# Java并发编程面试150问

Q1：线程越多程序是否就运行得越快？

答：并发编程的目的是为了让程序运行得更快，但是并不是启动得线程越多就能让程序最大限度地并发执行。在并发编程时，如果希望通过多线程执行任务让程序运行得更快会面临很多挑战，比如上下文切换的问题、死锁的问题，以及受限于硬件和软件的资源限制问题。

Q2：多线程并发是怎么实现的，必须要用多核处理器实现吗？

答：即使是单核处理器也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短（一般是几十毫秒），所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的。

Q3：什么是上下文切换？

答:CPU是通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是在切换前会保存上一个任务的状态，以便下次再切换回这个任务时可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

Q4：如何减少上下文切换？

答：①无锁并发编程：多线程竞争锁时会引起上下文切换，所以多线程处理数据时，可以通过一些方法来避免使用锁，例如将数据的id按照hash算法取模分段，不同的线程处理不同数据段的数据。②CAS算法：Java的atomic包使用CAS算法来更新数据而不需要加锁。③使用最少线程：避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都处于等待状态。④协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

Q5：多线程避免死锁的方法？

答：①避免一个线程同时获得多个锁。②避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。③尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制。④对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的问题。

Q6：**volatile关键字的作用？**

答：①volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的可见性。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。②如果volatile变量修饰符使用恰当的话，它比synchronized的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。③如果一个字段被声明成volatile，Java线程内存模型确保所有线程看到这个变量的值是一样的。

Q7：**volatile的底层是如何实现的？**

答：有volatile修饰的共享变量在进行写操作时的汇编代码是具有lock前缀的指令，lock

前缀的指令在多核处理器下会引发两件事：①将当前处理器缓存行的数据写回到系统内存。②处理器将缓存回写到内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存后再进行操作，但操作完不知道何时会写回内存。如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条lock前缀的指令，将这个变量在缓存行的数据写回到系统内存。但是就算写回内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题（ABA问题）。所以在多处理器下，为了保证各个处理器的缓存是一致的就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置为无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

Q8：volatile如何优化性能？

答：可以通过追加字节的方式优化性能，例如JDK7中的队列集合类LinkedTransferQueue就是使用了追加字节的方式来优化队列出队和入队的性能。由于一些处理器的高速缓存行是64个字节宽，不支持部分填充缓存行，如果队列的头节点和尾节点都不足64字节，当一个处理器试图修改头节点时就会将整个缓存行锁定，那么在缓存一致性的作用下会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队又会频繁修改头节点和尾节点，因此多处理器情况下会严重影响队列的入队和出队效率。追加到64字节后就可以填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行，使它们的操作不会互相锁定。

但以下两种场景不应该使用这种方式：①缓存行非64字节宽的处理器。②共享变量不会被频繁地写，因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定性能消耗。如果共享变量不被频繁写，锁的几率很小没有必要避免互相锁定。不过这种追加字节的方式在Java7可能不生效，因为Java7可以淘汰或重新排列无用字段，需要使用其他追加字节的方式。

Q9：synchronized锁的形式有哪些？

答：①对于同步普通方法，锁是当前实例对象。②对于静态同步方法，锁是当前类的Class对象。③对于同步方法块，锁是synchronized括号里配置的对象。

Q10：**synchronized的底层是怎么实现的？**

答：JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。代码块同步是使用monitorenter和monitorexit指令实现的，而方法同步是使用另一种方式实现的，细节并未在JVM规范中详细说明，但是方法的同步也可以使用这两个指令来实现。

monitorenter指令是编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有monitorexit与之配对。任何对象都有一个monitor与之关联，当一个monitor被持有后它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁。

Q11：**什么是锁升级（锁优化）？**

答：JDK1.6为了减少获得锁和释放锁带来的性能消耗，引入了偏向锁和轻量级锁，在JDK1.6中，锁一共有4个状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，如果偏向锁升级成轻量级锁后就不能降级成偏向锁，这种只能升级不能降级的锁策略是为了提高获得锁和释放锁的效率。

Q12：**偏向锁的获得原理是什么？**

答：大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步代码块并获取锁时，会在对象头和帧栈中的锁记录里存储锁偏向的线程ID，以后该线程再进入和退出同步代码块不需要进行CAS操作来加锁和解锁，只需要简单地测试一下对象头（synchronized用的锁存在Java的对象头里）的Mark Word里是否存储着指向当前线程的偏向锁。

如果测试成功表示线程已经获得了锁，如果测试失败则需要再测试一下Mark Word（主要存储锁状态、对象的hashCode、对象的分代年龄、是否是偏向锁、锁标志位）中偏向锁的标识是否设置成了1（表示当前是偏向锁），如果设置了就尝试使用CAS将对象头的偏向锁指向当前线程，否则使用CAS竞争锁。

Q13：**偏向锁的撤销原理是什么？**

答：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（该时间点上没有正在执行的字节码），它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态则将对象头设为无锁状态；如果线程还活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。

Q14：偏向锁的打开和关闭是怎么实现的？

答：偏向锁在Java6和Java7中默认是开启的，但是它在应用程序启动几秒后才激活，如果有必要可以使用JVM参数来关闭延迟：-XX：BiasedLockingStartupDelay=0。如果你确定应用程序里所有的锁通常情况处于竞争状态，可以通过JVM参数来关闭偏向锁：-XX：UseBiasedLocking=false，那么程序默认会进入轻量级锁状态。

Q15：**轻量级锁的加锁原理是什么？**

答:线程在执行同步块之前，JVM会先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针，如果成功那么当前线程获得锁，如果失败表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

Q16：**轻量级锁的解锁原理是什么？**

答：轻量级锁解锁时，会使用原子的CAS操作将Displaced Mark Word替换回到对象头，如果成功则表示没有竞争发生。如果失败则表示当前存在锁竞争，锁就会膨胀为重量级锁。

因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞了），一旦锁升级为重量级锁，就不会再恢复到轻量级锁的状态。在这种情况下，其他线程视图获取锁时都会被阻塞，当持有锁的线程释放锁后才会唤醒这些线程，被唤醒的线程就会对锁资源进行新一轮的争夺。

Q17：**偏向锁、轻量级锁和重量级锁的区别？**

答：①偏向锁的优点是加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距，缺点是如果线程间存在锁竞争会带来额外锁撤销的消耗，适用于只有一个线程访问同步代码块的场景。②轻量级锁的优点是竞争的线程不会阻塞，提高了程序的响应速度，缺点是如果线程始终得不到锁会自旋消耗CPU，适用于追求响应时间和同步代码块执行非常快的场景。③重量级锁的优点是线程竞争不使用自旋不会消耗CPU，缺点是线程会被阻塞，响应时间很慢，适应于追求吞吐量，同步代码块执行较慢的场景。

Q18：**原子操作是什么，处理器是怎么实现原子操作的？**

答:原子操作即不可被中断的一个或一系列操作，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

①通过总线锁定保证原子性：如果多个处理器同时对共享变量进行读改写操作（例如i++），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子性的，操作完之后共享变量的值会和期望的不一样。例如i=1，进行两次i++操作，但是结果可能为2。这是因为多个处理器同时从各自的缓存读取变量i，分别进行加1操作，然后分别写入系统内存中。如果想要保证读改写操作的原子性，就必须保证CPU1读改写共享变量时CPU2不能操作缓存了该共享变量内存地址的缓存。处理器使用总线锁来解决这个问题，总线锁就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞，该处理器就可以独占共享内存。

②通过缓存锁定来保证原子性：同一时刻只需要保存对某个内存地址的访问是原子性即可，但总线锁定把CPU和内存之间的通信锁住了，这使得锁定期间其他处理器不能操作其他内存地址的数据，开销比较大，目前的处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。频繁使用的内存会缓存在处理器的高速缓存里，原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁。缓存锁定是指内存区域如果被缓存在处理器的缓存行中并且在Lock操作期间被锁定，那么当它执行锁操作回写内存时，处理器不在总线上声言LOCK#信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作原子性，因为缓存一致性会阻止同时修改由两个以上处理器缓存的内存区域，当其他处理器回写已被锁定的缓存行数据时会使缓存行无效。

Q19：不会使用缓存锁定的情况有哪些？

答：①当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，处理器会调用总线锁定。②有些处理器不支持缓存锁定，例如Intel486和Pentium处理器，即使锁定的内存区域在处理器的缓存行中也会调用总线锁定。

Q20：**缓存行和CAS是什么？**

答：①缓存行：缓存的最小操作单位。②CAS：Compare and Swap，比较并交换，CAS需要两个数值，一个是旧值（期望操作前的值）和一个新值，在操作期间先比较旧值有没有发生变化，如果没有发生变化才交换成新值，发生了变化则不交换。

Q21：**Java中如何实现原子操作？**

答：Java中可以通过锁和循环CAS的方式来实现原子操作。

锁机制保证了只有获得锁的线程才能操作锁定的内存区域，JVM内部实现了很多锁，除了偏向锁JVM实现锁的方式都用了循环CAS，即当一个线程想进入同步代码块时使用循环CAS方式获取锁，退出时使用循环CAS释放锁。

