synchronized锁静态对象和实例对象的区别

锁的是类的class对象，属于类级别锁，所有实例共享同一把锁

锁的是当前实例，属于对象级别锁，同一实例的同步方法会互斥，不同实例的同步方法互不影响

可重入锁的工作原理

用一个计数器来记录锁被获取的次数，获取锁一次计数器+1，释放锁一次计数器-1，当计数器为0时，表示锁可用

synchronized原理

使用 synchronized 给对象上锁，该对象头的Mark Word 会设置一个指向 Monitor 对象的指针。第一个线程执行 synchronized(obj)，会将 Monitor 的所有者 Owner 设置为该线程。第二个线程再去执行的时候，就会进入EntryList。

synchronized锁升级

当一个线程获取到锁后，这把锁就是偏向锁。如果有第二个线程来竞争锁，偏向锁就会升级为轻量级锁，轻量级锁底层是通过自旋实现的，不会阻塞。如果自旋次数过多。则会升级为重量级锁，导致线程线程阻塞。（阻塞和唤醒需要操作系统去进行，耗时间）

AtomicInteger的底层原理

底层思想就是用CAS原理，保证线程安全。CAS即比较并交换，是一种非阻塞式锁，也称为乐观锁，在无锁的情况下保证线程操作共享资源的原子性。

工作原理：CAS有3个操作数，内存值，旧的预期值，要修改的新值。当预期值和内存值相同时，将内存值修改为新值，否则，会采用自旋的方式再次尝试完成更新，当自旋次数达到一定值时就更新失败。

