一、HashMap线程安全性问题

1 多线程的put可能导致元素的丢失
同时put的覆盖。
2 put和get并发时，可能导致get为null
线程1执行put时，因为元素个数超出threshold而导致rehash，线程2此时执行get，有可能导致这个问题。
3 JDK7中HashMap并发put会造成循环链表，导致get时出现死循环
发生在多线程并发resize的情况下。

二、HashMap 与 ConcurrentHashMap 的实现原理是怎样的？ConcurrentHashMap 是如何保证线程安全的？

1 数组链表(1.7)红黑树(1.8) / 数组链表(1.7)红黑树(1.8)
2 不建议并发 / Segment(1.7)Node(1.8)

首先new一个新的hash表(nextTable)出来，大小是原来的2倍。后面的rehash都是针对这个新的hash表操作，不涉及原hash表(table)。
然后会对原hash表(table)中的每个链表进行rehash，此时会尝试获取头节点的锁。这一步就保证了在rehash的过程中不能对这个链表执行put操作。
3 通过sizeCtl控制，使扩容过程中不会new出多个新hash表来。
2 最后，将所有键值对重新rehash到新表(nextTable)中后，用nextTable将table替换。这就避免了HashMap中get和扩容并发时，可能get到null的问题。
在整个过程中，共享变量的存储和读取全部通过volatile或CAS的方式，保证了线程安全。
1 put的时候加锁。

三、volatile 关键字解决了什么问题，它的实现原理是什么？

可见性、有序性。

volatile可见性的实现就是借助了CPU的lock指令，通过在写volatile的机器指令前加上lock前缀，使写volatile具有以下两个原则：
写volatile时处理器会将缓存写回到主内存。
一个处理器的缓存写回到内存会导致其他处理器的缓存失效。

禁止指令重排序又是如何实现的呢？答案是加内存屏障。JMM为volatile加内存屏障有以下4种情况：
在每个volatile写操作的前面插入一个StoreStore屏障，防止写volatile与后面的写操作重排序。
在每个volatile写操作的后面插入一个StoreLoad屏障，防止写volatile与后面的读操作重排序。
在每个volatile读操作的后面插入一个LoadLoad屏障，防止读volatile与后面的读操作重排序。
在每个volatile读操作的后面插入一个LoadStore屏障，防止读volatile与后面的写操作重排序。

四、Synchronized 关键字底层是如何实现的？它与 Lock 相比优缺点分别是什么？

synchronized同步代码块的时候通过加入字节码monitorenter和monitorexit指令来实现monitor的获取和释放， 也就是需要JVM通过字节码显式的去获取和释放monitor实现同步，
而synchronized同步方法的时候，没有使用这两个指令， 而是检查方法的ACC_SYNCHRONIZED标志是否被设置，如果设置了则线程需要先去获取monitor，执行完毕了线程再释放monitor，也就是不需要JVM去显式的实现。

这两个同步方式实际都是通过获取monitor和释放monitor来实现同步的，而monitor的实现依赖于底层操作系统的mutex互斥原语， 而操作系统实现线程之间的切换的时候需要从用户态转到内核态，这个转成过程开销比较大。

1、原始构成
    JVM层面/api层面
2、使用方法
    手动释放锁/自动释放
3、等待是否可中断
    不可中断/可中断
        中断方法：设置超时方法tryLock()、代码块中放lockInterruptibly()，调用interrupt()。
4、加锁是否公平
    非公平/非公平，可实现公平
5、绑定多个条件Condition
    达咩/可用Condition精确唤醒

五、Java 中垃圾回收机制中如何判断对象需要回收？常见的 GC 回收算法有哪些？

引用计数算法：可能有循环引用的方法，故放弃
可达性分析算法：
可作为 GC Root 的对象包括以下4种：
1虚拟机栈（栈帧中的本地变量表）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 JNI（即一般说的 Native 方法）引用的对象

标记 --- 清除算法
复制算法
标记整理算法
分代收集算法：在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。
而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记-清理或者标记 --- 整理算法来进行回收

六、ThreadLocal key为什么设计成弱引用？

对于一个 ThreadLocal 对象，通常会有两个引用指向它：
一个是线程中声明的 threadLocal 变量，这是个强引用；
一个是线程底层 ThreadLocalMap 中键值对的 key，这是弱引用。
不再需要使用某 ThreadLocal 对象时，会采用将变量设置为 null（threadLocal = null）的方式释放掉线程中 threadLocal 变量与对象之间的引用关系，方便 GC 对 ThreadLocal 对象的回收。
但此时线程的 ThreadLocalMap 中还有 key 引用着这个 ThreadLocal 对象：如果这个引用是强引用，那么这个 ThreadLocal 对象就可能永远不会被回收了，造成内存泄露；
但现在这里设计成弱引用，那么当垃圾收集器发现这个 ThreadLocal 对象只有弱引用相关联时，就会回收它的内存。