JVM中的CAS操作利用了处理器提供的交换指令CMPXCHG实现，自旋CAS的基本思路就是循环进行CAS操作直到成功为止。从Java1.5开始JDK的并发包里提供了一些类来支持原子操作，例如AtomicBoolean（用原子方式更新的boolean值），AtomicInteger（用原子方式更新的int值）和AtomicLong（用原子方式更新的long值），这些原子包装类还提供了有用的工具方法，比如以原子的方式将当前值自增1和自减1。

Q22：**CAS实现原子操作有什么问题？**

答：①ABA问题：因为CAS需要在操作值的时候检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成B，又变成了A，那么使用CAS检查时会发现它的值并未发生变化。ABA问题的解决思路就是使用版本号，在变量前面追加版本号，每次更新时把版本号加1，那么A->B->A就会i变成1A->2B->3A。从Java1.5开始，JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题，这个类的compareAndSet方法首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和标志的值设置为给定的更新值。

②循环时间长，开销大：自旋CAS如果长时间不成功会给CPU带来非常大的执行开销。如果JVM支持处理器提供的pause指令，就可以提升效率，因为pause指令可以延迟流水线执行指令，使CPU不会消耗过多的执行资源，延迟的时间取决于具体版本；避免在退出循环时因为内存顺序冲突而引起CPU流水线被清空，从而提高CPU的执行效率。

③只能保证一个共享变量的原子操作：循环CAS不能保证多个共享变量操作的原子性，可以使用锁或把多个共享变量合成为一个，放在一个对象里进行CAS操作。

Q23：**Java中线程是如何通信和同步的？**

答：通信是指线程之间以何种机制来交换信息，在命令式编程中线程之间的通信机制有两种，共享内存和消息传递。在共享内存的并发模型里线程之间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信。在消息传递的并发模型里线程之间没有公共状态，线程之间必须通过发送消息来显示通信。

同步是指程序中用于控制不同线程间操作发生相对顺序的机制，在共享内存的并发模型里同步是显示进行的，程序员必须显示指定某个方法或代码需要在线程之间互斥执行。在消息传递的并发模型里，由于消息的发送必须在接受之前，同步是隐式进行的。

Java并发采用共享内存模型，线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。

Q24：哪些数据会存在内存可见性问题？

答：在Java中，所有实例域、静态域和数组元素都存储在堆中，堆内存在线程之间共享，因此这些共享变量存在内存可见性问题。局部变量、方法定义参数和异常处理器参数等不会在线程之间共享，不会存在内存可见性问题，也不受内存模型的影响。

Q25：**JMM是什么？**

答：JMM（Java Memory Model）是Java内存模型，Java线程之间的通信由JMM控制，JMM决定了一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，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。 两个线程之间的通信必须经过主内存，JMM通过控制主内存与每个线程的本地内存之间的交互来为Java程序员提供内存可见性保证。

Q26：**指令重排序是什么？**

答：重排序指从源代码到指令序列的重排序，在执行程序时为了提高性能，编译器和处理器通常会对指令进行重排序，重排序分为三种类型。

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：现代处理器才以来指令级并行技术ILP来将多条指令重叠执行，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操作看上去可能是乱序执行。

Q27：**指令重排序的问题及解决？**

从Java源代码到最终实际执行的指令序列，会分别经历编译器优化重排序、指令级并行重排序和内存系统重排序。这些重排序可能会导致多线程程序出现内存可见性问题。对于编译器，JMM的编译器重排序规则会禁止特定类型的编译器重排序。对于处理器重排序，JMM的处理器重排序规则会要求Java编译器在生成指令序列时，插入特定类型的内存屏障指令（一组用于实现对内存操作顺序限制的处理器指令），通过内存屏障指令来禁止特定类型的处理器重排序。

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的可见性内存保证。

Q28：JMM内存屏障指令的分类有哪些？

答：①Load Load，确保Load1的数据装载先于Load2及所有后续装载指令的装载。②Store Store，确保Store1数据对其他处理器可见（刷新到内存）先于Store2及所有后续存储指令的存储。③Load Store，确保Load1数据装载先于Store2及所有后续存储指令刷新到内存。④Store Load，确保Store1数据对其他处理器变得可见（刷新到内存）先于Load2及所有后续装载指令的装载。Store Load会使该屏障之间的所有内存访问指令（存储和装载指令）完成之后才执行该屏障之后的内存访问指令。该指令是一个“全能型”屏障，同时具备其他三个屏障的效果，现代的多处理器大多支持该屏障，执行该屏障的开销很昂贵，因为当前处理器通常要把写缓冲区的数据全部刷新到内存中。

Q29：**happens-before是什么？**

答：从JDK5开始，Java使用新的JSR-133内存模型，JSR-133使用happens-before的概念来阐述操作之间的内存可见性。在JMM中如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系，这两个操作既可以是在一个线程之内，也可以是不同线程之内。

与程序员密切相关的happens-before规则如下：①程序顺序规则：一个线程中的每个操作，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。③volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。④传递性，如果A happens-before B，且B happens-before C，那么A happens-before C。⑤start()规则，如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。⑥join()规则，如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

注意：两个操作之间具有happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行。happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。

Q30：什么是数据依赖性？

答：如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时两个操作之间就存在数据依赖性。数据以来分为以下三种类型：①写后读，写一个变量之后再读这个位置。②读后写，读一个变了之后再写这个变量。③写后写，写一个变量之后再写这个变量。

上述三种情况只要重排序两个操作的执行顺序，程序的执行结果就会被改变。编译器和处理器为了性能优化可能会对操作重排序，在重排序时会遵守数据依赖性，不会改变存在数据依赖关系的两个操作的执行顺序。这里说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

Q31：as-if-serial语义是什么？

答：as-if-serial指不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器、runtime和处理器都必须遵循该语义。为了遵循该语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但如果操作之间不存在数据依赖关系，这些操作就可能被重排序。

as-if-serial语义将单线程保护了起来，遵循as-if-serial语义的编译器、runtime和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按照程序的顺序执行的。as-if-serial使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。

as-if-serial实例：例如计算一个圆的面积，A操作给半径赋值，B操作给圆周率赋值，C操作计算圆的面积。由于C依赖于A和B因此不会被重排到A和B的前面，但A和B之间没有数据依赖关系，所以程序的执行顺序可以是ABC或BAC，结果是一样的。

Q32：控制依赖关系对指令重排序的影响？

答：当代码中存在控制依赖性时（例如A操作判断某标志位，B操作根据A的结果执行对应逻辑），会影响指令序列执行的并行度。为此编译器和处理会采用猜测执行来克服控制相关性对并行度的影响，可以提前计算出值保存到名为重排序缓冲的硬件缓存中，如果之前的控制条件满足就执行对应操作。

在单线程程序中，对存在控制依赖的操作重排序并不会改变程序的执行结果（这也是as-if-serial语义允许对存在控制依赖的操作做重排序的原因）。但在多线程程序中，对存在控制依赖的操作从排序可能会改变程序的执行结果。

Q33：数据竞争和顺序一致性是什么？

答：当程序未正确同步时就可能存在数据竞争。JMM规范对数据竞争的定义如下：在一个线程中写一个变量，在另一个线程中读同一个变量，而且写和读没有通过同步来排序。当代码中包含数据竞争时程序的执行往往产生违反直觉的结果，如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

JMM对正确同步的多线程程序的内存一致性做了如下保证：如果程序是正确同步的，程序的执行将具有顺序一致性，即程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同。这里的同步是指广义上的同步，包括对常用同步用语（synchronized、volatile和final）的正确使用。

Q34：顺序一致性内存模型的特点？

答：顺序一致性内存模型是一个理想化的理论参考模型，它为程序员提供了极强的内存可见性保证。顺序一致性内存模型有两大特性：①一个线程中的所有操作必须按照程序的顺序来执行。②不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序，在顺序一致性的内存模型中，每个操作都必须原子执行并且立即对所有线程可见。

在概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关可以连接到任意一个线程，同时每一个线程必须按照程序的顺序来执行内存读/写操作。在任意时间点最多只能有一个线程可以连接到内存，当多个线程并发执行时，开关装置能把线程的所有内存读/写操作串行化（即在顺序一致性模型中所有操作之间具有全序关系）。

Q35：未同步程序在JMM中的问题？

答：未同步程序在顺序一致性模型中虽然整体执行顺序无序但是所有线程都能看到一个一致的整体执行顺序。之所以能得到这个保证是因为顺序一致性内存模型中的每个操作必须立即对任意线程可见。

JMM中没有这个保证，未同步程序在JMM中不但整体的执行顺序无序，并且所有线程看到的操作执行顺序也可能不一致。比如当前线程把写过的数据缓存到本地内存，在没有刷新到主内存前，这个写操作仅对当前线程可见。从其他线程的角度会认为这个写操作并没有执行，只有当前线程把本地内存中写过的数据刷新回主内存之后这个写操作才对其他线程可见，这种情况下当前线程和其他线程看到的操作执行顺序不一致。

Q36：未同步程序的执行特性？

答：对于未同步或未正确同步的多线程程序，JMM只提供最小安全性：线程执行时读取到的值要么是之前某个线程写入的值，要么是默认值，JMM保证线程读操作读取到的值不会无中生有。为了实现最小安全性，JVM在堆上分配对象时首先会对内存空间进行清零，然后才会在上面分配对象（JVM内部同步这两个操作）。因此在已清零的内存空间分配对象时，域的默认初始化已经完成了。