创建线程池的方式

Executors.newFixedThreadPool：创建一个固定大小的线程池，可控制并发的线程数，超出的线程会在队列中等待；

Executors.newCachedThreadPool：创建一个可缓存的线程池，若线程数超过处理所需，缓存一段时间后会回收，若线程数不够，则新建线程；

Executors.newSingleThreadExecutor：创建单个线程数的线程池，它可以保证先进先出的执行顺序；

Executors.newScheduledThreadPool：创建一个可以执行延迟任务的线程池；

Executors.newSingleThreadScheduledExecutor：创建一个单线程的可以执行延迟任务的线程池；

Executors.newWorkStealingPool：创建一个抢占式执行的线程池（任务执行顺序不确定）【JDK 1.8 添加】。

ThreadPoolExecutor：最原始的创建线程池的方式，它包含了 7 个参数可供设置，后面会详细讲。

synchronized 和 volatile 的区别

volatile可以修饰变量；synchronized 可以修饰类、方法、变量

volatile 可以保证变量的修改可见性，不可以保证原子性；

synchronized 可以保证变量的修改可见性和原子性。

volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞

synchronized 和 Lock 区别

synchronized是关键字，Lock接口

synchronized 会自动加锁和释放锁，Lock需要手动调用lock和unlock方法加锁和释放锁

synchronized只支持非公平锁，Lock支持公平锁和非公平锁

synchronized在低竞争情况下性能较好，但在高竞争情况下性能可能下降。

Lock在高竞争情况下通常性能更好

synchronized 和 ReentrantLock 区别

synchronized 用在方法和代码块上，而 ReentrantLock 只能用在代码块上

synchronized 会自动加锁和释放锁，ReentrantLock 需要手动调用lock和unlock方法加锁和释放锁

synchronized是非公平锁；ReentrantLock可指定非公平锁与公平锁

sleep() 和 wait() 有什么区别

sleep是Thread类的方法，wait是Object类的方法

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步代码块sychronized

sleep可以在任何地方使用，wait，notify和notifyAll只能在同步代码块里面使用

sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

volatile的原理

volatile 的底层实现原理是内存屏障

保证可见性

写屏障保证在该屏障之前，对共享变量的改动，都同步到主存当中

读屏障保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据

保证有序性

写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后

读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前

并行和并发有什么区别

**并行是指两个或多个事件在同一时刻发生；并发是指两个或多个事件在同一时间间隔发生。**

线程和进程的区别

**进程是程序的一次执行过程，作为资源分配的最小单位。**

**一个进程之内可以分为一到多个线程，一个线程就是一个指令流，交给CPU执行，作为最小调度单位。**

线程有几种，在什么情况下应用

用户线程，在前台运行的线程都是用户线程，例如主线程

守护线程一般为后台运行的线程，守护线程是用来为用户线程服务的。例如JVM中的线程都为守护线程，典型的有GC线程

一旦所有用户线程都结束，守护线程会随 JVM 一起结束。而守护线程不会影响 JVM 的退出。

创建线程有哪几种方式？

继承 Thread 类

（1）定义Thread类的子类，重写run方法

（2）创建对象并调用start()方法启动线程

实现 Runnable 接口

（1）定义runnable接口的实现类，重写run方法

（2）创建实现类对象并传入Thread的构造方法中

（3）创建对象并调用start()方法启动线程

实现 Callable 接口

（1）定义Callable接口的实现类，重写call()方法

（2）创建实现类对象，并传入FutureTask构造方法中，创建FutureTask对象

（3）FutureTask对象传入Thread的构造方法中，创建Thread对象。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

使用 Executors 工具类创建线程池

runnable 和 callable 区别

**Runnable接口中的run()方法的返回值是void。**

**Callable接口中的call()方法是有返回值的，是一个泛型，和FutureTask配合可以用来获取异步执行的结果。**

**call方法可以抛出异常，run方法不可以**

线程的 run()和 start()区别

调用 start 方法可以启动线程并使线程进入就绪状态，而 run 方法只是thread 的一个普通方法调用，还是在主线程里执行

线程的生命周期？线程有几种状态

新建状态（New）：新创建了一个线程对象。

就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start方法。该状态的线程位于可运行线程池中，等待获取CPU的使用权。

运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

死亡状态（Dead）：线程执行完了或者因异常退出了run方法，该线程结束生命周期。

ThreadLocal

ThreadLocal提供了线程局部变量，每个Thread内部维护一张ThreadLocalMap，threadLocal对象作为key，threadLocal存的值作为这张map的值，实现多个独立的副本，避免资源共享。

使用完ThreadLocal方法后，最好手动调用remove()方法。否则，内存泄漏，复用线程拿到之前的旧值

如何停止一个正在运行的线程

使用interrupt方法中断线程。使用stop方法强行终止。

两阶段终止模式（使用退出标志，使线程正常退出，也就是当run方法完成后线程终止）

线程池有什么优点？

复用线程，减少对象创建销毁的开销。

提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等到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，如果无限制的创建，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的分配，调优和监控。

线程池中 submit() 和 execute() 方法有什么区别？

接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行Runnable 和 Callable 类型的任务。

返回值：submit()方法可以返回持有计算结果的Future对象，而execute()没有

线程池有几种

FixedThreadPool，核心线程数和最大线程数是一样的

CachedThreadPool，线程数是可达到整型最大值，Interger. MAX\_VALUE

ScheduledThreadPool，支持周期性执行任务。比如每隔 10 秒钟执行一次任务

SingleThreadExecutor，一个线程

线程池参数

ThreadPoolExecutor

corePoolSize ：核心线程数

maximumPoolSize ：最大线程数

keepAliveTime：救急线程的生存时间。多余的空闲线程存活时间，当空间时间达到keepAliveTime值时，多余的线程会被销毁直到只剩下corePoolSize个线程为止。

unit ：时间单位

workQueue：阻塞队列，被提交但尚未被执行的任务。

threadFactory：线程工厂-可以为线程创建时起个好名字

handler ：拒绝策略，表示当线程队列满了并且工作线程大于等于线程池的最大显示数(maxnumPoolSize)时如何来拒绝请求执行的runnable的策略

线程池运行流程

线程池创建，准备好 core 数量的核心线程，接受任务。core 满了，就将新任务放入阻塞队列中，空闲的核心线程就会去阻塞队列获取任务执行。阻塞队列也满了，就直接开新线程去执行，最大只能开到 max 指定的数量。max满了就用拒绝策略。max 都执行好了，Max-core 数量空闲的线程会在 keepAliveTime 指定的时间后自动销毁。

