简述java内存模型（JMM）

java内存模型定义了程序中各种变量的访问规则。其规定所有变量都存储在主内存，线程均有自己的工作内存。
工作内存中保存被该线程使用的变量的主内存副本，线程对变量的所有操作都必须在工作空间进行，不能直接读写主内存数据。操作完成后，线程的工作内存通过缓存一致性协议将操作完的数据刷回主存。

简述as-if-serial

编译器等会对原始的程序进行指令重排序和优化。但不管怎么重排序，其结果和用户原始程序输出预定结果一致。

简述happens-before八大原则

程序次序规则：一个线程内写在前面的操作先行发生于后面的。

锁定规则： unlock 操作先行发生于后面对同一个锁的 lock 操作。

volatile 规则：对 volatile 变量的写操作先行发生于后面的读操作。

线程启动规则：线程的 start 方法先行发生于线程的每个动作。

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。

线程终止规则：线程中所有操作先行发生于对线程的终止检测。

对象终结规则：对象的初始化先行发生于 finalize 方法。

传递性规则：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C

as-if-serial 和 happens-before 的区别

as-if-serial 保证单线程程序的执行结果不变，happens-before 保证正确同步的多线程程序的执行结果不变。

简述原子性操作

一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行，这就是原子性操作。

简述线程的可见性

可见性指当一个线程修改了共享变量时，其他线程能够立即得知修改。volatile,synchronized,final都能保证可见性。

简述有序性

即虽然多线程存在并发和指令优化等操作，在本线程内观察该线程的所有执行操作是有序的。

简述java中volatile关键字作用

1. 保证变量对所有线程的可见性。
   当一条线程修改了变量值，新值对于其他线程来说是立即可以得知的。
2. 禁止指令重排序优化。使用 volatile 变量进行写操作，汇编指令带有 lock 前缀，相当于一个内存屏障，编译器不会将后面的指令重排到内存屏障之前。

java线程的实现方式

1. 实现Runnable接口
2. 继承Thread类。
3. 实现Callable接口

简述java线程的状态

线程状态有New, RUNNABLE, BLOCK, WAITING, TIMED_WAITING, THERMINATED
NEW：新建状态，线程被创建且未启动，此时还未调用 start 方法。

RUNNABLE: 运行状态。其表示线程正在JVM中执行，但是这个执行，不一定真的在跑，也可能在排队等CPU。

BLOCKED：阻塞状态。线程等待获取锁，锁还没获得。

WAITING: 等待状态。线程内run方法运行完语句Object.wait()/Thread.join()进入该状态。

TIMED_WAITING：限期等待。在一定时间之后跳出状态。调用Thread.sleep(long) Object.wait(long) Thread.join(long)进入状态。其中这些参数代表等待的时间。

TERMINATED：结束状态。线程调用完run方法进入该状态。

简述线程通信的方式

1. volatile 关键词修饰变量，保证所有线程对变量访问的可见性。
2. synchronized关键词。确保多个线程在同一时刻只能有一个处于方法或同步块中。
3. wait/notify方法
4. IO通信

简述线程池

没有线程池的情况下，多次创建，销毁线程开销比较大。如果在开辟的线程执行完当前任务后执行接下来任务，复用已创建的线程，降低开销、控制最大并发数。

线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后还会循环获取工作队列中的任务来执行。

将任务派发给线程池时，会出现以下几种情况

1. 核心线程池未满，创建一个新的线程执行任务。
2. 如果核心线程池已满，工作队列未满，将线程存储在工作队列。
3. 如果工作队列已满，线程数小于最大线程数就创建一个新线程处理任务。
4. 如果超过大小线程数，按照拒绝策略来处理任务。