JMM不保证未同步程序的执行结果与该程序在顺序一致性模型的执行结果一致，因为如果想要保证一致需要禁止大量的处理器和编译器优化，这对程序执行性能会有很大影响。而且未同步程序在顺序一致性模型中执行时整体是无序的，结果无法预知，因此保证未同步执行程序在两个模型的执行结果一致没什么意义。

Q37：未同步程序在JMM和顺序一致性模型的执行区别？

答：①顺序一致性模型保证单线程内的操作会按程序的顺序执行，而JMM不保证单线程内的操作会按程序的顺序执行。

②顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而JMM不保证所有线程能看到一致的操作执行顺序。

③JMM不保证对64位的long类型和double类型变量的写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。

Q38：总线的工作机制是什么？

答：在计算机中，数据通过总线在处理器和内存之间传递。每次处理器和内存之间的数据传递都是通过一系列步骤来完成的，这一系列步骤称为总线事务。总线事务包括读事务和写事务。读事务从内存中传输数据到处理器，写事务从处理器传送数据到内存，每个事务会读/写内存中的一个或多个物理上连续的字，总线会同步试图并发使用总线的事务。在一个处理器执行总线事务期间，总线会禁止其他的处理器和IO设备执行内存的读/写。

Q39：总线工作机制的好处？

答：总线的工作机制可以把所有处理器对内存的访问以串行化的方式来执行，在任意时间点最多只能有一个处理器访问内存，这个特性确保了单个总线事务之中的内存读/写操作具有原子性。

Q40：long和double变量的原子性问题？

答：在一些32位的处理器上如果要求对64位数据的写操作具有原子性，会有比较大的开销。为了照顾这种处理器，Java语义规范鼓励但不强求JVM对64位的long和double类型变量的写操作具有原子性，当JVM在这种处理器上运行时可能会把一个64位的long/double变量写操作拆分为两个32位的写操作执行，这两个32位的写操作可能会被分配到不同的总线事务中执行，此时对这个64位的写操作不具有原子性。

在JSR-133之前的旧内存模型允许把一个64位的double/long变量的读/写操作拆分位两个64位的读/写操作执行。从JSR-133内存模型（JDK5）开始，仅仅只允许把一个64位的long/double类型变量写操作拆分为两个32位的写操作来执行，任意的读操作在JSR-133中必须具有原子性（即任意读操作必须要在单个读事务中执行）。

Q41：**volatile变量的特性？**

答：①可见性：对任意一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。②原子性：对任意单个volatile变量的读/写具有原子性，但类似于i++这种复合操作不具有原子性。

Q42：**volatile变量的内存语义？**

答：从JSR-133开始，volatile变量的写-读可以实现线程之间的通信。从内存语义的角度来说，volatile的写-读与锁的释放-获取具有相同的内存效果。

volatile写的内存语义如下：当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。volatile读的内存语义如下：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。

线程A写一个volatile变量，实质上是线程A向接下来要读这个volatile变量的某个线程发出了（其对共享变量所修改的）消息。线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。线程A写一个volatile变量，线程B读一个volatile变量，实质上是线程A通过主内存向线程B发送消息。

Q43：**volatile指令重排序的特点？**

答：①当第二个操作是volatile写时，不管第一个操作是什么都不能重排序，这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。

②当第一个操作是volatile读时，不管第二个操作是什么都不能重排序，这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。

③当第一个操作是volatile写，第二个操作是volatile读时不能重排序。

Q44：**volatile内存语义是怎么实现的？**

答：JMM通过分别限制编译器重排序和处理器重排序来实现volatile的内存语义。编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此JMM采取保守策略。

Q45：JMM内存屏障插入策略有哪些？

答：①在每个volatile写操作之前插入一个Store Store屏障,禁止之前的普通写和之后的volatile写重排序。

②在每个volatile写操作之后插入一个Store Load屏障，防止之前的volatile写与之后可能有的volatile读/写重排序，也可以在每个volatile变量读之前插入该屏障，考虑到一般是读多于写所以选择用这种方式提升执行效率，也可以看出JMM在实现上的一个特点：首先确保正确性，然后再去追求效率。

③在每个volatile读操作之后插入一个Load Load屏障，禁止之后的普通读操作和之前的volatile读重排序。

④在每个volatile读操作之后插入一个Load Store屏障，禁止之后的普通写操作和之前的volatile读重排序。

Q46：**JSR-133增强volatile内存语义的原因？**

答：在旧的内存模型中，虽然不允许volatile变量之间重排序，但允许volatile变量与普通变量重排序，可能导致内存不可见问题。在旧的内存模型中volatile的写-读没有锁的释放-获取所具有的内存语义，为了提供一种比锁更轻量级的线程通信机制，严格限制了编译器和处理器对volatile变量与普通变量的重排序，确保volatile的写-读和锁的释放-获取具有相同的内存语义。只要volatile变量与普通变量之间的重排序可能会破坏volatile的内存语义这种重排序就会被编译器重排序规则和处理器内存屏障插入策略禁止。

Q47：**锁的内存语义？**

答：当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。当线程获取锁时，JMM会把线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须从主内存中读取共享变量。

锁的释放与volatile写具有相同的内存语义，锁获取与volatile读具有相同的内存语义。线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。线程A释放这个锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

Q48：**锁的内存语义是怎么实现的？**

答：公平锁和非公平锁释放时，最后都要写一个volatile变量state。公平锁获取锁时，首先会去读volatile变量，非公平锁获取锁时，首先会用CAS更新volatile变量的值，这个操作同时具有volatile读和volatile写的内存语义。因此锁的释放-获取内存语义的实现方式为：①利用volatile变量的写-读具有的内存语义。②利用CAS所附带的volatile读和volatile写的内存语义。

Q49：**Java中concurrent包的原子性是如何保证的？**

答：由于Java的CAS同时具有volatile读和volatile写的内存语义，因此Java线程通信有以下四种方式：①A线程写volatile变量，随后B线程读这个volatile变量。②A线程写volatile变量，随后B线程用CAS更新该变量。③A线程用CAS更新一个volatile变量，随后B线程用CAS更新该变量。④A线程用CAS更新一个volatile变量，随后B线程读这个变量。

Java的CAS会使用现代处理器上提供的高效机器级别的原子指令，这些原子指令以原子方式对内存进行读-改-写操作。同时volatile变量的读/写和CAS可以实现线程之间的通信，这些特性就是concurrent包的基石。concurrent包有一个通用的实现模式：首先声明共享变量为volatile，然后使用CAS的原子条件更新来实现线程之间的同步，同时配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。包括AQS，非阻塞数据结构和原子变量类这些基础类都是通过这种模式来实现的，而concurrent包中的高层类又是依赖这些基础类来实现的。

Q50：**final域的重排序规则？**

答：对于final域，编译器和处理器要遵守两个重排序规则：①在构造方法内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。②初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

Q51：**写final域重排序的实现原理？**

答：写final域的重排序规则禁止把final域的写重排序到构造方法之外，这个规则的实现包含以下两方面：①JMM禁止编译器把final域的写重排序到构造方法之外。②编译器会在final域的写之后，构造方法的return之前，插入一个Store Store屏障，这个屏障禁止把final域的写重排序到构造方法之外。

写final域的重排序可以确保在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域不具有这个保障。

Q52：**读final域重排序的实现原理？**

答：读final域的重排序规则是，在一个线程中，初次读对象引用和初次读该对象包含的final域，JMM禁止处理器重排序这两个操作（仅针对处理器）。编译器会在读final域操作的前面插入一个Load Load屏障。

初次读对象引用与初次读该对象包含的final域，这两个操作之间存在间接依赖关系。由于编译器遵守间接依赖关系因此编译器不会重排序这两个操作。大多数处理器也会遵守间接依赖，也不会重排序这两个操作。但有少数处理器允许对存在间接依赖关系的操作做重排序（例如alpha处理器），因此该规则就是专门针对这种处理器的。

读final域的重排序规则可以确保在读一个对象的final域之前，一定会先读包含这个final域的对象的引用。

Q53：**final域为引用对象时重排序的特点？**

答：对于引用类型，写final域的重排序规则对编译器和处理器增加了如下约束：在构造方法内对一个final引用的对象的成员域的写入，与随后在构造方法外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

Q54：**final引用的可见性问题？**

答：写final域的排序规则可以确保在对象引用为任意线程可见之前，该引用变量指向对象的final域已经在构造方法中被正确初始化过了。其实要实现这个保证还需要在构造方法内部，不能让这个被构造对象的引用为其他线程所见，也就是对象引用不能在构造方法中逸出。在构造方法返回前，被构造对象的引用不能为其他线程所见，因为此时的final域可能还没有被正确地初始化。在构造方法返回后，任意线程都将保证能看到final域正确初始化后地值。

Q55：final语义在X86处理器的实现原理是什么？

答：写final域的重排序规则是要求编译器在final域的写之后，构造方法return之前插入一个Store Store屏障，读final域的重排序规则是要求编译器在读final域的操作前插入一个Load Load屏障。

由于X86处理器不会对写-写操作重排序，所以写final域需要的Store Store屏障会被省略。同样，由于X86处理器不会对存在间接依赖关系的操作做重排序，所以在X86处理器中读final域需要的Load Load屏障也会被省略掉。也就是说，X86处理器不会对final域的读/写插入任何内存屏障。