1.在创建了线程池之后，等待提交过来的任务请求

2.当线程池调用execute()方法添加一个请求任务时，会进行判断：

2.1线程数量小于核心线程数，创建核心线程执行任务

2.2 否则，将这个任务放入阻塞队列

2.3 队列满了，并且线程数量小于最大线程数，创建救急线程执行任务

2.4 队列满了，并且线程数量大于最大线程数，启动拒绝策略

空闲的救急线程会在 keepAliveTime 时间后自动销毁

3. 当一个线程完成任务的时候，它会从队列中取下一个任务来执行

4. 当一个救急线程无事可做超过一定时间（keepAliveTime）时：线程会判断如果当前运行的线程数大于corePoolSize，那么这个线程就会被停掉。

简单来说就是优先核心线程，其次阻塞队列，最后救急线程。

(核心线程：只要线程池不关闭，就不会被销毁)

线程池中的线程分为核心线程和救急线程。核心线程并不是在线程池创建之初就建立线程，而是使用者每提交一次任务就创建一个新线程，不关闭，直到达到核心线程数。

线程池饱和策略

AbortPolicy:直接抛出异常，默认策略

DiscardPolicy: 直接丢弃任务

DiscardOldestPolicy:丢弃下一个将要被执行的任务(最旧任务)

CallerRunsPolicy: 将任务回退到调用者，主线程中执行任务，（用调用者所在的线程来执行任务）

线程池都有哪些状态？

RUNNING：这是最正常的状态，接受新任务，处理队列中的任务。

SHUTDOWN：不接受新任务，处理队列中的任务

STOP：不接受新任务，不再处理队列中的任务，中断正在执行任务的线程。

TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为TIDYING 状态时，会执行钩子方法 terminated()。

TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

线程池参数设置

线程池的核心线程数

CPU密集型：核心线程数 = CPU核数 + 1

IO密集型：核心线程数 = CPU核数 \* 2

最大线程数：最大线程数一般设置为 2 \* 核心线程数

阻塞队列大小：如果任务是 CPU 密集型，即计算任务比较多，可以将队列大小设置为 0 或者 1，这样可以让线程池及时处理任务，避免任务堆积。如果任务是 IO 密集型，即网络请求比较多，可以设置队列大小为 2 \* 核心线程数，这样可以缓存一些请求，避免线程池因为等待 IO 密集型任务而阻塞