线程池参数

1. corePoolSize：常驻核心线程数。超过该值后如果线程空闲会被销毁。
2. maximumPoolSize：线程池能够容纳同时执行的线程最大数。
3. keepAliveTime：线程空闲时间，线程空闲时间达到该值后会被销毁，直到只剩下 corePoolSize 个线程为止，避免浪费内存资源。
4. workQueue：工作队列。
5. threadFactory：线程工厂，用来生产一组相同任务的线程。
6. handler：拒绝策略。有以下几种拒绝策略：

- AbortPolicy：丢弃任务并抛出异常
- CallerRunsPolicy： 重新尝试提交该任务

- DiscardOldestPolicy 抛弃队列里等待最久的任务并把当前任务加入队列
- DiscardPolicy 表示直接抛弃当前任务但不抛出异常。

线程池创建方法

1. newFixedThreadPool，创建固定大小的线程池。
2. newSingleThreadExecutor，使用单线程线程池。
3. newCachedThreadPool，maximumPoolSize 设置为 Integer 最大值，工作完成后会回收工作线程
4. newScheduledThreadPool：支持定期及周期性任务执行，不回收工作线程。
5. newWorkStealingPool：一个拥有多个任务队列的线程池。

简述Executor框架

Executor框架目的是将任务提交和任务如何运行分离开来的机制。用户不再需要从代码层考虑设计任务的提交运行，只需要调用Executor框架实现类的Execute方法就可以提交任务。产生线程池的函数ThreadPoolExecutor也是Executor的具体实现类。

简述Executor的继承关系

- Executor：一个接口，其定义了一个接收Runnable对象的方法executor，该方法接收一个Runable实例执行这个任务。
- ExecutorService：Executor的子类接口，其定义了一个接收Callable对象的方法，返回 Future 对象，同时提供execute方法。
- ScheduledExecutorService：ExecutorService的子类接口，支持定期执行任务。
- AbstractExecutorService：抽象类，提供 ExecutorService 执行方法的默认实现。
- Executors：实现ExecutorService接口的静态工厂类，提供了一系列工厂方法用于创建线程池。
- ThreadPoolExecutor：继承AbstractExecutorService，用于创建线程池。
- ForkJoinPool: 继承AbstractExecutorService，Fork 将大任务分叉为多个小任务，然后让小任务执行，Join 是获得小任务的结果，类似于map reduce。
- ThreadPoolExecutor：继承ThreadPoolExecutor，实现ScheduledExecutorService，用于创建带定时任务的线程池。

简述线程池的状态

- Running：能接受新提交的任务，也可以处理阻塞队列的任务。
- Shutdown：不再接受新提交的任务，但可以处理存量任务，线程池处于running时调用shutdown方法，会进入该状态。

- Stop：不接受新任务，不处理存量任务，调用shutdownnow进入该状态。
- Tidying：所有任务已经终止了，worker_count（有效线程数）为0。
- Terminated：线程池彻底终止。在tidying模式下调用terminated方法会进入该状态。

简述阻塞队列

阻塞队列是生产者消费者的实现具体组件之一。当阻塞队列为空时，从队列中获取元素的操作将会被阻塞，当阻塞队列满了，往队列添加元素的操作将会被阻塞。具体实现有：

- ArrayBlockingQueue：底层是由数组组成的有界阻塞队列。
- LinkedBlockingQueue：底层是由链表组成的有界阻塞队列。
- PriorityBlockingQueue：阻塞优先队列。
- DelayQueue：创建元素时可以指定多久才能从队列中获取当前元素
- SynchronousQueue：不存储元素的阻塞队列，每一个存储必须等待一个取出操作
- LinkedTransferQueue：与LinkedBlockingQueue相比多一个transfer方法，即如果当前有消费者正等待接收元素，可以把生产者传入的元素立刻传输给消费者。
- LinkedBlockingDeque：双向阻塞队列。

谈一谈ThreadLocal

ThreadLocal 是线程共享变量。ThreadLoacl 有一个静态内部类 ThreadLocalMap，其 Key 是 ThreadLocal 对象，值是 Entry 对象，ThreadLocalMap是每个线程私有的。