Q56：**JSR-133增强final语义的原因？**

答：在旧的Java内存模型中，一个最严重的缺陷就是线程可能看到final域的值会改变。比如一个线程看到一个int类型final域的值为0（还未初始化之前的默认值），过一段时间之后这个线程再去读这个final域的值会发现值变为1（被某个线程初始化之后的值）。最常见的例子就是旧的Java内存模型中String的值可能会改变。

为了修复该漏洞，JSR-133通过为final域增加写和读重排序规则，可以为Java程序员提供初始化安全保证：只要对象是正确构造的（被构造对象的引用在构造方法中没有逸出），那么不需要使用同步（指lock和volatile的使用）就可以保证任意线程都能看到这个final域在构造方法中被初始化之后的值。

Q57：**happens-before的重排序策略？**

答：JMM将happens-before要求禁止的重排序分为了下面两类：会改变程序执行结果的重排序和不会改变程序执行结果的重排序。JMM对这两种不同性质的重排序采取了不同的策略，对于会改变程序执行结果的重排序JMM要求编译器和处理器必须禁止这种重排序；对于不会改变程序执行结果的重排序，JMM对编译器和处理器不做要求（JMM允许这种重排序）。

JMM向程序员提供happens-before规则能满足程序员的需求，其规则不但简单易懂而且也向程序员提供了足够强的内存可见性保证（有些内存保证性不一定真实存在，例如不改变执行结果的指令重排序对程序员是透明的）。

JMM对编译器和处理器的束缚已经尽可能地少，JMM遵循一个基本原则：只要不改变程序执行结果，编译器和处理器怎么优化都行。例如编译器分析某个锁只会单线程访问就消除该锁，某个volatile变量只会单线程访问就把它当作普通变量。

Q58：**happens-before的具体定义是什么？**

答：①如果一个操作happens-before另一个操作，那么第一个操作的执行结果对第二个操作可见，并且第一个操作的执行顺序排在第二个操作之前。这是JMM对程序员的承诺。

②两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序执行，如果重排序之后的执行结果与按照happens-before关系的执行结果一致，那么这种重排序是可以允许的。这是JMM对编译器和处理器的约束规则，JMM遵循一个基本原则：只要不改变程序执行结果，编译器和处理器怎么优化都行。JMM这么做的原因是程序员对于这两个操作是否真的被重排序并不关心，程序员关心的是程序执行的语义不能被改变（即执行结果不能被改变）。因此happens-before关系的本质和as-if-serial一样。

Q59：happens-before和as-if-serial的区别？

答：as-if-serial语义保证单线程程序的执行结果不被改变，happens-before保证正确同步的多线程程序的执行结果不被改变。as-if-serial语义给编写单线程程序的程序员创造了一种单线程程序是顺序执行的幻觉，happens-before关系给编写正确同步的多线程程序员创造了一种多线程程序是按照happens-before指定顺序执行的幻觉。这两种语义的目的都是为了在不改变程序执行结果的前提下尽可能提高程序执行的并行度。

Q60：happens-before规则的相关实现原理？

答：①程序顺序规则：编译器和处理器都要遵守as-if-serial语义，as-if-serial语义保证了程序顺序执行规则。②volatile规则：对一个volatile变量的读总是能看到（任意线程）之前对这个volatile变量最后的写入，因此volatile的这个特性可以保证实现volatile规则。③传递性规则：由volatile的内存屏障插入策略和volatile的编译器重排序规则共同保证。

Q61：处理器内存模型的分类？

答：①放松程序中写-读操作的顺序，由此产生了TSO内存模型。②在TSO的基础上继续放松程序中写-写操作的顺序，由此产生了PSO内存模型。③在TSO和PSO的基础上，继续放松程序中读-写（以两个操作之间不存在数据依赖性为前提）和读-读操作的顺序，由此产生了RMO和PowerPC内存模型。

Q62：JMM对不同处理器模型的处理？

答：不同的处理器模型，性能越好，内存模型的设计越弱，因为处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。由于常见的处理器内存模型比JMM要弱，Java编译器在生成字节码时，会在执行指令序列的适当位置插入内存屏障来限制处理器的重排序。同时由于各种处理器内存模型的强弱不同，为了在不同的处理器平台向程序员展示一个一致的内存模型，JMM在不同的处理器中需要插入的内存屏障的数量和种类也不同。JMM屏蔽了不同处理器内存模型的差异，它在不同的处理器平台之上为Java程序员呈现了一个一致的内存模型。

Q63:Java程序内存可见性保证的分类？

答：①单线程程序：单线程程序不会出现内存可见性问题。编译器、runtime和处理器会共同确保单线程程序的执行结果与该程序在一致性模型中的执行结果相同。

②正确同步的多线程程序：正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在一致性模型中的执行结果相同）。这是JMM关注的重点，JMM通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。

③未同步/未正确同步的多线程程序：JMM为它们提供了最小安全性保证，线程执行读取到的值要么是之前某个线程写入的值，要么是默认值，但不保证该值是正确的。

Q64：JSR-133对旧内存模型的修补有什么？

答：①增强volatile的内存语义，旧内存模型允许volatile变量与普通变量重排序。JSR-133严格限制volatile变量与普通变量的重排序，使volatile的写-读和锁的释放-获取具有相同的内存语义。

②增强final的内存语义，旧内存模型中多次读取同一个final变量的值可能会不相同，为此JSR-133为final增加了两个重排序规则。在保证final引用不会从构造方法逸出的情况下，final具有了初始化安全性。

Q64：**什么是线程？**

答：现代操作系统在运行一个程序时会为其创建一个进程，而操作系统调度的最小单位是线程，线程也叫轻量级进程。在一个进程中可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上告诉切换，让使用者感觉到这些线程在同时执行。

Q65：为什么要使用多线程？

答：①可以更好地利用多处理器核心。线程是大多数操作系统调度的基本单位，一个程序作为一个进程来运行，程序运行过程中能创建多个线程，而一个线程一个时刻只能运行在一个处理器核心上。单线程最多使用一个处理器核心，加入再多的处理器核心也无法提升程序的执行效率，多线程技术将计算逻辑分配到多个处理器核心，显著提升程序执行效率。

②可以获得更快的响应时间。在一些复杂的业务逻辑中，可以使用多线程技术，将数据一致性不强的操作派发给其他线程处理，可以缩短响应时间，提升用户体验。

③Java为程序员提供了良好的一致的编程模型，使开发者可以更加专注于问题的解决，而不是思考如何使其多线程化。

Q66：什么是线程优先级？

答：现代操作系统基本采用时分形式调度运行的线程，操作系统会分出一个个时间片，线程会分配到若干时间片，当线程的时间片用完了就会发生线程调度，并等待下次分配。线程分配到的时间片多少也就决定了线程使用处理器资源的多少，而线程优先级就是决定线程需要多或者少分配一些处理器资源的线程属性。

Q67：如何设置线程优先级？

答：在Java中通过一个整形成员变量priority来控制线程优先级，优先级的范围从1~10，在线程构建的时候可以通过setPriority(int)方法来修改优先级，默认优先级是5,优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者IO操作）的线程需要设置较高优先级，而偏重计算（需要较多CPU时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。在不同的JVM以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

Q68：线程有哪些状态？

答：①NEW：初始状态，线程被构建，但还没有调用start()方法。②RUNNABLE：运行状态，Java线程将操作系统中的就绪和运行两种状态统称为运行中。③BLOCKED：阻塞状态，表示线程阻塞于锁。④WAITING：等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）。⑤TIME\_WAITING：超时等待状态，该状态不同于WAITING，可以在指定时间内自行返回。⑥TERMINATED：终止状态，表示当前线程已经执行完毕。

Q69：什么是daemon线程？

答：daemon线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作，这意味着当一个Java虚拟机中不存在非daemon线程的时候，Java虚拟机将会退出，可以通过Thread.setDaemon(true)将线程设置为daemon线程（需要在线程启动之前设置）。

daemon线程被用于完成支持性工作，但是在JVM退出时daemon线程中的finally块并不一定会被执行，因为当JVM中已经没有非daemon线程时JVM需要立即退出，所有daemon线程都需要立即终止。因此不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

Q70：线程的中断是什么？

答：中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。其他线程通过调用该线程的interrupt()方法对其进行中断操作。

线程通过检查自身是否被中断来进行响应，线程通过方法isInterrupted()来判断是否被中断，也可以调用静态方法Thread.interrupted()对当前线程的中断标识位进行复位。如果该线程处于终结状态，即使该线程被中毒过，在调用该对象的isInterrupted()时依然返回false。

许多声明抛出InterruptedException的方法（例如Thread.sleep(long mills))在抛出异常之前，JVM会将该线程的中断标识位清除，然后再抛出异常，此时调用isInterrupted()时将会返回false。

Q71：为什么suspend、resume、stop方法被废弃了？

答：以suspend方法为例，在调用后线程不会释放已经占有的资源（比如锁），而是占着资源进入睡眠状态，这样容易引发死锁问题。同样，stop方法在终结一个线程时不会保证线程的资源正常释放，通常是没有基于线程完成资源释放工作的机会，因此会导致程序可能运行在不确定状态下。因为这些方法的副作用因而被标注为不建议使用的废弃方法，而暂停/恢复机制可以用等待/唤醒机制代替。

