2020 面试题总结——多线程篇

1、多线程的几种实现方式，什么是线程安全。

Java 多线程实现方式主要有四种：

- 继承 Thread 类，启动线程的唯一方法就是通过 Thread 类的 start()方法，实例后调用 start()方法启动。
- 实现 Runable 接口。
- 实现 Callable 接口。
- 通过 FutureTask 包装器。

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

2、volatile 的原理，作用，能代替锁吗?

volatile 是轻量级的 synchronized，它让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。如果一个字段被声明为 volatile，Java 线程内存模型确保所有线程看到这个变量的值是一致的。具有几条特性。

- volatile **无法保证复合操作的原子性**。Java 只保证了基本数据类型变量的赋值操作才是原子性的，当然，可以用过锁、synchronized 来确保原子性。其实严格的说，**对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++这种复合操作是不具有原子性。**
- volatile 可以保证**可见性**，当一个变量被 volatile 修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会自己被更新到主内存中，当其他线程读取共享变量时，他会直接从主内存中读取。当然，synchronized 和锁能都保证可见性。
- volatile 可以保证**有序性**，禁止指令重排序。

综上，volatile 可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在 JVM 底层 volatile 是采用"内存 屏障"来实现的。

在使用场景中，轻量级锁 volatile 是不能取代 synchronized，但也可以在有限的一些情形下可以用 volatile 变量代替锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

- 对变量的写操作不依赖于当前值。比如使 volatile 变量不能用作线程安全计数器，不能 x++。

- 该变量没有包含在具有其他变量的不变式中。这块理解像是 volatile 变量不能用于约束条件。

参考来源：深入分析 volatile 的实现原理

3、sleep 和 wait 的区别。

主要有四点区别：

- sleep()方法是 Thread 类的静态方法，wait()方法是 Object 超类的成员方法。
- sleep()方法导致程序暂停指定的时间，让出 cpu 给其他线程，但是它的监控状态依然保持着，当指定的时间到了又会自动恢复运行状态。在调用 sleep()方法的过程中，线程是不会释放锁的。而调用 wait()方法会释放对象锁，只有当此对象调用 notify()方法后才会唤醒线程。
- sleep()方法可以在任何地方使用，wait()方法只能在同步方法和同步代码块中配合 synchronized 使用。
- sleep()方法需要抛出异常，wait()方法不需要。

4、sleep(0)的意义。

Thread.Sleep(0) 并非是真的要线程挂起 0 毫秒，意义在于这次调用 Thread.Sleep(0)的当前线程确实的被冻结了一下，让其他线程有机会优先执行。Thread.Sleep(0) 是你的线程暂时放弃 cpu，也就是释放一些未用的时间片给其他线程或进程使用，就相当于一个让位动作。

在线程没退出之前，线程有三个状态，就绪态，运行态，等待态。sleep(n)之所以在 n 秒内不会参与 CPU 竞争，是因为，当线程调用 sleep(n)的时候，线程是由运行态转入等待态，线程被放入等待队列中，等待定时器 n 秒后的中断事件，当到达 n 秒计时后，线程才重新由等待态转入就绪态，被放入就绪队列中，等待队列中的线程是不参与 cpu 竞争的，只有就绪队列中的线程才会参与 cpu 竞争，所谓的 cpu 调度，就是根据一定的算法（优先级，FIFO 等），从就绪队列中选择一个线程来分配 cpu 时间。

而 sleep(0)之所以马上回去参与 cpu 竞争，是因为调用 sleep(0)后，因为 0 的原因，线程直接回到就绪队列，而非进入等待队列，只要进入就绪队列，那么它就参与 cpu 竞争。

5、Lock 和 Synchronized 的区别。

下面主要以可重入锁 ReentrantLock 为例。

两者相同点是：

- 都实现了多线程同步和内存可见性语义。

- 都是可重入锁。

两者不同点是：

- 同步实现机制不同，synchronized 是通过 Java 对象头锁标记和 Monitor 对象实现同步；而 ReentrantLock 则是通过 CAS、AQS 和 LockSupport（用于阻塞和解除阻塞）实现同步。
- 可见性实现机制不同，synchronized 依赖 JVM 内存模型保证包含共享变量的多线程内存可见性；而 ReentrantLock 是通过 AQS 中的 volatile state 状态来保证包含共享变量的多线程内存可见性。
- 使用方式不同，synchronized 可以修饰实例方法（锁住实例对象）、静态方法（锁住类对象）、代码块（显示指定锁对象）；ReentrantLock 显示调用 tryLock 和 lock 方法，需要在 finally 块中释放锁。
- 功能丰富程度不同，synchronized 不可设置等待时间、不可被中断（interrupted）；ReentrantLock 提供有限时间等候锁（设置过期时间）、可中断锁（lockInterruptibly）、condition（提供 await、condition（提供 await、signal 等方法）等丰富功能。
- 锁类型不同，synchronized 只支持公平锁；而 ReentrantLock 提供公平锁和非公平锁实现，当然，在大部分情况下，非公平锁是高效的选择。