- set 给ThreadLocalMap设置值。
- get 获取ThreadLocalMap。
- remove 删除ThreadLocalMap类型的对象。

存在的问题

1. 对于线程池，由于线程池会重用 Thread 对象，因此与 Thread 绑定的 ThreadLocal 也会被重用，造成一系列问题。
2. 内存泄漏。由于 ThreadLocal 是弱引用，但 Entry 的 value 是强引用，因此当 ThreadLocal 被垃圾回收后，value 依旧不会被释放，产生内存泄漏。

聊聊你对java并发包下unsafe类的理解

对于 Java 语言，没有直接的指针组件，一般也不能使用偏移量对某块内存进行操作。这些操作相对来讲是安全（safe）的。

Java 有个类叫 `Unsafe` 类，这个类类使 Java 拥有了像 C 语言的指针一样操作内存空间的能力，同时也带来了指针的问题。这个类可以说是 Java 并发开发的基础。

JAVA中的乐观锁与CAS算法

对于乐观锁，开发者认为数据发送时发生并发冲突的概率不大，所以读操作前不上锁。

到了写操作时才会进行判断，数据在此期间是否被其他线程修改。如果发生修改，那就返回写入失败；如果没有被修改，那就执行修改操作，返回修改成功。

乐观锁一般都采用 Compare And Swap（CAS）算法进行实现。顾名思义，该算法涉及到了两个操作，比较（Compare）和交换（Swap）。

CAS 算法的思路如下：

1. 该算法认为不同线程对变量的操作时产生竞争的情况比较少。
2. 该算法的核心是对当前读取变量值 E 和内存中的变量旧值 V 进行比较。
3. 如果相等，就代表其他线程没有对该变量进行修改，就将变量值更新为新值 N。
4. 如果不等，就认为在读取值 E 到比较阶段，有其他线程对变量进行过修改，不进行任何操作。

ABA问题及解决方法简述

CAS 算法是基于值来做比较的，如果当前有两个线程，一个线程将变量值从 A 改为 B ，再由 B 改回为 A ，当前线程开始执行 CAS 算法时，就很容易认为值没有变化，误认为读取数据到执行 CAS 算法的期间，没有线程修改过数据。

juc 包提供了一个 AtomicStampedReference，即在原始的版本下加入版本号戳，解决 ABA 问题。

简述常见的Atomic类

在很多时候，我们需要的仅仅是一个简单的、高效的、线程安全的++或者--方案，使用synchronized关键字和lock固然可以实现，但代价比较大，此时用原子类更加方便。
基本数据类型的原子类有：

- AtomicInteger 原子更新整形
- AtomicLong 原子更新长整型
- AtomicBoolean 原子更新布尔类型

Atomic数组类型有：

- AtomicIntegerArray 原子更新整形数组里的元素
- AtomicLongArray 原子更新长整型数组里的元素
- AtomicReferenceArray 原子更新引用类型数组里的元素。

Atomic引用类型有

- AtomicReference 原子更新引用类型
- AtomicMarkableReference 原子更新带有标记位的引用类型，可以绑定一个 boolean 标记
- AtomicStampedReference 原子更新带有版本号的引用类型

FieldUpdater类型：

- AtomicIntegerFieldUpdater 原子更新整形字段的更新器
- AtomicLongFieldUpdater 原子更新长整形字段的更新器
- AtomicReferenceFieldUpdater 原子更新引用类型字段的更新器

简述Atomic类基本实现原理

以AtomicIntger 为例：
方法getAndIncrement：以原子方式将当前的值加1，具体实现为：

1. 在 for 死循环中取得 AtomicInteger 里存储的数值
2. 对 AtomicInteger 当前的值加 1
3. 调用 compareAndSet 方法进行原子更新
4. 先检查当前数值是否等于 expect
5. 如果等于则说明当前值没有被其他线程修改，则将值更新为 next，
6. 如果不是会更新失败返回 false，程序会进入 for 循环重新进行 compareAndSet 操作。