Q72：**可见性问题的原因？**

答：Java支持多线程同时访问一个对象或者对象的成员变量，由于每个线程可以拥有这个变量的拷贝（虽然对象以及成员变量分配的内存是在共享内存中的，但是每个执行的线程还是可以拥有一份拷贝，这样做的目的是加速程序的执行，这是现代多核处理器的一个显著特性），所以程序在执行过程中一个线程看到的变量并不一定是最新的。

Q73：**volatile和synchronized是如何解决可见性问题的？**

答：volatile可以用来修饰成员变量，就是告知程序任何对该变量的访问均需从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

synchronized可以修饰方法或者以同步代码块的形式进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法和同步代码块中，它保证了线程对变量访问的可见性和排他性。

Q74：**监视器是什么，有什么作用？**

答：任意一个对象都拥有自己的监视器，当这个对象由同步块或者这个对象的同步方法调用时，执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法，而没有获取到监视器（执行该方法）的线程就会被阻塞在同步块和同步方法的入口处，进入同步队列，线程状态变为BLOCKED状态，直到前一个获得了锁的线程释放了锁该阻塞线程才会被唤醒重新尝试获取监视器。

Q75：**wait()、notify()和notifyAll()的使用细节？**

答：①使用wait()、notify()和notifyAll()时需要先对调用对象加锁。②调用wait()方法后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列。③notify()和notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()和notifyAll()的线程释放锁后，等待线程才有机会从wait()返回。④notify()方法将等待队列的一个等待线程从等待队列移到同步队列中，而notifyAll()方法将等待队列中的全部线程移到同步队列，被移动的线程状态由WAITING变为BLOCKED。⑤从wait()方法返回的前提是获得了调用对象的锁。

Q76：管道IO流的作用？

答：管道IO流和普通IO流或网络IO流的不同之处在于它主要用于线程之间的数据传输，而传输的媒介为内存。管道IO流主要包括4种具体实现：PipedOutputStream、PipedInputStream、PipedWriter、PipedReader，前两种面向字节，后两种面向字符。

Q77：**join方法的作用？**

答：join方法使当前线程必须等待调用join方法的线程执行完毕后才能继续执行，除了无参join方法外还有带超时参数的join方法，在指定时间内没有结束就会从该方法返回。底层是通过wait和notifyAll方法实现的，当调用join的线程终止时会调用自身notifyAll方法通知所有等待在该线程对象上的线程。

Q78：**ThreadLocal的作用？**

答：ThreadLocal即线程变量，是一个以ThreadLocal对象为键，任意对象为值得存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。可以通过set方法设置一个值，在当线程下再通过get方法获取到原先设置的值。

Q79：**为什么要使用线程池？**

答：对于服务端的程序经常面对的是客户端传入的短小（执行时间短、工作内容较为单一）任务，需要服务端快速处理并返回结果。如果服务端每接收到一个任务就创建一个线程然后进行执行，这在原型阶段是个不错的选择但是面对成千上万的任务递交服务器时如果还是采用该方式那么将会创建数以万计的线程，使操作系统频繁进行线程上下文切换，无故增加系统负载，而线程的创建和消亡都是需要耗费系统资源的。

线程池技术可以很好地解决该问题，它预先创建了若干数量的线程，并且不能由用户直接对线程的创建进行控制，在这个前提下重复使用固定或较为固定数目的线程来完成任务的执行。这样做一方面消除了频繁创建和消亡消除的系统资源开销，另一方面，面对过量任务的提交能够平缓地处理。

Q80：**Lock锁和synchronized的区别？**

答：在JDK1.5之后，并发包中新增了Lock接口以及相关实现类用来实现锁功能，它提供了与synchronized关键字类似的同步功能，只是在使用时需要显式地获取和释放锁。虽然它缺少了隐式获取锁的便捷性，但是却拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等多种synchronized关键字不具备的同步特性。

使用synchronized关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放，简化了同步的管理，但是扩展性没有显式锁好。

Q81：**Lock接口提供的synchronized关键字不具备的主要特性？**

答：①Lock可以尝试非阻塞地获取锁，当前线程尝试获取锁，如果这一时刻没有被其他线程获取到则成功获取并持有锁。②能被中断地获取锁，与synchronized不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放。③超时获取锁，在指定的截至时间之前获取锁，如果截止时间到了仍旧无法获取锁则返回。

Q82：**AQS是什么？**

答：AQS是抽象队列同步器Abstract Queued Synchronizer，是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作，并发包的作者期望它成为实现大部分同步需求的基础。

Q83：AQS的主要实现方式是什么？

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法（getState、setState和compareAndSetState）来进行操作，因为它们能够保证状态的改变是安全的。子类推荐被定义为自定义同步组件的静态内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型地同步组件（ReentrantLock、ReentrantReadWriteLock和CountDownLacth等）。

Q84：同步器和锁的联系？

答：同步器是实现锁的关键，在锁的实现中聚合同步器，利用同步器实现锁的语义。锁是面向使用者的，它定义了使用者与锁交互的接口，隐藏了实现细节；同步器面对的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。锁和同步器很好地隔离了使用者和实现者所关注的领域。

Q85：AQS的实现包括哪些方面？

答：队列同步器的从实现角度分为多方面，主要包括同步队列、独占式同步状态的获取与释放、共享式同步状态的获取与释放，以及超时获取同步状态等同步器的核心数据与模板方法。

Q86：同步队列的原理？

答：同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等构造成一个节点并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。

Q87：同步队列的节点保存哪些信息？

答：同步队列中的节点用来保存获取同步状态失败的线程引用、等待状态以及前驱和后继节点。节点是构成同步队列的基础，同步器拥有首节点和尾节点，没有成功获取同步状态的线程将会成为节点加入该队列的尾部。

Q88：同步队列节点的等待状态有哪些类型？

答：①CANCELLED，值为1，由于在同步队列中等待的线程等待超时或者被中断需要从同步队列中取消等待，节点进入该状态将不会变化。②SIGNAL，值为-1，后继节点的线程处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行。③CONDITION，值为-2，节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal方法后该节点将会从等待队列转移到同步队列中，加入到对同步状态的获取中。④PROPAGATE，值为-3，表示下一次共享式同步状态获取将会无条件地被传播下去。⑤INITIAL，值为0，初始状态。

Q89：独占式同步状态的获取和释放流程？

答：在获取同步状态时，同步器调用acquire方法，维护一个同步队列，使用tryAcquire方法安全地获取线程同步状态，获取状态失败的线程会构造同步节点并通过addWaiter方法被加入到同步队列的尾部，并在队列中进行自旋。之后会调用acquireQueued方法使得该节点以死循环的方式获取同步状态，如果获取不到则阻塞节点中的线程，而被阻塞线程的唤醒主要依靠前驱节点的出队或阻塞节点被中断实现，移出队列或停止自旋的条件是前驱节点是头结点并且成功获取了同步状态。

在释放同步状态时，同步器调用tryRelease方法释放同步状态，然后调用unparkSuccessor方法（该方法使用LockSupport唤醒处于等待状态的线程）唤醒头节点的后继节点，进而使后继节点重新尝试获取同步状态。

Q90：为什么只有当前驱节点是头节点时才能够尝试获取同步状态？

答：①头节点是成功获取到同步状态的节点，而头节点的线程释放同步状态之后，将会唤醒其后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点。

②维护同步队列的FIFO原则，节点和节点在循环检查的过程中基本不相互通信，而是简单地判断自己的前驱是否为头节点，这样就使得节点的释放规则符合FIFO，并且也便于对过早通知的处理（过早通知是指前驱节点不是头结点的线程由于中断而被唤醒）。

Q91：共享式同步状态的获取和释放流程？

答：在获取同步状态时，同步器调用acquireShared方法，该方法调用tryAcquireShared方法尝试获取同步状态，返回值为int类型，当返回值大于等于0时，表示能够获取到同步状态。因此在共享式获取锁的自旋过程中，成功获取到同步状态并退出自旋的条件就是该方法的返回值大于等于0。

释放同步状态时，调用releaseShared方法，释放同步状态之后将会唤醒后续处于等待状态的节点。对于能够支持多线程同时访问的并发组件，它和独占式的主要区别在于tryReleaseShared方法必须确保同步状态（或资源数）线程安全释放，一般通过循环和CAS来保证，因为释放同步状态的操作会同时来自多个线程。

Q92：独占式超时获取同步状态的流程？

答：通过调用同步器的doAcquireNanos方法可以超时获取同步状态，即在指定的时间段内获取同步状态，如果获取到同步状态则返回true，否则返回false。该方法提供了传统Java同步操作（例如synchronized关键字）所不具备的特性。

Q93：响应中断的同步状态获取过程？

答：在JDK1.5之前当一个线程获取不到锁而被阻塞到synchronized之外时，对该线程进行中断操作，此时该线程的中断标志位会被修改，但线程依旧阻塞在synchronized上等待着获取锁。在JDK1.5中，同步器提供了acquireInterruptibly方法，这个方法在等待获取同步状态时，如果当前线程被中断，会立即返回并抛出InterruptedException。

Q94：独占式超时获取同步状态的原理？

答：超时获取同步状态的过程可以被视为响应中断获取同步状态过程的“增强版”，doAcquireNanos方法在支持响应中断的基础上增加了超时获取的特性，针对超时获取，主要需要计算出需要睡眠的时间间隔nanosTimeout，为了防止过早通知，nanosTimeout的计算公式为nanosTimeout-=now-lastTime，其中now为当前唤醒时间，lastTime为上次唤醒时间，如果nanosTimeout大于0则表示超时时间未到，需要继续睡眠nanosTimeout纳秒，否则表示已经超时。