在 synchronized 优化以前，它的性能是比 ReenTrantLock 差很多的，但是自从 synchronized 引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用 synchronized 。

6、synchronized 的原理是什么，一般用在什么地方（比如加载静态方法和非静态方法的区别）

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

- 普通同步方法，锁的是当前实例对象。
- 静态同步方法，锁的是当前类的 class 对象。
- 同步方法块，锁的是括号里面的对象。

同步方法块时使用 monitorenter（插入到同步代码块开始的位置）和 monitorexit（插入到方法块结束处和异常处）指令实现的，而对于**同步方法**则是依靠方法修饰符上的 ACC_SYNCHRONIZED 来完成的。无论采用哪种方式，其本质都是对一个对象的监视器（monitor）进行获取，而这个获取过程是排他的，也就是同一时刻只能有一个线程获取到由 synchronized 所保护对象的监视器。任何一个对象都有自己的监视器，

当这个对象由方法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方法块和同步方法，而没有获取到监视器（执行该方法）的线程将会被阻塞在同步块和同步方法的入口出，进入 BLOCKED 状态。

Java 对象头和 monitor 是实现 synchronized 的基础。Java 对象头中主要包括两部分数据，Mark Word（标记字段，用处存储对象自身的运行时数据，比如哈希码、GC 分代年龄、锁状态标志、偏向线程 ID 等）和 Klass Pointer（类型指针）。其中 Klass Point 是对象指向它的类元数据的指针，虚拟机通过这个指针来确定对象是哪个类的实例。Mark Word 用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键。

之前 synchronized 是一个重量级锁，相对于 Lock，会显得笨重。

jdk1.6 对 synchronized 进行了各种优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

锁的主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

- 偏向锁：当一个线程访问同步块并获取锁时，会在对象头（具体是 Mark Word）和栈桢中的锁记录里存储锁偏向的线程 ID，以后该线程在进入和退出同步块时不需要进行 CAS 操作来加锁和解锁，只需简单地测试一下对象头的 Mark Word 里是否存储着指向当前线程的偏向锁。如果其他线程竞争该偏向锁，则尝试使用 CAS 将对象头的偏向锁指向当前咸亨。
- 轻量级锁：线程在执行同步块之前，JVM 会先在当前线程的栈桢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，官方成为 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，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。

参考来源（强烈建议去看）：

《Java 并发编程的艺术》-2.2 节 synchronized 的实现原理与应用

4.3.1 节-volatile 和 synchronized 关键字

深入分析 synchronized 的实现原理

【死磕 Java 并发】----- synchronized 的锁膨胀过程

7、用过哪些原子类，他们的原理是什么。

Java 中有 13 个原子操作类，都属于 Atomic 包，基本都是使用 Unsafe 实现的包装类，再底层就都是 CAS 操作实现的。

8、用过线程池吗？如果用过，请说明原理，并说说 newCache 和 newFixed 有什么区别，构造函数的各个参数的含义是什么，比如 coreSize、maxSize 等。

Java 的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来 3 个好处。

- 线程复用：线程池中的线程是可以复用的，省去了创建、销毁线程的开销，提高了资源利用率（创建、销毁等操作都是要消耗系统资源的）和响应速度（任务提交过来线程已存在就不用等待线程创建了）；
- 合理利用利用资源：通过调整线程池大小，可以让所有处理器尽量保持忙碌，并且又能放置过多线程产生过多竞争浪费资源。
- 提高线程的可管理型。使用线程池可以进行统一分配、调优和监控。

当提交一个新任务到之后，线程池的处理流程如下：

- 线程池先判断其核心线程池里的线程（corePoolSize，基本线程数量）是否都在执行任务，如果不是，则创建一个新的工作线程。如果核心线程池的线程都在执行任务，则进入下个流程。
- 线程池判断工作队列是否已满，如果工作序列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
- 线程池判断线程池的线程（maximumPoolSize，最大线程数量）是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。其中默认的饱和策略是 AbortPolicy，表示无法处理新任务时抛出异常。