简述CountDownLatch

countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。
是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，调用countDown方法，计数器的值就减1，当计数器的值为0时，表示所有线程都执行完毕，然后在等待的线程就可以恢复工作了。
只能一次性使用，不能reset。

简述CyclicBarrier

CyclicBarrier 主要功能和countDownLatch类似，也是通过一个计数器，使一个线程等待其他线程各自执行完毕后再执行。但是其可以重复使用（reset）。

简述Semaphore

Semaphore即信号量。
Semaphore 的构造方法参数接收一个 int 值，设置一个计数器，表示可用的许可数量即最大并发数。使用 acquire 方法获得一个许可证，计数器减一，使用 release 方法归还许可，计数器加一。如果此时计数器值为0,线程进入休眠。

简述Exchanger

Exchanger类可用于两个线程之间交换信息。可简单地将Exchanger对象理解为一个包含两个格子的容器，通过exchanger方法可以向两个格子中填充信息。线程通过exchange 方法交换数据，第一个线程执行 exchange 方法后会阻塞等待第二个线程执行该方法。当两个线程都到达同步点时这两个线程就可以交换数据当两个格子中的均被填充时，该对象会自动将两个格子的信息交换，然后返回给线程，从而实现两个线程的信息交换。

简述ConcurrentHashMap

JDK7采用锁分段技术。首先将数据分成 Segment 数据段，然后给每一个数据段配一把锁，当一个线程占用锁访问其中一个段的数据时，其他段的数据也能被其他线程访问。

get 除读到空值不需要加锁。该方法先经过一次再散列，再用这个散列值通过散列运算定位到 Segment，最后通过散列算法定位到元素。
put 须加锁，首先定位到 Segment，然后进行插入操作，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置，然后将其放入数组。

JDK8的改进

1. 取消分段锁机制，采用CAS算法进行值的设置，如果CAS失败再使用 synchronized 加锁添加元素
2. 引入红黑树结构，当某个槽内的元素个数超过8且 Node数组 容量大于 64 时，链表转为红黑树。
3. 使用了更加优化的方式统计集合内的元素数量。

Synchronized底层实现原理

Java 对象底层都关联一个的 monitor，使用 synchronized 时 JVM 会根据使用环境找到对象的 monitor，根据 monitor 的状态进行加解锁的判断。如果成功加锁就成为该 monitor 的唯一持有者，

monitor 在被释放前不能再被其他线程获取。

synchronized在JVM编译后会产生monitorenter 和 monitorexit 这两个字节码指令，获取和释放 monitor。这两个字节码指令都需要一个引用类型的参数指明要锁定和解锁的对象，对于同步普通方法，锁是当前实例对象；对于静态同步方法，锁是当前类的 Class 对象；对于同步方法块，锁是 synchronized 括号里的对象。

执行 monitorenter 指令时，首先尝试获取对象锁。如果这个对象没有被锁定，或当前线程已经持有锁，就把锁的计数器加 1，执行 monitorexit 指令时会将锁计数器减 1。一旦计数器为 0 锁随即就被释放。

Synchronized关键词使用方法

1. 直接修饰某个实例方法
2. 直接修饰某个静态方法
3. 修饰代码块

简述java偏向锁

JDK 1.6 中提出了偏向锁的概念。该锁提出的原因是，开发者发现多数情况下锁并不存在竞争，一把锁往往是由同一个线程获得的。偏向锁并不会主动释放，这样每次偏向锁进入的时候都会判断该资源是否是偏向自己的，如果是偏向自己的则不需要进行额外的操作，直接可以进入同步操作。