Q95：独占式超时获取同步状态和独占式获取同步状态的区别？

答：在独占式超时获取同步状态的过程的doAcquireNanos中，当节点的前驱节点为头节点时尝试获取同步状态，如果获取成功则从该方法返回，这个过程和独占式同步获取的过程类似，但是在同步状态获取失败的处理上有所不同。

如果当前线程获取同步状态失败，独占式超时获取同步状态中会判断是否超时，如果没有超时就重新计算超时间隔，然后使当前线程等待该间隔时间，如果在该时间内没有获取到同步状态就会从等待逻辑中自动返回。而独占式获取同步状态的过程中如果没有获取到同步状态就会使当前线程一直处于等待状态。

Q96：超时时间过小时对超时等待有哪些影响？

答：nanosTimeout过小时（小于等于1000纳秒），将不会使线程进行超时等待，而是进入快速自旋过程。因为非常短的超市等待无法做到精确，如果这时再进行超时等待相反会让nanosTimeout的超时从整体上表现得反而不精确，因此在超市非常短的情况下同步器会进入无条件的快速自旋。

Q97：**什么是可重入锁？**

答：重入锁就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁，除此之外该锁还支持获取锁的公平和非公平性选择。synchronized关键字隐式地支持重进入，ReentrantLock虽然不能像synchronized关键字一样支持隐式的重进入，但是在调用lock方法时已经获取到锁的线程能够再次调用lock方法获取锁而不被阻塞。

Q98：**什么是锁的公平性？**

答：如果在绝对时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反正就是不公平的。公平的获取锁也就是等待时间最长的线程优先获取锁，也可以说锁的获取是顺序的，ReentrantLock的构造方法中可以通过设置参数控制锁的公平性。

公平锁机制往往没有非公平锁的效率高，非公平锁地吞吐量更大，但是公平锁能够减少饥饿发生的概率，保证了锁地获取按照FIFO顺序，等待越久的请求越是能优先得到满足。

Q99：**什么是重进入？**

答：重进入指的是任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决两个问题：①线程再次获取锁，锁需要去识别获取锁的线程是否为当前占有锁的线程，如果是则再次获取成功。②锁的最终释放，线程重复n次获取了锁，随后在第n次释放该锁后，其他现场能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而被锁释放时，技术自减，当计数为0时表示锁已经成功释放。

Q100：**ReentrantLock的可重入如何实现？**

答：以非公平锁为例，通过nonfairTryAcquire方法获取锁，该方法增加了再次获取同步状态的处理逻辑：通过判断当前线程是否为获取锁的线程来决定获取操作是否成功，如果是获取锁的线程再次请求则将同步状态值进行增加并返回true，表示获取同步状态成功。

成功获取锁的线程再次获取锁，只是增加了同步状态值，这就要求ReentrantLock在释放同步状态时减少同步状态值。如果该锁被获取了n次，那么前（n-1）次tryRelease方法必须都返回fasle，只有同步状态完全释放了才能返回true，可以看到该方法将同步状态是否为0作为最终释放的条件，当同步状态为0时，将占有线程设置为null，并返回true，表示释放成功。

Q101：**ReentrantLock的可重入的公平锁如何实现？**

答：对于非公平锁只要CAS设置同步状态成功则表示当前线程获取了锁，而公平锁则不同。公平锁使用tryAcquire方法，该方法与nonfairTryAcquire的唯一区别就是判断条件中多了对同步队列中当前节点是否有前驱节点的判断，如果该方法返回true表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

Q102：**什么是读写锁？**

答：像Mutex和ReentrantLock都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读写锁使得并发性相比一般的排他锁有了很大提升。

Q103：**读写锁的特点？**

答：除了保证写操作对读操作的可见性以及并发性的提升之外，读写锁能够简化读写交互场景的编程方式。只需要在读操作时获取读锁，写操作时获取写锁即可，当写锁被获取时后续（非当前写操作线程）的读写操作都会被阻塞，写锁释放之后所有操作继续执行，编程方式相对于使用等待/通知机制的实现方式而言变得简单明了。

Q104：**读写锁ReentrantReadWriteLock的特性？**

答：①公平性选择：支持非公平（默认）和公平的锁获取方式吞吐量还是非公平性优于公平。

②重进入：该锁支持重进入，以读写线程为例：读线程在获取了读锁之后能够再次获得读锁。而写线程在获取了写锁之后能再次获得写锁，同时也可以获取读锁。

③锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级为读锁。

Q105：读写锁的状态是怎么设计的？

答：读写锁同样依赖自定义同步器来实现同步功能，而读写状态就是其同步器的同步状态。读写锁的自定义同步器需要在同步状态（一个整形变量）上维护多个读线程和一个写线程的状态。如果在一个int型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写。

假设同步状态值为S，写状态等于S&0x0000FFFF（将高17位全部抹去），读状态等于S>>>16（无符号右移16位），当写状态增加1时，等于S+1，当读状态增加1时，等于S+（1<<16）。根据状态的划分能得出一个推论：S不等于0时，当写状态等于0时，则读状态大于0，即读锁已被获取。

Q106：写锁的获取和释放过程？

答：写锁是一个支持重进入的排他锁，如果当前线程已经获得了写锁则增加写状态，如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获得写锁的线程则当前线程进入等待状态。写锁的释放与ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续读写线程可见。

Q107：**为什么存在读锁时写锁会阻塞？**

答：读写锁要确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取则其他读写线程的后续访问均被阻塞。

Q108：读锁的获取和释放过程？

答：读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有其他写线程访问（或者写线程为0）时，读锁总会被成功地获取，而所做的只是线程安全地增加读状态。如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已被其他线程获取则进入等待状态。如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全，依靠CAS保证）增加读状态，成功获取读锁。

读锁的每次释放均会减少读状态，减少的值是（1<<16），读锁的每次释放是线程安全的，可能有多个读线程同时释放读锁。

Q109：JDK1.6对读锁有什么改动？

答：获取读锁的实现从JDK1.5到JDK1.6变得复杂许多，主要原因是新增了一些功能，例如getReadHoldCount方法，作用是返回当前线程获取读锁的次数。读状态是所有线程获取读锁次数的总和，而每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，由线程自身维护，这使获取读锁的实现变得复杂。

Q110：**锁降级是什么？**

答：锁降级指的是写锁降级成为读锁，如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级指的是把持住（当前拥有的）写锁，再获取到读锁，随后释放先前拥有的写锁的过程。

Q111：**锁降级中读锁的获取是否有必要？**

答：是必要的，主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程A获取了写锁修改了数据，那么当前线程是无法感知线程A的数据更新的。如果当前线程获取读锁，即遵循锁降级的步骤，线程A将会被阻塞，直到当前线程使用数据并释放读锁之后，线程A才能获取写锁并进行数据更新。

Q112：LockSupport是什么？

答：当需要阻塞或唤醒一个线程的时候，都会使用LockSupport工具类完成相应工作，LockSupport定义了一组公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而LockSupport也成为构建同步组件的基础工具。

LockSupport定义了一组以park开头的方法用来阻塞当前线程，以及unpark方法来唤醒一个被阻塞的线程。在JDK1.6中，新增了3个含义阻塞对象的park方法，用以替代原有的park方法。

Q113：**Condition的作用？**

答：Condition接口提供了类似Object监视器方法，与Lock配合可以实现等待/通知模式。Condition对象是由Lock对象创建出来的，因此Condition是依赖Lock对象的。一般会将Condition对象作为成员变量，当调用await方法后当前线程会释放锁并在此等待，而其他线程调用Condition对象的signal方法，通知当前线程后，当前线程才从await方法返回并且在返回前已经获取了锁。

Q114：Condition是怎么实现的？

答：ConditionObject是同步器AQS的内部类，因为Condition的操作需要获取相关的锁，所以作为同步器的内部类也较为合理。每个Condition对象都包含着一个等待队列，该等待队列是Condition对象实现等待/通知功能的关键。Condition的实现主要包括了等待队列、等待和通知。

Q115：等待队列的原理？

答：等待队列是一个FIFO队列，在队列中的每个节点都包含了一个线程引用，该线程就是在ConditionObject对象上等待的线程，如果一个线程调用了await方法，那么该线程会释放锁、构造成节点加入等待队列并进入等待状态。事实上，节点的定义复用了同步其中节点的定义，也就是说同步队列和等待队列中的节点类型都是同步器的静态内部类Node。

一个ConditionObject包含一个等待队列，ConditionObject拥有首节点和尾节点。Object拥有首尾节点的引用，而新增节点只需要将原有的尾节点nextWaiter指向它，并且更新尾节点即可。节点引用更新的过程并没有用CAS保证，因为调用await方法的线程必定是获取了锁的线程，也就是说该过程是由锁来保证线程安全的。

Q116：await方法的原理？

答：如果从队列的角度看await方法，当调用await方法时相当于同步队列的首节点（获取了锁的节点）移动到Condition对象的等待队列中，首节点不会直接加入等待队列，而是通过addConditionWaiter方法把当前线程构造成一个新的节点并将其加入等待队列中。加入等待队列后，释放同步状态，唤醒同步队列中的后继节点然后进入等待状态。如果不是通过其他线程调用signal方法唤醒而是对await线程进行中断，会抛出InterruptedException。