Java 中通过 ThreadPoolExecutor 来创建线程池。

```
new ThreadPoolExecutor(int corePoolSize,//线程池的基本大小
                       int maximumPoolSize,//线程池最大数量
                       long keepAliveTime,//线程活动保持时间，线程池的工作线程空间后，保持存活的时间
                       TimeUnit unit,//线程活动保持时间的单位
                       BlockingQueue<Runnable> workQueue,//任务队列，用于保存等待执行的任务的阻塞队列
                       ThreadFactory threadFactory,//用于设置创建线程的工厂
                       RejectedExecutionHandler handler)//饱和策略
```

JDK 预定义了四种线程池：

- newFixedThreadPool：固定大小线程池，创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列（无界）中等待。

- newCachedThreadPool：创建一个可缓存的线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。
- newScheduledThreadPool：创建一个定长线程池，支持定时及周期性任务执行。
- newSingleThreadExecutor：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

对于高并发，可以将 SynchronousQueue 作为参数，使 maximumPoolSize 发挥作用，以防止线程被无限制的分配，同时可以通过提高 maximumPoolSize 来提高系统吞吐量，另外，也自定义一个 RejectedExecutionHandler，当线程数超过 maximumPoolSize 时进行处理，处理方式为隔一段时间检查线程池是否可以执行新 Task，如果可以把拒绝的 Task 重新放入到线程池，检查的时间依赖 keepAliveTime 的大小。

9、线程池的关闭方式有几种，各自的区别是什么。

可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池。他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中止。

10、spring 的 controller 是单例还是多例，怎么保证并发的安全。

spring bean 作用域有五种：

- singleton：单例模式，当 spring 创建 applicationContext 容器的时候，spring 会欲初始化所有的该作用域实例，加上 lazy-init 就可以避免预处理；
- prototype：原型模式，每次通过 getBean 获取该 bean 就会新产生一个实例，创建后 spring 将不再对其管理；

====下面是在 web 项目下才用到的===

- request：搞 web 的大家都应该明白 request 的域了吧，就是每次请求都新产生一个实例，和 prototype 不同就是创建后，接下来的管理，spring 依然在监听。
- session：每次会话，同上。
- global session：全局的 web 域，类似于 servlet 中的 application。

spring 中的 controller 默认是单例，也就是 singleton 模式了。

所以如果 controller 中有一个私有变量 a，所有请求到同一个 controller 时，使用的 a 变量都是共用的，即若是某个请求修改了这个变量 a，则，在别的请求中能够读到这个修改的内容。

为了保证并发的安全，常见有两种解决方法。

- 在 controller 中使用 ThreadLocal 变量。
- 在 spring 配置文件 Controller 中声明为 scope="prototype"，每次都创建新的 controller，不再使用单例模式。

另外，Servlet 也不是线程安全的，Servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

11、ThreadLocal 用过吗，用途是什么，原理是什么，需要注意什么。

ThreadLocal，即线程变量，是一个以 ThreadLocal 对象为键，任意对象为值的存储结构。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的一个值。

- Thread 类中有一个成员变量属于 ThreadLocalMap 类（一个定义在 ThreadLocal 类中的内部类），它是一个 map，它的 key 是 ThreadLocal 实例对象。
- 当为 ThreadLocal 类的对象 set 值时，首先获取当前线程的 ThreadLocalMap 变量，然后以 ThreadLocal 类的对象为 key，设定 value。get 值时则类似。
- ThreadLocal 变量的活动范围为某线程，是该线程"专有的，独自霸占"的，对该变量的所有操作均由该线程完成！也就是说，ThreadLocal 不是用来解决共享对象的多线程访问的竞争问题的，因为 ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。当线程终止后，这些值会作为垃圾回收。
- 由 ThreadLocal 的工作原理决定了：每个线程独自拥有一个变量，并非是共享的。

需要注意的是，每次 set/get 值，不直接用线程 id 来作为 ThreadLocalMap 的 key，因为若直接用线程 id 当作 key，无法区分放入 ThreadLocalMap 中的多个 value。所以是使用 ThreadLocal 作为 key，因为每一个 ThreadLocal 对象都可以由 threadLocalHashCode 属性（final 修饰，每次实例创建后就不会更改了）唯一区分或者说每一个 ThreadLocal 对象都可以由这个对象的名字唯一区分，所以可以用不同的 ThreadLocal 作为 key，区分不同 value。