CPU密集型应用程序通常需要大量的CPU运算能力

IO密集型应用程序通常需要大量的IO操作

线程工厂的作用

作用就是设置线程名称

守护线程和用户线程有什么区别呢？

一旦所有用户线程都结束，守护线程会随 JVM 一起结束。而守护线程不会影响 JVM 的退出。

什么是线程死锁

死锁是指多个线程竞争资源而造成阻塞

形成死锁的四个必要条件是什么

互斥条件：线程(进程)对于所分配到的资源具有排它性，即一个资源只能被一个线程占用，直到被该线程(进程)释放

请求与保持条件：一个线程因请求被占用资源而发生阻塞时，对已获得的资源保持不放

不剥夺条件：线程(进程)已获得的资源在末使用完之前不能被其他线程剥夺，只有自己使用完毕后才释放资源。

循环等待条件：当发生死锁时，所等待的线程形成一个环路

解决死锁的方法

**死锁预防**

通过破坏四个必要条件中的至少一个

破坏互斥：允许资源共享（某些资源无法做到，如打印机）。

破坏持有并等待：要求进程一次性申请所有所需资源（资源利用率低）。

破坏不可抢占：允许系统强制回收资源（可能导致进程执行中断）。

破坏循环等待：强制资源按顺序申请（如所有进程必须先申请资源A再申请资源B）。

**死锁避免**

动态检查资源分配是否会导致系统进入不安全状态：

- \*\*银行家算法\*\*：通过模拟资源分配，判断是否允许当前请求。

**死锁检测与恢复**

检测：定期构建资源分配图，检查是否存在环路。

恢复：终止进程：强制终止部分进程以释放资源（如终止优先级最低的进程）。

资源回滚：撤销某些进程的操作，释放资源并恢复到安全状态。

实际应用中的处理

**加锁顺序（线程按照一定的顺序加锁） 按固定顺序获取锁（避免循环等待）**

**加锁时限（线程获取锁的时候加上时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）** 避免无限等待

使用死锁检测机制

volatile作用

volatile能禁止进行指令重排序，不能保证线程安全，保证可见性

并发编程三要素

并发编程三要素（线程的安全性问题体现在）：

原子性：要么所有操作都做，要么所有操作都不做

可见性：一个线程对共享变量修改后，其他线程可以立即看到修改的结果。在一个线程中对某个变量进行赋值，然后在另外一个线程中读取该变量的值，读取到的可能仍然是以前的值。（synchronized,volatile）java内存模型中，主内存的数据需要在工作内存中有副本，才能被工作线程使用。 避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值。volatile

有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）

为什么要使用并发编程

充分利用多核CPU的计算能力，方便进行业务拆分

并发编程有什么缺点

并发编程可能会遇到很多问题，比如内存泄漏、线程安全、死锁等问题

什么是多线程，多线程的优劣？

多线程：多线程是指在一个程序中可以同时运行多个线程来执行不同的任务。

多线程的好处：

可以提高 CPU 的利用率

多线程的劣势：

线程越多占用内存也越多

线程之间对共享资源的访问会相互影响

什么是上下文切换

线程切换时，当前线程任务从保存到再加载的过程就是一次上下文切换

什么是 FutureTask

FutureTask是Future接口的实现，FutureTask可以传入一个 Callable 的实现类，调用get方法可以对异步运算的结果进行等待获取。可以将其提交给 ExecutorService 进行执行

一个线程运行时发生异常会怎样

如果异常没有被捕获该线程将会停止执行。

什么是阻塞式方法？

阻塞式方法是指程序会一直等待该方法完成

sleep()方法和yield()方法区别

线程执行sleep()方法后转入阻塞状态，而执行yield()方法后转入就绪状态； sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常

Java中用到的线程调度算法是什么

Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU

notify() 和 notifyAll() 有什么区别

notifyAll() 会唤醒所有的线程，notify() 只会唤醒一个线程

如何在两个线程间共享数据？

在两个线程间共享变量即可实现共享。

一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。

juc常见的类及其作用

ThreadPoolExecutor: 线程池实现，管理线程的生命周期和任务执行。

ScheduledThreadPoolExecutor: 支持调度的线程池。

ConcurrentHashMap: 线程安全的哈希表

CountDownLatch: 允许线程等待一组操作完成。

AtomicInteger: 支持原子操作的整数。

AtomicLong: 支持原子操作的长整数。

AtomicReference: 支持原子操作的对象引用。

AtomicBoolean: 支持原子操作的布尔值。

ReentrantLock: 可重入锁，提供比`synchronized`更灵活的锁定机制。

ReentrantReadWriteLock: 可重入的读写锁，支持读操作的并发性。

Callable: 有返回值的任务，类似于Runnable，但可以返回结果和抛出异常。

FutureTask: 可取消的异步任务，实现了`Runnable`和`Future`。

CompletableFuture: 支持异步编程，允许组合多个异步操作。

Semaphore: 控制同时访问特定资源的线程数。

Future: 表示异步计算的结果。

线程同步和通信

同步：通过 `synchronized`、`ReentrantLock`、`ReadWriteLock` 等机制确保线程安全。

通信：通过 `wait()`/`notify()`、`Condition`（Condition类的await()/signal()/signalAll()）、`BlockingQueue` （线程安全的队列）等机制实现线程间的协调。

这些工具和机制帮助开发者在多线程环境中实现线程安全和高效通信。

Completablefuture用法

runAsync是创建没有返回值的异步任务

supplyAsync是创建带有返回值的异步任务

使用 allOf，等待多个 CompletableFuture 都完成，调用get方法阻塞等待