建出的非核心线程立即执行。
          - 2.4 如果workQueue满了，且正在运行的线程数大于等于maximumPoolSize，那么线程池会启动拒绝策略
          - 3 当一个线程完成任务时，它会从队列中取下一个任务执行
          - 4. 当一个线程空闲超过keepAliveTime时，线程池会判断：若当前运行线程数大于corePoolSize，那么这个空闲线程将会被销毁。
        - 线程池的拒绝策略
          - AbortPolicy：默认策略，直接抛出异常
          - CallerRunsPolicy：交由提交任务的线程直接执行
          - DiscardOldestPolicy：抛弃队列中等待最久的任务，并把当前任务加入到队列中
          - DiscardPolicy：直接丢弃任务未，不予以任何处理也不抛出异常
        - 如何合理的配置线程池参数
          - CPU密集型 —— 核心线程数设置为 cpu核心数+1
          - IO密集型 —— CPU核心数/1-阻塞系数 —— 阻塞系数一般为0.8~0.9
        - 死锁定位以及定位分析
          - 定义 —— 死锁产生的原因
            - 资源互斥：每个资源一次只能被一个进程（线程）持有
            - 请求与保持：进程（线程）因请求资源而阻塞时，对已获得的资源保持不放
            - 禁止强制剥夺：进程（线程）已获取的资源，无法被其他进程（线程）强制剥夺
            - 循环等待：两个或两个以上进程（线程）形成一种首尾相接的循环等待资源的关系
          - 代码Demo
  - JVM+GC解析
    - JVM内存结构
    - GC作用域 —— 方法区、堆
    - 常见的垃圾回收算法
    - JVM题目
      - 1. JVM垃圾回收的时候如何确定垃圾？哪些是Gc Roots?
        - 什么是JVM垃圾 —— 内存中已经不再被使用到的空间
        - 要进行垃圾回收，如何判断一个对象是否可以被回收？
          - 引用计数
          - 枚举根节点做可达性分析 —— 通过一系列名为“Gc Roots”的对象作为起始点，从这个被称为Gc Roots的对象开始向下搜索。如果一个对象没有到Gc Roots没有任何引用链，说明此对象不可用。 —— Java中可以作为Gc Roots的对象
            - 虚拟机栈（栈帧中的局部变量区，也叫局部变量表）中引用的对象
            - 方法区中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（native修饰的方法）引用的对象
      - 2. 说说接触过的JVM调优和参数配置，如何查看JVM参数的默认值?
      - 说说你平时工作中常用的JVM参数
      - 强引用、软引用、弱引用、虚引用分别是什么
      - 谈谈对OOM的理解
      - GC垃圾回收算法和垃圾回收器的关系
        - GC算法（引用计数/复制/标记清理/标记整理）是垃圾回收理论，垃圾回收器是算法的具体实现
        - 目前没有完美的垃圾回收器，只能针对具体应用选择最合适的垃圾回收器
        - 四种主要的垃圾回收器
          - 串行垃圾回收器（Serial） —— 为单线程环境设计且只使用一个线程进行垃圾回收，会暂停所有用户线程，不适合服务器环境
          - 并行垃圾回收器（Parallel） —— 多个垃圾回收线程并行工作，此时用户线程是暂停的，适用于科学计算、大数据处理等弱交互的场景
          - 并发垃圾回收器（CMS） —— 用户线程和垃圾回收线程同时执行（不一定并行，可能交替执行），不需要暂停用户线程。互联网公司项目多用它，适用对响应时间有要求的场景。
          - 上述三个收集器总结
            - STW：stop the world，暂停整个服务，时间可能会很长
            - CMS：垃圾回收线程与用户线程并行，不会暂停用户线程，但会抢占CPU资源
          - G1垃圾回收器 —— G1垃圾回收器将堆内存分割成不同的区域，然后并发的对各个区域进行垃圾回收
      - 怎么查看服务器默认的垃圾收集器
      - 生产上如何配置垃圾收集器
      - 谈谈你对垃圾收集器的理解
      - G1垃圾收集器
      - 生产环境服务器变慢，诊断思路和性能评估
        - 整机：top —— uptime，系统性能命令的精简版
        - CPU：vmstat
        - 内存：free
        - 硬盘：df
        - 磁盘IO：iostat
        - 网络IO：ifstat