如何保证两个同时实例化的 ThreadLocal 对象有不同的 threadLocalHashCode 属性：在 ThreadLocal 类中，还包含了一个 static 修饰的 AtomicInteger（提供原子操作的 Integer 类）类变量（nextHashCode）和一个 static final 修饰的常

量（作为两个相邻 nextHashCode 的差值）。由于 nextHashCode 是类变量，所以每一次创建 ThreadLocal 对象都可以保证 nextHashCode 被更新到新的值，并且下一次调用 ThreadLocal 类这个被更新的值仍然可用，同时 AtomicInteger 保证了 nextHashCode 自增的原子性。

TreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

- 在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。
- 而 ThreadLocal 则从另一个角度来解决多线程的并发访问。ThreadLocal 会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。
- 概括起来说，对于多线程资源共享的问题，同步机制采用了**“以时间换空间”**的方式，而 ThreadLocal 采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

在默认单例的 Spring bean 中，将一些非线程安全的变量以 ThreadLocal 存放，在同一次请求响应的调用线程中，所有关联的对象引用到的都是同一个变量。

需要注意的是，TheadLocalMap 作为 hash 表的一种实现方式，是通过**开放寻址法**来解决哈希冲突，这点不同于 HashMap。开放寻址法的核心是如何出现了散列冲突，就重新探测一个空闲位置，将其插入。当我们往散列表插入数据时，如果某个数据经过散列函数散列之后，存储位置已经被占用了，我们就从当前位置开始，以此往后查找，看是否有空闲位置，直到找到为止。

另外，ThreadLocalMap 的初始长度为 16。当集合中 size 数量大于规定长度的 1/2（）时，则执行 resize()操作，扩容到原来两倍。具体代码如下：

```
private void rehash() {
    expungeStaleEntries();
    // Use lower threshold for doubling to avoid hysteresis
    if (size >= threshold - threshold / 4) //相当于 size>= 3/4 * threshold，
其中，threshold=2/3 * len，所以 size>=1/2 * len 时，即发生扩容
        resize();
}
```

12、如何实现一个并发安全的链表。

有以下几种方式：

- 采用粗粒度锁，完全锁住链表。

- 采用细粒度锁，只锁住需要修改的节点。
- 利用 CAS 来修改节点。

13、有哪些无锁的数据结构，怎么做。

要实现一个线程安全的队列有两种方式：阻塞和非阻塞。阻塞队列无非就是锁的应用，而非阻塞则是 CAS 算法的应用（无锁）。比较常见的是 ConcurrentLinkedQueue，这是一个基于链表节点的无边界的线程安全队列，它采用 FIFO 原则对元素进行排序，采用“wait-free”算法（即 CAS 算法）来实现，这是单向链表。

14、讲讲 java 同步机制的 wait 和 notify。

wait 与 notify 是 Java 同步机制中的重要组成部分。结合与 synchronized 关键字使用，可以建立很多优秀的同步模型，例如生产者-消费者模型。需要注意几点：

- wait()、notify()、notifyAll()方法不属于 Thread 类，而是属于 Object 基础类，因为 Java 为每个 Object 对象都分配了一个 monitor。
- 当需要调用以上的方法的时候，一定要对竞争资源进行加锁，如果不加锁的话，则会报 IllegalMonitorStateException 异常。
- 当想要调用 wait( )进行线程等待时，必须要取得这个锁对象的控制权（对象监视器），一般是放到 synchronized(obj)代码中。
- notify( )方法只会通知等待队列中的第一个相关线程（不会通知优先级比较高的线程）
- notifyAll( )通知所有等待该竞争资源的线程（也不会按照线程的优先级来执行）

15、countdowlatch 和 cyclicbarrier 的内部原理和用法，以及相互之间的差别(比如 countdownlatch 的 await 方法和是怎么实现的)。

CyclicBarrier：允许一组线程互相等待，直到到达某个公共屏障点，才会进行后续任务，内部是使用重入锁 ReentrantLock 和 Condition。

CountDownLatch：在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待，内部依赖 Sync 实现，而 Sync 继承 AQS。

两者的区别：

- CountDownLatch 的作用是允许 1 或 N 个线程等待其他线程完成执行；而 CyclicBarrier 则是允许 N 个线程相互等待。
- CountDownLatch 的计数器无法被重置；CyclicBarrier 的计数器可以被重置后使用，因此它被称为是循环的 barrier。

await 方法对比：

CyclicBarrier 中 await()方法内部调用 dowait 方法，每当进来一个线程，则对设定的总量 count--，直到为 0，才会继续执行后续的任务 Runnable。
CountDownLatch 中 await()方法让当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。其中计数器是用 AQS 的状态值来表示的。如果计数器值不为零，则会调用 AQS 的自选方法尝试一直去获取同步状态。