Q117：signal方法的原理？

答：该方法会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移到同步队列中。调用该方法的前置条件是当前线程必须获取了锁，signal方法进行了检查，判断当前线程是否是获取了锁的线程，接着获取等待队列的首节点，将其移动到同步队列并使用LockSupport唤醒节点中的线程。被唤醒后的线程将从await方法中的while循环退出，进而调用同步器的acquireQueued方法加入到获取同步状态的竞争中。成功获取同步状态（或者说锁）后，被唤醒的线程将从先前调用的await方法返回，此时该线程已成功获取了锁。signalAll方法相当于对等待队列中的每个节点执行一次signal方法，效果就是将等待队列中的节点全部移到到同步队列中并唤醒每个节点的线程。

Q118：**什么是阻塞队列？**

答：阻塞队列是一个支持两个附加操作的队列，这两个附加的操作支持阻塞的插入和移除方法。当队列满时，队列会阻塞插入元素的线程，直到队列不满。当队列为空时，获取元素的线程会等待队列变为非空。阻塞队列常用于生产者和消费者的场景，生产者向队列里添加元素，消费者从队列中获取元素，阻塞队列就是生产者用来存放元素，消费者用来获取元素的容器。

Q119：**Java中有哪些阻塞队列？**

答：①ArrayBlockingQueue，一个由数组结构组成的有界阻塞队列，按照FIFO的原则对元素排序，默认情况下不保证线程公平地访问队列，有可能先阻塞地线程最后才访问队列。

②LinkedBlockingQueue，一个由链表结构组成的有界阻塞队列，队列的默认和最大长度为Integer的最大值，按照FIFO原则排序。

③PriorityBlockingQueue，一个支持优先级排序的无界阻塞队列，默认情况下元素按照顺序升序排序。也可以自定义compareTo方法指定元素排序规则，或者初始化时指定构造方法的参数Comparator对元素排序，不能保证同优先级元素的顺序。

④DelayQueue，一个支持延时获取元素的无界阻塞队列，使用优先级队列实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素，只有延时期满时才能从队列中获取元素。适用于以下场景：①缓存系统的设计，一旦能从延迟队列获取元素说明缓存有效期到了。②定时任务调度，保存当天将要执行的任务和执行时间，一旦获取到任务就立刻开始执行。

⑤SynchronousQueue，一个不存储元素的阻塞队列，每一个put操作必须等待一个take操作，否则不能继续添加元素。默认使用非公平策略，也支持公平策略，适用于传递性场景，吞吐量高于ArrayBlockingQueue和LinkedBlockingQueue。

⑥LinkedTransferQueue，一个由链表结构组成的无界阻塞队列，相对于其他阻塞队列多了tryTransfer和transfer方法。transfe方法：如果当前有消费者正在等待接收元素，transfer方法可以把生产者传入的元素立刻传输给消费者，如果没有，会将元素放在队列的尾节点等到该元素被消费者消费了才返回。tryTransfer方法：用来试探生产者传入的元素能否直接传给消费者，如果没有消费者等待接收元素返回false，和transfer的区别时无论消费者是否接受都会立即返回，transfer是等到消费者消费了才返回。

⑦LinkedBlockingDeque，一个由链表结构组成的双向阻塞队列，可以从队列的两端插入和移除元素，多了一个操作队列的入口，在多线程同时入队时就少了一半竞争。

Q120：**阻塞队列的实现原理？**

答：使用通知模式实现，所谓通知模式就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。JDK中使用了Condition条件对象来实现。当往队列里插入一个元素，如果队列不可用，那么阻塞生产者主要通过LockSupport.park(this)实现。

Q121：原子操作类是什么？

答：JDK1.5开始提供了atomic包，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。主要包括4类，原子更新基本类型、原子更新数组、原子更新引用类型和原子更新属性。在原子更新属性类中有支持带有版本号的更新方法，可用于解决CAS操作时出现的ABA问题。

Q122：CountDownLatch的作用？

答：允许一个或多个线程等待其他线程完成操作，构造方法接收一个int类型的参数作为计数器，如果要等待n个点就传入n。每次调用countDown方法时n就会减1，await方法会阻塞当前线程直到n变为0，由于countDown方法可用在任何地方，所以n个点既可以是n个线程也可以是1个线程里的n个执行步骤。用在多线程时，只需要把这个CountDownLatch的引用传递到线程里即可。

Q123：CyclicBarrier的作用？

答：CyclicBarrier是同步屏障，它的作用是让一组线程到达一个屏障（或同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被拦截的线程才会继续运行。构造方法中的参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier自己已到达屏障，然后当前线程被阻塞。还支持在构造方法中传入一个Runable类型的任务，当线程到达屏障时会优先执行该任务。适用于多线程计算数据，最后合并计算结果的应用场景。

Q124：CountDownLacth和CyclicBarrier的区别？

答：CountDownLacth的计数器只能用一次，而CyclicBarrier的计数器可使用reset方法重置，所以CyclicBarrier能处理更为复杂的业务场景，例如计算错误时可用重置计数器重新计算。

CyclicBarrier还提供了其他有用的方法，例如getNumberWaiting可以获取CyclicBarrier阻塞的线程数量，isBroken方法用来了解阻塞的线程是否被中断。

Q125：Semaphore的作用？

答：Semaphore是信号量，用来控制同时访问特定资源的线程数量，它通过协调各个线程以保证合理的使用公共资源。信号量可以用于流量控制，特别是公共资源有限的应用场景，比如数据库连接。Semaphore的构造方法参数接收一个int型数字，表示可用的许可证数量，即最大并发数。使用acquire获得一个许可证，使用release方法归还许可证，还可以用tryAcquire尝试获得许可证。

Q126：Exchanger的作用？

答：Exchanger交换者是用于线程间协作的工具类，用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange方法它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时这两个线程就可以交换数据，将本线程生产出的数据传递给对方。应用场景包括遗传算法、校对工作等。

Q127：**线程池有哪些好处？**

答：①降低资源消耗，通过重复利用已创建的线程降低线程创建和消耗的开销。

②提高响应速度，当任务到达时，任务可以不需要等到线程创建就可以立即执行。

③提高线程的可管理性，线程是稀缺资源，如果无限制地创建不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

Q128：**线程池的处理流程？**

答：①线程池判断核心线程池是否已满，如果不是则创建一个新的工作线程来执行任务（工作线程数<corePoolSize，这一步需要获取全局锁）。②如何核心线程池已经满了，判断工作队列是否已满，如果没有就将任务存储在工作队列中（工作线程数>=corePoolSize）。③如果工作队列满了，判断线程池是否已满，如果没有就还是创建一个新的工作线程来执行任务（工作线程数<maximumPoolSize）。④如果线程池已满，就按照线程池的拒绝执行策略来处理无法执行的任务（工作线程数>maximumPoolSize）。

线程池采取这种设计思路是为了在执行execute方法时尽可能地避免获取全局锁，在线程池完成预热之后，即当前工作线程数>=corePoolSzie时，几乎所有的execute方法都是执行步骤2，不需要获取全局锁。

Q129：**工作线程的任务是什么？**

答：线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务之后，还会循环获取工作队列中的任务来执行。线程池中的线程执行任务分为两种情况：①在execute方法中创建一个线程时会让这个线程执行当前任务。②这个线程执行完任务之后，就会反复从阻塞工作队列中获取任务并执行。

Q130：**ThreadPoolExecutor创建有哪些参数，具体含义是什么？**

答：①corePoolSize：线程池的基本大小，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池的基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有的基本线程。

②workQueue：工作队列，用于保存等待执行任务的阻塞队列，可以选择以下的阻塞队列：ArrayBlockQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockQueue等。

③maximumPoolSize：线程池允许的最大线程数，如果工作队列已满，并且创建的线程数小于最大线程数，则线程池还会创建新的线程执行任务，如果使用的时无界阻塞队列该参数是无意义的。

④threadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

⑤handler：拒绝策略，当队列和线程池都满了说明线程池处于饱和状态，那么必须采取一种拒绝策略处理新提交的任务，默认情况下使用AbortPolicy直接抛出异常，CallerRunsPolicy表示重新尝试提交该任务，DiscardOldestPolicy表示抛弃队列里最近的一个任务并执行当前任务，DiscardPolicy表示直接抛弃当前任务不处理。也可以自定义该策略。

⑥keepAliveTime：线程活动的保持时间，线程池工作线程空闲后保持存活的时间，所以如果任务很多，且每个任务的执行时间较短，可以调大时间提高线程的利用率。

⑦unit：线程活动保持时间的单位，有天、小时、分钟、毫秒、微秒、纳秒。

Q131：**如何向线程池提交任务？**

答：可以使用execute和submit方法向线程池提交任务。execute方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功了。submit方法用于提交需要返回值的任务，线程池会返回一个Future类型的对象，通过该对象可以判断任务是否执行成功，并且可以通过该对象的get方法获取返回值，get方法会阻塞当前线程直到任务完成，带超时参数的get方法会在指定时间内返回，这时任务可能还没有完成。

Q132：**关闭线程池的原理？**

答：可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。区别是shutdownNow首先将线程池的状态设为STOP，然后尝试停止所有正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设为SHUTDOWN，然后中断所有没有正在执行任务的线程。