其申请流程为：

1. 首先需要判断对象的 Mark Word 是否属于偏向模式，如果不属于，那就进入轻量级锁判断逻辑。否则继续下一步判断；
2. 判断目前请求锁的线程 ID 是否和偏向锁本身记录的线程 ID 一致。如果一致，继续下一步的判断，如果不一致，跳转到步骤4；
3. 判断是否需要重偏向。如果不用的话，直接获得偏向锁；
4. 利用 CAS 算法将对象的 Mark Word 进行更改，使线程 ID 部分换成本线程 ID。如果更换成功，则重偏向完成，获得偏向锁。如果失败，则说明有多线程竞争，升级为轻量级锁。

简述轻量级锁

轻量级锁是为了在没有竞争的前提下减少重量级锁出现并导致的性能消耗。

其申请流程为：

1. 如果同步对象没有被锁定，虚拟机将在当前线程的栈帧中建立一个锁记录空间，存储锁对象目前 Mark Word 的拷贝。

2. 虚拟机使用 CAS 尝试把对象的 Mark Word 更新为指向锁记录的指针
3. 如果更新成功即代表该线程拥有了锁，锁标志位将转变为 00，表示处于轻量级锁定状态。
4. 如果更新失败就意味着至少存在一条线程与当前线程竞争。虚拟机检查对象的 Mark Word 是否指向当前线程的栈帧
5. 如果指向当前线程的栈帧，说明当前线程已经拥有了锁，直接进入同步块继续执行
6. 如果不是则说明锁对象已经被其他线程抢占。
7. 如果出现两条以上线程争用同一个锁，轻量级锁就不再有效，将膨胀为重量级锁，锁标志状态变为 10，此时Mark Word 存储的就是指向重量级锁的指针，后面等待锁的线程也必须阻塞。

简述锁优化策略

即自适应自旋、锁消除、锁粗化、锁升级等策略偏。

简述java的自旋锁

线程获取锁失败后，可以采用这样的策略，可以不放弃 CPU ，不停的重试内重试，这种操作也称为自旋锁。

简述自适应自旋锁

自适应自旋锁自旋次数不再人为设定，通常由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。

简述锁粗化

锁粗化的思想就是扩大加锁范围，避免反复的加锁和解锁。

简述锁消除

锁消除是一种更为彻底的优化，在编译时，java编译器对运行上下文进行扫描，去除不可能存在共享资源竞争的锁。

简述Lock与ReentrantLock

Lock 接是 java并发包的顶层接口。

可重入锁 ReentrantLock 是 Lock 最常见的实现，与 synchronized 一样可重入。ReentrantLock 在默认情况下是非公平的，可以通过构造方法指定公平锁。一旦使用了公平锁，性能会下降。

简述AQS

AQS（AbstractQuenedSynchronizer）抽象的队列式同步器。
AQS是将每一条请求共享资源的线程封装成一个锁队列的一个结点（Node），来实现锁的分配。
AQS是用来构建锁或其他同步组件的基础框架，它使用一个 volatile int state 变量作为共享资源，如果线程获取资源失败，则进入同步队列等待；如果获取成功就执行临界区代码，释放资源时会通知同步队列中的等待线程。

子类通过继承同步器并实现它的抽象方法getState、setState 和 compareAndSetState对同步状态进行更改。

AQS获取独占锁/释放独占锁原理

获取：（acquire）

1. 调用 tryAcquire 方法安全地获取线程同步状态，获取失败的线程会被构造同步节点并通过 addWaiter 方法加入到同步队列的尾部，在队列中自旋。
2. 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。

释放：（release）

1. 调用 tryRelease 方法释放同步状态
2. 调用 unparkSuccessor 方法唤醒头节点的后继节点，使后继节点重新尝试获取同步状态。

AQS获取共享锁/释放共享锁原理

获取锁（acquireShared）

1. 调用 tryAcquireShared 方法尝试获取同步状态，返回值不小于 0 表示能获取同步状态。

释放（releaseShared）

1. 释放，并唤醒后续处于等待状态的节点。

我是小牛，非科班转行的微软程序员新人一枚。

我会在这里分享一下自己的转行学习路线、个人经历、内推信息等等！欢迎大家转载我的原创文章，可在公众号下留言！