参考链接：

【死磕 Java 并发】-----J.U.C 之并发工具类：CountDownLatch

【死磕 Java 并发】—- J.U.C 之并发工具类：CyclicBarrier

16、对 AbstractQueuedSynchronizer 了解多少，讲讲加锁和解锁的流程，独占锁和公平锁加锁有什么不同。

AQS，即队列同步器，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。

同步器的主要使用方法是继承，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。AQS 使用一个 int 类型的成员变量 state 来表示同步状态，当 state>0 时表示已经获取了锁，当 state = 0 时表示释放了锁。它提供了三个方法（getState()、setState(int newState)、compareAndSetState(int expect,int update)）来对同步状态 state 进行操作，当然 AQS 可以确保对 state 的操作是安全的。

AQS 通过内置的 FIFO 同步队列来完成线程获取资源的排队工作，如果当前线程获取同步状态失败（锁）时，AQS 则会将当前线程以及等待状态等信息构造成一个节点（Node）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，则会把节点中的线程唤醒，使其再次尝试获取同步状态。

同步器是实现锁（也可以是任意同步组件）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。

共享式获取和独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。以读写锁 ReentrantReadWriteLock 为例，它的读取锁 ReadLock 是共享式的，可以允许多个程序同时对文件进行读操作；但它的写入锁 WriteLock 是独占式的，同一时刻只能允许一个线程对文件进行写操作。

独占锁主要代码如下：

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
```

```
}
```

- tryAcquire：去尝试获取锁，获取成功则设置锁状态并返回 true，否则返回 false。**该方法自定义同步组件自己实现，该方法必须要保证线程安全的获取同步状态。**
- addWaiter：如果 tryAcquire 返回 FALSE（获取同步状态失败），则调用该方法将当前线程加入到 CLH 同步队列尾部。
- acquireQueued：当前线程会根据公平性原则来进行阻塞等待（自旋），直到获取锁为止；并且返回当前线程在等待过程中有没有中断过。
- selfInterrupt：产生一个中断。

共享锁主要代码如下：

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

方法尝试获取同步状态，如果获取失败则调用 doAcquireShared(int arg)自旋方式获取同步状态，共享式获取同步状态的标志是返回 >= 0 的值表示获取成功。同样，**tryAcquireShared()方法也需要自定义同步组件自己实现。在 ReentrantReadWriteLock.ReadLock 中重写的 tryAcquireShared()方法中，通过获取锁的共享计数是否超过限制（MAX_COUNT，65535）来进行判断。**

锁的公平性，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，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获得锁，也就是说锁获取是顺序的。当然，公平锁机制往往没有非公平的效率高，但也能够减少"饥饿"发生的概率。

公平锁与非公平锁的区别在于获取锁的时候是否按照 FIFO 的顺序来。释放锁不存在公平性和非公平性。

比较公平锁和非公平锁获取同步状态的 tryAcquire()方法（以独占式排他锁 ReentrantLock 为例，所以是 tryAcquire()，如果是共享锁，则是 tryAcquireShared()），两者区别在于公平锁在获取同步状态时多了一个限制条件 hasQueuedPredecessors()，定义如下。