七、String 类能不能被继承？为什么？

主要是为了“ 效率 ”和“ 安全性 ”的缘故。 若 String 允许被继承, 由于它的高度被使用率, 可能会降低程序的性能，所以 String 被定义成 final。

因为字符串是不可变的，所以是多线程安全的。
类加载器要用到字符串，不可变性提供了安全性。
只有当字符串是不可变的，字符串池才有可能实现。
因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键。

八、JMM 中内存模型是怎样的？什么是指令序列重排序？

堆、方法区。本地方法栈、虚拟机栈、程序计数器。

指令序列的重排序：
1）编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
2）指令级并行的重排序。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
3）内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

九、Java 线程和操作系统的线程是怎么对应的？Java线程是怎样进行调度的?（？？）

Linux 2.6上的HotSpot使用了NPTL机制，JVM线程跟内核轻量级进程有一一相应的关系
Java线程在Windows及Linux平台上的实现方式是内核线程的实现方式。
这样的方式实现的线程，是直接由操作系统内核支持的——由内核完毕线程切换，内核通过操纵调度器（Thread Scheduler）实现线程调度，并将线程任务反映到各个处理器上。
内核线程是内核的一个分身。程序一般不直接使用该内核线程，而是使用其高级接口，即轻量级进程（LWP）。也即线程。

十、简述 BIO, NIO, AIO 的区别？

bio :同步阻塞，服务器实现模式是一个连接一个线程，当客户端发来连接时服务器就需要启动一个线程进行处理，
　　如果这个连接不做任何事情就会造成不必要的线程开销，当然线程池机制可以改善。

nio:同步非阻塞，服务器实现模式为多个请求一个线程，即客户端发来的请求都会注册到多路复用器上，
　　多路复用器轮训的连接有io请求时才开启一个线程进行处理。

aio:异步非阻塞，服务器实现模式为多个有效请求一个线程。
　　即客户端发来的请求由os处理完成才会通知服务器应用启动线程进行处理。

十一、请你谈谈对OOM的认识

1 java.lang.StackOverflowError
栈空间溢出，递归调用卡死

2 java.lang.OutOfMemoryError:Java heap space
堆内存溢出，对象过大

3 java.lang.OutOfMemoryError:GC overhead limit exceeded
GC回收时间过长：超过98%的时间用来做GC并且回收了而不倒2%的堆内存。连续多次GC，cpu使用率一直是100%，而GC却没有任何成果。

4 java.lang.OutOfMemoryError:Direct buffer memory
本地内存挂了
写NIO程序经常使用ByteBuffer来读取或写入数据，这是一种基于通道（Channel）与缓存区（Buffer)的I/O方式，它可以使用Native函数库直接分配堆外内存，
然后通过一个存储在java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作，这样能在一

些场景中显著提高性能，因为避免了在java堆和native堆中来回复制数据

ByteBuffer.allocate(capability) 第一种方式是分配JVM堆内存，属于GC管辖，由于需要拷贝所以速度较慢

ByteBuffer.alloctedDirect(capability)分配os本地内存，不属于GC管辖，不需要拷贝，速度较快

但如果不断分配本地内存，堆内存很少使用，那么jvm就不需要执行GC，DirectByteBuffer对象们就不会被回收，这时候堆内存充足，但本地内存可能已经使用光了，再次尝试分配本地内存就会出现oom，程序崩溃

5 java.lang.OutOfMemoryError:unable to create new native thread

应用创建了太多线程，一个应用进程创建了多个线程，超过系统承载极限

你的服务器并不允许你的应用程序创建这么多线程，linux系统默认允许单个进程可以创建的线程数是1024，超过这个数量，就会报错

解决办法：

降低应用程序创建线程的数量，分析应用给是否针对需要这么多线程，如果不是，减到最低

修改linux服务器配置，扩大默认限制。

6 java.lang.OutOfMemoryError:Metaspace

元空间主要存放了虚拟机加载的类的信息、常量池、静态变量、即时编译后的代码

十二、GC垃圾回收算法和垃圾收集器的关系？分别是什么？※

垃圾收集器就是算法的具体实现

GC算法：引用计数、复制、标记清理、标记整理

4种主要垃圾收集器

Serial串行回收：为单线程换进该设计且只是用过一个线程进行垃圾回收，会暂停所有的用户线程，不适合服务器环境

Paralle并行回收：多个垃圾收集线程并行工作，此时用户线程是暂停的，适用于科学计算/大数据处理首台处理等弱交互场景

CMS并发标记清除：用户线程和垃圾手机线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线程，互联网公司多用它，适用堆响应时间有要求的场景

G1：将堆内存分割城不同的区域然后并发的对其进行垃圾回收

十三、怎么查看服务器默认的垃圾收集器是哪个？生产上如何配置垃圾收集器？对垃圾收集器的理解？

查看：java -XX:+PrintCommandLinedFlags -version（查看jvm默认的命令行参数）

配置：有六种：UseSerialGC UseParallelGC UseConcMarkSweepGC UseParNewGC UseParallelOldGC UseG1GC