只要调用了这两个方法中的一个，isShutdown方法就会返回true，当所有任务都已关闭后才表示线程池关闭成功，这时调用isTerminated方法会返回true。通常调用shutdown方法来关闭线程池，如果任务不一定要执行完则可以调用shutdownNow方法。

Q133：如何合理设置线程池？

答：首先可以从以下角度分析：①任务的性质：CPU密集型任务、IO密集型任务和混合型任务。②任务的优先级：高、中和低。③任务的执行时间：长、中和短。④任务的依赖性：是否以来其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理，CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。混合型任务如果可以拆分将其拆分为一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大那么分解后的吞吐量将高于串行执行的吞吐量，如果相差太大则没必要分解。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue处理。

执行时间不同的任务可以交给不同规模的线程池处理，或者使用优先级队列。

以来数据库连接池的任务，由于线程提交SQL后需要等待数据库返回的结果，等待的时间越长CPU空闲的时间就越长，因此线程数应该尽可能地设置大一些提高CPU的利用率。

建议使用有界队列，能增加系统的稳定性和预警能力，可以根据需要设置的稍微大一些。

Q134：线程池如何进行监控？

答：①taskCount，线程池需要执行的任务数量。②completedTaskCount，线程池在运行过程中已经完成的任务数量，小于或等于taskCount。③largestPoolSize，线程池里曾经创建过的最大线程数量，通过这个数据可以知道线程池是否曾经满过，如果该数值等于线程池的最大大小表示线程池曾经满过。④getPoolSize，获取线程池的线程数量，如果线程池不销毁的化线程池里的线程不会自动销毁，所以这个数值只增不减。⑤getActiveCount，获取活动的线程数。

通过扩展线程池进行监控，可以继承线程池来自定义，重写线程池的beforeExecute、afterExecute和terminated方法，也可以在任务执行前、执行后和线程池关闭前来执行一些代码进行监控，例如监控任务的平均执行时间、最大执行时间和最小执行时间。

Q135：Executor框架的调度模型是什么？

答：在HotSpot VM的线程模型中，Java线程被一对一映射为本地操作系统线程，Java线程启动时会创建一个本地操作系统线程，当该Java线程终止时，这个操作系统线程也会被回收，操作系统会调度所有线程并将它们分配给可用的CPU。

Executor框架的调度模型是一种两级调度模型。在上层，Java多线程程序通常把应用分解为若干任务，然后使用用户级的调度器即Executor框架将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。

Q136：**Executor框架的结构？**

答：主要由以下三部分组成：

①任务，包括被执行任务需要实现的接口，Runnable或Callable接口。

②任务的执行，包括任务执行机制的核心接口Executor（Executor框架的基础，将任务的提交和执行分离开来），以及继承自Executor的ExecutorService接口（ThreadPoolExecutor和ScheduledThreadPoolExecutor）。

③异步计算的结果，包括接口Future和实现Future接口的FutureTask类。当我们把Runnable接口或Callable接口的实现类提交（submit）给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，ThreadPoolExecutor或ScheduledThreadPoolExecutor会向我们返回一个FutureTask对象。

Q137：**ThreadPoolExecutor是什么？**

答：ThreadPoolExecutor是Executor框架最核心的类，是线程池的实现类，主要有三种。

①FixedThreadPool，可重用固定线程数的线程池，corePoolSize和maximumPoolSize都被设置为创建时的指定参数nThreads，当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止，这里将其设置为0L表示多余空闲线程将被立即终止。该线程池使用的工作队列是无界阻塞队列LinkedBlockingQueue（队列容量为Integer的最大值）。适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，适用于负载比较重的服务器。

②SingleThreadExecutor，使用单个线程的线程池，corePoolSize和maximumPoolSize都被设置为1，其他参数和FiexedThreadPool相同。适用于需要保证顺序执行各个任务，并且在任意时间点不会有多个线程是活动的的应用场景。

③CachedThreadPool，一个根据需要创建线程的线程池，corePoolSize被设置为0，maximumPoolSize被设置为Integer的最大值，将keepAliveTime设为60L，意味着空闲线程等待时间最长为1分钟。该线程池使用的工作队列是没有容量的SynchronousQueue，但是maximumPoolSize设为Integer最大值，如果主线程提交任务的速度高于线程处理的速度，线程池会不断创建新线程，极端情况下会创建过多线程而耗尽CPU和内存资源。适用于执行很多短期异步任务的小程序，或者负载较轻的服务器。

Q138：**ScheduledThreadPoolExecutor是什么？**

答：ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，主要用来在给定的延迟之后运行任务，或者定期执行任务。其功能与Timer类似，但是功能更加强大、更灵活。Timer对应的是单个后台线程，而ScheduledThreadPoolExecutor可以在构造方法中指定多个后台线程数。为了实现周期性的执行任务，使用DelayQueue作为工作队列，获取任务和执行周期任务后的处理都不同，主要有两种。

①ScheduledThreadPool：包含若干线程的ScheduledThreadPoolExecutor，创建固定线程个数的线程池。适用于需要多个后台线程执行周期任务，同时为了满足资源管理的需求而需要限制后台线程数量的应用场景。

②SingleThreadScheduledExecutor：只包含一个线程的ScheduledThreadPoolExecutor，适用于单个后台线程执行周期任务，同时需要保证顺序执行各个任务的应用场景。

Q139：**ScheduledThreadPoolExecutor的原理？**

答：将待调度任务放入一个DelayQueue中，调度任务主要有三个参数，long类型的time表示这个任务将要被执行的具体时间，long类型的sequenceNumber表示这个任务被添加到线程池的序号，long类型的period表示任务执行时间间隔。DelayQueue封装了一个PriorityQueue，队列按照time进行排序，如果time相同则比较sequenceNumber，越小的排在前面，即如果两个任务的执行时间相同，先提交的任务先被执行。

Q140**：Runnable接口和Callable接口的区别？**

答：两个接口的相同点是Runnable接口和Callable接口的实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行，不同点是Runnable不会返回结果，Callable可以返回结构。除了可以自己创建Callable接口的对象外，还可以使用工厂类Executors将一个Runnable对象包装为一个Callable对象。

Q141：使用无界阻塞队列对线程池的影响？

答：①当线程池中的线程数达到corePoolSize之后新任务将在无界队列中等待，因此线程池中的数量不会超过corePoolSize。②因此使用无界队列时maximumPoolSize和keepAliveTime均是无效参数。③由于使用无界队列，线程池不会拒绝任务。

Q142：FutureTask有哪些状态？

答：FutureTask除了实现了Future接口之外，还实现了Runnable接口。因此FutureTask可以交给Executor执行，也可以由调用线程直接执行即调用FutureTask对象的run方法，根据run方法被执行的时机，FutureTask可以处于三种状态：①未启动，当FutureTask对象被创建，且没有执行run方法之前的状态。②已启动，当run方法处于被执行过程中，FutureTask对象处于已启动状态。③已完成，当run方法执行后正常完成或执行run方法中抛出异常或调用cancel方法取消时，FutureTask对象处于已完成状态。

当处于未启动或已启动状态时，get方法将阻塞线程，当处于已完成状态时会立即返回结果或抛出异常。当处于未启动状态时，cancel方法会导致此任务永远不会执行，当处于已启动状态时，执行cancel(true)方法，将以中断执行此任务的方式来试图停止该任务，执行cancel(false)方法，将不会对正在执行此任务的线程产生应用，当处于已完成状态时，cancel方法返回false。

Q143：FutureTask的实现原理？

答：FutureTask的实现基于AQS，基于合成复用的设计原则，FutureTask声明了一个内部私有的继承于AQS的子类Sync，对Future的所有公有方法的调用都会委托给这个内部的子类。AQS被作为模板方法模式的基础类提供给FutureTask的内部子类Sync，这个内部的子类只需要实现状态检查和更新的方法即可，这些方法将控制FutureTask的获取和释放操作。具体来说，Sync实现了AQS的tryAcquireShared和tryReleaseShared方法来检查和更新同步状态。

Q144：基于AQS实现的同步器有什么共同点？

答：①至少有一个acquire操作，这个操作阻塞调用线程，直到AQS的状态允许这个线程继续执行。FutureTask中的acquire操作为get方法调用。②至少有一个release操作，这个操作改变AQS的状态，改变后的状态可允许一个多多个阻塞线程解除阻塞。FutureTask中的release操作包括run方法和cancel方法。

Q145：FutureTask的get方法原理？

答：①调用AQS的acquireSharedInterruptibly方法，首先回调在子类Sync中实现的tryAcquireShared方法来判断acquire操作是否可以成功。acquire操作成功的条件为：state为执行完成状态或取消状态，且runner不为null。②如果成功get方法立即返回，如果失败则到线程等待队列中去等待其他线程执行release操作。③当其他线程执行release操作唤醒当前线程后，当前线程再次执行tryAcquireShared将返回1，当前线程将理课线程等待队列并唤醒它的后继线程。④返回最终结果或抛出异常。

Q146：FutureTask的run方法原理？

答：①执行在构造方法中的指定任务。②以原子方式更新同步状态，如果操作成功就设置代表计算结果的变量result的值为Callable的call方法的返回值，然后调用AQS的releaseShared方法。③AQS的releaseShared方法首先回调子类Sync中实现的tryReleaseShared来执行release操作（设置运行任务的线程runner为null，然后返回true），然后唤醒线程等待队列的第一个线程。④调用FutureTask的done方法。