```
public final boolean hasQueuedPredecessors() {
       Node t = tail;  //尾节点
       Node h = head;  //头节点
       Node s;
       //头节点 != 尾节点
       //同步队列第一个节点不为 null
       //当前线程是同步队列第一个节点
       return h != t &&
               ((s = h.next) == null || s.thread != Thread.currentThread());
   }
```

该方法主要判断当前线程是否位于 CLH 同步队列中的第一个，如果是则返回 true，否则返回 false。这点保证了公平性。 **具体代码强烈建议**这块去看一下 ReentrantReadWriteLock、ReentrantLock 类的源码。

17、简述 ConcurrentLinkedQueue 和 LinkedBlockingQueue 的用处和不同之处。

ConcurrentLinkedQueue 类型基于 lock-free，采用 CAS 操作 Node 节点（Node 节点里的元素也用 volatile 修饰，类似于 CLH），来保证元素的一致性。（需要看一波这个实现代码了^_~）LinkedBlockingQueue 是用一个独占锁来保持线程安全，然后用 Condition 来做阻塞操作。实现了先进先出等特性，是作为生产者消费者的首选。

18、导致线程死锁的原因，怎么解除线程死锁。

死锁是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的必要条件：

- 互斥条件，所谓互斥就是线程在某一个时间内独占资源。
- 请求与保持条件，一个进程因请求资源而阻塞时，对已获得的资源保持不变。
- 不剥夺条件，线程已获得资源，在未使用之前，不能强行剥夺。
- 循环等待条件，若干线程之间形成一种头尾相接的循环等待资源关系。

解除线程死锁的方法：

- 加锁顺序，当多个线程需要相同的一些锁，必须确保所有线程都是按照相同的顺序获得锁。
- 加锁时限，在尝试获取锁的时候加一个超时时间，若超过这个时限该线程则放弃对该锁的请求。
- 死锁检测：每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph 等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。
- 当出现死锁时，释放所有锁，回退，并且等待一段随机的时间后重试。或者更好的做法是给这些线程设置优先级，让一个（或多个）线程回退，其他线程继续保持它们需要的锁。

19、用过读写锁吗，原理是什么，一般在什么场景下用。

与排他锁 ReentrantLock 不同，读写锁 ReentrantReadWriteLock 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。读写锁维护了一个对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大的提升。

除了保证写操作对读操作的可见性以及并发性的提升外，读写锁能够简化读写交互场景的编程方法。假设在程序中定义一个共享的用作缓存数据结构，它大部分时间提供读服务（例如查询和搜索），而写操作占用的时间很少，但写操作完成之后的更新需要对后续的读服务可见。

写锁就是一个支持可重入的排他锁。写锁的状态获取最终会调用 tryAcquire(int arg)方法，注意，在判断重入时加入了一项条件：读锁是否存在。因为要确保写锁的操作对读锁是可见的，因此只有等读锁完全释放后，写锁才能够被当前线程所获取。一旦写锁获取了，所有其他读、写线程均会被阻塞。

读锁为一个可重入的共享锁，它能够被多个线程同时持有，在没有其他写线程访问时，读锁总是或获取成功。读锁是通过调用 tryAcqurireShared(int arg)方法尝试获取读同步状态，该方法主要用于获取共享式同步状态，获取成功返回 >= 0 的返回结果，否则返回 < 0 的返回结果。读锁获取方式比写锁麻烦一些，需要注意的是：

- 因为存在锁降级情况，如果存在写锁且锁的持有者不是当前线程则直接返回失败，否则继续。
- 依据公平性原则，判断读锁是否需要阻塞，读锁持有线程数小于最大值（65535），且设置锁状态成功。

参考链接：【死磕 Java 并发】-----J.U.C 之读写锁：ReentrantReadWriteLock

20、开启多个线程，如果保证顺序执行，有哪几种实现方式，或者如何保证多个线程都执行完毕后再拿到结果。

保证多个线程顺序执行的方法有几种：

- 采用信号量 Sephmore，下一个线程执行的条件是上一个线程执行完毕后释放信号量。
- 使用 ReentrantLock，每次只有一个线程可以获取锁来执行。
- 采用 Countdownlatch，可以让下一个线程执行的条件是前一个线程为 0。
- 使用单线程池，这样就能根据传入的顺序执行线程。等等，还比如阻塞队列等。

保证多个线程都执行完毕后再拿到结果可以采用 Countdowlatch 和 Cyclicbarrier 来实现。

21、当一个线程进入某一个对象的一个 synchronized 的实例方法后，其他线程是否可以进入此对象的其他方法。

如果该实例的其他方法没有 synchronized 修饰的话，其他线程是可以进入的。

另外，需要注意的是，synchronized 是实例锁（锁在某一个实例对象上，如果该类是单例，那么该锁也具有全局锁的概念），static synchronized 是类锁（该锁针对的是类，无论实例多少个对象，那么线程都是共享该锁），并且对象锁与类锁互不干扰，与对象无关。

- synchronized 是对类的**当前实例**（当前对象）进行加锁，防止其他线程同时访问该类的该实例的所有 synchronized 块（注：是所有），注意这里是"类的当前实例"， 类的两个不同实例就没有这种约束了。
- static synchronized 恰好就是要控制类的所有实例的并发访问，static synchronized 是限制**多线程中该类的所有实例**同时访问 jvm 中该类所对应的代码块。