收集器：

新生代：serial收集器（串行GC）、 parNew（新生代并行老年代串行）、 Parallel（并行GC）

老年代：CMS 、 Serial Old 、 Parallel Old

参数	新生代垃圾收集器	新生代算法	老年代垃圾收集器	老年代算法
UseSerialGC	SerialGC	复制	SerialOldGC	标整
UseParNewGC	ParNew	复制	SerialOldGC	标整
UseParallelGC UseParallelOldGC	Parallel[Scavenge]	复制	Parallel Old	标整
UseConcMarkSweepGC	ParNew	复制	CMS+Serial Old的收集器组合(Serial Old 作为CMS出错的后备收集器)	标清
UseG1GC	G1整体上采用标整	局部是通过复制算法		

十四、如何选择垃圾选择器？

- 单CPU或小内存，单机内存
  -XX:+UseSerialGC
- 多CPU，需要最大吞吐量，如后台计算型应用
  -XX:+UseParallelGC -XX:+UseParallelOldGC
- 多CPU，最求低停顿时间，需快速相应，如互联网应用
  -XX:+ParNewGC -XX:+UseConcMarkSweepGC

十五、G1和CMS相比优势

1. G1不会产生内存碎片
2. 可以精确控制停顿。该收集器是把整个堆划分成多个固定大小的区域，每根据允许停顿的时间去收集垃圾最多的区域

十六、生产环境服务器变慢，诊断思路和性能评估谈谈？

```
1. 整机：top 系统性能
    uptime 是它的精简版
    load average：系统负载均衡 1min 5min 15min 系统的平均负载值 相加/3>60%压力够
2. CPU：vmstat
    - 查看CPU
        vmstat -n 2 3 第一个参数是采样的时间间隔数单位s，第二个参数是采样的次数

    - procs
        **r**：运行和等待CPU时间片的进程数，原则上1核CPu的运行队列不要超过2，真个系统的运行队列不能超过总核数的2倍，否则表示系统压力过大
        **b**：等待资源的进程数，比如正在等待磁盘I/O，网络I/O等
    - cpu
```

```
        **us**：用户进程消耗cpu时间百分比，us高，用户进程消耗cpu时间多，如果长期大于50%，
优化程序
        **sy**：内核进程消耗的cpu时间百分比
        **us+sy**：参考值为80%，如果大于80，说明可能存在cpu不足
    - 查看额外
        - 查看所有cpu核信息    mpstat -P ALL 2
        - 每个进程使用cpu的用量分解信息    pidstat -u 1 -p 进程编号
3. 内存：free
    查看内存    free -m    free -g
    pidstat -p 进程编号 -r 采样间隔秒数
4. 硬盘：df
    查看磁盘剩余空间  df -h
5. 磁盘IO：iostat
    - 磁盘I/O性能评估
        iostat -hdk 2 3
        - util 一秒中又百分几的时间用于I/O操作，接近100%时，表示磁盘带宽跑满，需要优化程序
或加磁盘

    - pidstat -d 采样间隔秒数 -p 进程号
6. 网络IO：ifstat
   ifstat l
```

十七、假如生产环境CPU占用过高，谈谈分析思路和定位？

1. 先用top命令找出cpu占比最高的pid
2. ps -ef或者jps进一步定位，得知是一个怎样的后台程序惹事（可省）
    - jps -l
    - ps -ef|grep java|grep -v grep
3. 定位到具体线程或者代码

    ps -mp 进程编号 -o Thread,tid,time
    定位到具体线程
    -m ：显示所有的线程
    -p pid 进程使用cpu的时间
    -o：该参数后是用户自定义格式
4. 将需要的线程ID转换为16禁止格式（英文小写格式）
    - printf "%x\n" 线程ID
5. jstack 进程Id|grep tid(16进制线程id小写英文) -A60
   查看运行轨迹，堆栈异常信息

十八、LockSupport是什么？

LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。
总之，比wait/notify，await/signal更强。
1、无需要放在锁块中。
2、LockSupport可无视park、unpark的先后顺序。

3种让线程等待和唤醒的方法：
方式1：使用Object中的wait()方法让线程等待，使用object中的notify()方法唤醒线程
方式2：使用JUC包中Condition的await()方法让线程等待，使用signal()方法唤醒线程
方式3：LockSupport类可以阻塞当前线程以及唤醒指定被阻塞的线程

十九、LockSupport为什么唤醒两次后阻塞两次，但最终结果还会阻塞线程？

因为凭证permit的数量最多为1（不能累加），连续调用两次 unpark和调用一次 unpark效果一样，只会增加一个凭证；而调用两次park却需要消费两个凭证，证不够，不能放行。

二十、简述AQS

AQS使用一个volatile的int类型的成员变量来表示同步状态，通过内置的FIFo队列来完成资源获取的排队工作

将每条要去抢占资源的线程封装成一个Node，节点来实现锁的分配，通过CAS完成对State值的修改。