参考链接：Synchronized（对象锁）和 Static Synchronized（类锁）的区别

22、可以创建 volatile 数组吗？

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

同理，对于 Java POJO 类，使用 volatile 修饰，只能保证这个引用的可见性，不能保证其内部的属性。

23、用三个线程按顺序循环打印 abc 三个字母，比如 abcabcabc？

采用等待/通知机制来实现，主要分为两种。

- 使用 Lock+Condition 来实现。参考用三个线程按顺序循环打印 abc 三个字母，比如 abcabcabc。
- 使用 synchronized+await/notifyAll 来实现。参考 Java 用三个线程按顺序循环打印 abc 三个字母,比如 abcabcabc。

24、一个线程池设计的最大线程数应该考量哪些因素。

要想合理地配置线程池的大小，首先要分析任务的特性，可以从以下几个角度分析：

- 任务的性质：计算密集型任务、IO 密集型任务。
- 任务的优先级：高、中、低。
- 任务的执行时间：长、中、短。
- 任务的依赖性：是否依赖其他系统资源，如数据库操作。

性质不同的任务可以交给不同规模的线程池执行。

在有 N 个 CPU 的系统上，计算密集型任务应配置尽可能少的线程，可以将线程池大小设置为 N+1；

IO 密集型任务应配置尽可能多的线程，因为 IO 操作不占用 CPU，不要让 CPU 闲下来，应加大线程数量，如线程池大小设置为 2N+1；

而对于混合型的任务，如果可以拆分，拆分成 IO 密集型和计算密集型分别处理，前提是两者运行的时间是差不多的，如果处理时间相差很大，则没必要拆分了。

若任务对其他系统资源有依赖，如某个任务依赖数据库的连接返回的结果，这时候等待的时间越长，则 CPU 空闲的时间越长，那么线程数量应设置得越大，才能更好的利用 CPU。

当然具体合理线程池值大小，需要结合系统实际情况，在大量的尝试下比较才能得出，以上只是前人总结的规律。

在这篇如何合理地估算线程池大小？有一个公认的估算合理值的公式，如下 $$ 最佳线程数目 = （线程等待时间/线程 CPU 时间 + 1）* CPU 数目 $$ 可以得出一个结论：线程等待时间所占比例越高，需要越多线程。线程 CPU 时间所占比例越高，需要越少线程。

以上公式与之前的 CPU 和 IO 密集型任务设置线程数基本吻合。

至于为什么要+1，我理解为要留一个给主线程使用的，避免后台任务将 CPU 资源完全耗尽。

另外，是否使用线程池就一定比单线程高效，答案是否定的，比如 Redis 就是单线程，但它却非常高效。从线程的角度，部分原因在于：

- 多线程会带来线程上下文切换开销，单线程就没有这种开销。
- 单线程避免了锁的设计。

当然，“Redis”更快的本质原因还在于：Redis 基于内存操作，这种情况下单线程可以很高效地利用 CPU。而多线程使用场景一般是：存在相当比例的 IO 和网络操作。

25、在 Java 中守护线程和本地线程区别。

Java 中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法 Thread.setDaemon(boolean)；true 则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在 Thread.start()之前调用，否则会抛出异常。

两者唯一的区别是判断虚拟机（JVM）何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经撤离，Daemon 没有可服务的线程，JVM

撤离。也可以理解为守护线程是 JVM 自动创建的线程（但不一定），用户线程是程序创建的线程。比如 JVM 的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。

另外，Thread Dump 打印出来的线程信息，含有 daemon 字样的线程即为守护线程，可能会有：服务守护线程、编译守护线程、windows 下的监听 Ctrl+break 的守护线程、GC 守护线程等。

26、Java 中用到的线程调度算法是什么。

计算机通常只有一个 CPU,在任意时刻只能执行一条机器指令,每个线程只有获得 CPU 的使用权才能执行指令.所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得 CPU 的使用权,分别执行各自的任务.在运行池中,会有多个处于就绪状态的线程在等待 CPU,JAVA 虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配 CPU 的使用权。

有两种调度模型：分时调度模型和抢占式调度模型。

- 分时调度模型是指让所有的线程轮流获得 cpu 的使用权,并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。
- 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，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

27、线程的状态。

一般来说，都把线程分为多个状态：NEW（新建状态）、RUNNABLE(运行状态)、BLOCKED(锁池)、TIMED_WAITING(定时等待)、WAITING(等待)、TERMINATED(终止、结束)。

具体的关系可以总结为一张图：

注意，线程状态 waiting 和 blocked 的区别在于：

- 线程可以通过 notify,join,LockSupport.park 方式进入 wating 状态，进入 wating 状态的线程等待唤醒(notify 或 notifyAll)才有机会获取 cpu 的时间片段来继续执行。
- 线程的 blocked 状态往往是无法进入同步方法/代码块来完成的。这是因为无法获取到与同步方法/代码块相关联的锁。

与 wating 状态相关联的是**等待队列**，与 blocked 状态相关的是**同步队列**，一个线程由等待队列迁移到同步队列时，线程状态将会由 wating 转化为 blocked。可以这样说，blocked 状态是处于 wating 状态的线程重新焕发生命力的必由之路。例如，notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中去，被移动的线程状态是从 WAITING 变成 BLOCKED。

这块具体可以看一下《Java 并发编程的艺术》中 4.3.2 小节 等待/通知机制，P101。

28、AQS 组件总结。

Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

CountDownLatch （倒计时器）： CountDownLatch 是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

CyclicBarrier(循环栅栏)： CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其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 我已经到达了屏障，然后当前线程被阻塞。

29、进程和线程的区别。

进程是操作系统进行资源分配的基本单位。进程控制块 (Process Control Block, PCB) 描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对 PCB 的操作。一个进程中可以有多个线程，线程是独立调度的基本单位。同一个进程中的多个线程之间可以并发执行，它们共享进程资源。

进程是系统中独立存在的实体，它可以拥有自己独立的资源，每个进程都拥有自己私有的地址空间，在没有进程本身运行的情况下是不能访问其中的内容的。

线程和进程的区别在于，子进程和父进程有不同的代码和数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文。多线程主要是为了节约 CPU 时间，发挥利用，根据具体情况而定。线程的运行中需要使用计算机的内存资源和 CPU。

线程和进程的区别归纳：

- **地址空间与其他资源**：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其他进程不可见。
- **通信**：进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要线程同步和互斥手段的辅助，以保证数据的一致性。
- **调度和切换**：线程上下文切换比进程上下文切换要快得多。并且线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
- **系统开销：**由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，因此操作系统所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置。而线程切换时只需保存和设置少量寄存器内容，开销很小。

30、进程间通信的方式。

进程间通信基本上有 5 种通讯方式。

- 无名管道通信：速度慢，容量有限，只有父子进程能通信。
- FIFO：任何进程间都能通信，但是速度慢。
- 消息队列：容量受到系统限制，可以实现消息的随即查询，消息不一定以先进先出的次序读取，也可以按照消息的类型读取。且要注意第一次读的时候，要考虑上一次没有读完数据的问题。
- 信号量：不能传递复杂消息，只能用来同步。
- 共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全。

31、LockSupport 的优势。

LockSuppor 也是用于线程挂起和唤醒的，相比 Object 的 wait/notify 有两大优势：

- LockSupport 不需要在同步代码块中，所以线程间不需要维护一个共享的同步对象了，实现了线程间的解耦。
- unpark 函数可以先与 park 调用，所以不需要担心线程间的执行的先后顺序。

32、为什么 wait(),notify(),notifyAll()必须在同步（Synchronized）方法/代码块中调用？

调用 wait()方法就是释放锁，释放锁的前提是必须要先获取锁，先获取锁才能释放锁。

notify()，notifyAll()是将锁交给含有 wait()方法的线程，让其继续执行下去，所以自身必须先有锁才行。

33、ThreadLocal 是否可以用 static 修饰。

在开发过程中，ThreadLocal 一般会采用 static 修饰，这样做既有好处也有坏处。好处是它一定程度上可以避免错误，至少可以避免 重复创建 TSO（Thead Specific Object，即 ThreadLocal 所关联的对象）所导致的浪费。坏处是这样做可能正好形成内存泄漏所需的条件。

我们知道，一个 ThreadLocal 实例对应当前线程中的一个 TSO 实例。因此，如果把 ThreadLocal 声明为某个类的实例变量（而不是静态变量），那么每创建一个实例都会导致一个新的 TSO 实例被创建。显然，这些被创建的 TSO 实例是同一个类的实例。于是，同一个线程可能会访问到同一个 TSO（指类）的不同实例，这既便不会导致错误，也会导致浪费（重复创建等同的对象）！因此，我们一般将 ThreadLocal 使用 static 修饰即可。

现在来讨论缺点。

由于 ThreadLocal 是某个类的一个静态变量，因此，只要相应的类没有被垃圾回收掉，那么这个类就会持有相应的 ThreadLocal 实例的应用。另外，ThreadLocal 的内部实现包括一个类似 HashMap 的对象，这里称之为 ThreadLocalMap。ThreadLocalMap 的 key 会持有对 ThreadLocal 实例的弱引用（Weak Reference），value 会引用 TSO 实例。当服务器的工作者线程不会被垃圾回收时，ThreadLocal 实例此时也不会被垃圾回收，这就产生了内存泄漏。