JAVA并发基础与关键字

- **进程，线程和协程的区别**
  - 进程
    - 是程序的一次执行实例，拥有独立的地址空间和系统资源
    - 资源隔离 — 每个进程拥有独立的内存地址空间，进程间的通信（IPC）需要特殊的机制，如管道、消息队列、共享内存等
  - 线程 — 线程是进程内的执行单元，是CPU调度的基本单位。一个进程可以包含多个线程，它们共享进程的资源，如内存和文件句柄
  - 协程（虚拟线程）
    - 协程是一种用户态的轻量级线程，它的调度完全由应用程序控制，而非操作系统。协程提供了非抢占式的多任务处理，通过任务协作而非竞争来实现并发
    - 协程间共享同一线程的资源，避免了多线程的锁机制，可以更高效地进行IO操作
- **线程是如何被调度的?** — 对于单CPU的计算机来说，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。
- **为啥要使用多线程** — CPU、内存、I/O 设备的速度是有极大差异的，为了合理利用 CPU 的高性能，平衡这三者的速度差异
- **多线程不安全的示例** — 如果多个线程对同一个共享数据进行访问而不采取同步操作的话，那么操作的结果是不一致的
- **什么是多线程中的上下文切换**
  - 上下文切换是指 CPU 从一个线程转到另一个线程时，需要保存当前线程的上下文状态，恢复另一个线程的上下文状态，以便于下一次恢复执行该线程时能够正确地运行
  - 过多的上下文切换会降低系统的运行效率，因此需要尽可能减少上下文切换的次数
- **能不能谈谈你对线程安全的理解** — 线程安全是指某个函数在并发环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成
- **如何选择阻塞队列?**
  - 功能需求：考虑是否需要阻塞队列提供特定的功能，如优先级排序、延迟执行等。如果有这样的需求，就应该选择具有相应功能的阻塞队列，例如PriorityBlockingQueue支持按优先级排序，DelayQueue支持延迟执行。
  - 容量需求：
    - 如果需要固定容量的队列，可以选择ArrayBlockingQueue。
    - 如果需要无限容量的队列，可以选择LinkedBlockingQueue。
    - 如果不需要存储任何元素，只是用作传递，可以选择SynchronousQueue。
  - 动态扩容：如果业务场景中存在高峰期和低谷期，且无法在初始时准确估计队列大小，那么可能需要一个能够动态扩容的队列。在这种情况下，PriorityBlockingQueue是一个不错的选择，因为它可以在需要时自动扩容。
  - 内存结构：从内存结构的角度考虑，ArrayBlockingQueue基于数组实现，空间利用率较高；而LinkedBlockingQueue等基于链表实现的队列则可能在每个节点上有一些额外的内存开销。
  - 性能考虑：不同的阻塞队列在实现上有所不同，这可能会影响它们的性能。在选择时，需要根据具体的应用场景和性能测试结果来做出决策。
- **线程基础**
  - **线程状态/生命周期（看图）**
    - 新建(New)
      - 当线程实例被创建，但在调用Thread.start()方法之前，线程处于这个状态
      - 在这个状态下，线程已经被分配了必要的内存和其他资源，但尚未开始执行
    - 可运行(Runnable)
      - 当调用线程的start()方法后，线程进入可运行状态 — 在这个状态下，线程已经具备了运行的所有条件，但线程调度器尚未选中它来分配CPU时间片
      - 可运行状态包括“就绪”和“运行”两个子状态
        - 就绪（Ready）：线程已经准备好运行，等待CPU调度
        - 运行（Running）：线程正在执行
    - 阻塞(Blocking)
      - 当线程试图访问一个被其他线程锁定的同步代码块时，它会移入阻塞状态
      - 线程因为等待监视器锁（在进入同步块或方法时）而被阻塞,其他线程释放后就退出阻塞状态
      - 阻塞和等待的区别在于
        - 阻塞是被动的，它是在等待获取一个排它锁。
        - 等待是主动的，通过调用 Thread.sleep() 和 Object.wait() 等方法进入。
    - 等待(Waiting)
      - 在等待状态的线程需要其他线程来执行特定操作（如通知或中断）才能回到可运行状态
      - 进入方法
        - Object.wait()
          - 线程调用某个对象的wait()方法后，会释放该对象的锁（如果有的话）并进入该对象的等待池
          - 这个方法必须在同步代码块或同步方法中调用。调用线程必须持有那个对象的锁
          - wait()方法可以使线程等待直到另一个线程调用同一个对象的notify()或notifyAll()方法
        - Thread.join()
          - 当一个线程A执行另一个线程B的join()方法时，线程A会进入等待状态直到线程B完成执行
          - join()方法允许一个线程等待另一个线程完成其生命周期
        - LockSupport.park()
          - 这是一个更灵活的阻塞线程的机制，不需要在同步块内使用
          - 当线程调用LockSupport.park()时，它会进入等待状态直到获得许可
      - 退出方法
        - Object.notify() / Object.notifyAll()
          - 当某个线程调用对象的notify()方法时，它会随机唤醒等待在该对象的等待池中的一个线程
          - 如果调用notifyAll()，则会唤醒所有在该对象等待池中的线程
          - 被唤醒的线程（们）会尝试重新获取对象的锁，一旦获取成功，就可以继续执行
        - 指定等待时间
          - 如果在调用wait()时指定了一个时间参数，线程会在指定时间后自动醒来（如果在此之前没有被notify()或notifyAll()唤醒）
          - 类似地，join(long millis)和LockSupport.parkNanos()或LockSupport.parkUntil()也支持带有超时的等待，允许线程在超时后自动退出等待状态
    - 限时等待(Timed Waiting)
      - 无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒
      - 线程在调用带有超时参数的sleep(), wait(), join(), 或 LockSupport.parkNanos() / parkUntil()方法后进入此状态
      - 如果在指定时间内没有收到通知，线程会自动返回到可运行状态
    - 死亡(Terminated) — 可以是线程结束任务之后自己结束，或者产生了异常而结束。
    - 多线程的过程 — 多线程的执行过程包括线程的创建、就绪、运行、阻塞和死亡五个状态。线程通过继承Thread类或实现Runnable接口等方式创建后，调用start()方法进入就绪状态。当线程获得CPU资源时开始运行，执行run()方法。线程可能因某些原因（如执行sleep()方法、等待同步锁等）进入阻塞状态。当run()方法执行完毕或线程被异常终止时，线程进入死亡状态。
  - **线程创建方式四种**
    - 实现Runnable接口
      - 需要实现 run() 方法
      - 通过 Thread 调用 start() 方法来启动线程
    - 实现 Callable 接口 — 与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装
    - 继承 Thread 类 — 也是需要实现 run() 方法，因为 Thread 类也实现了 Runable 接口
    - 通过线程池创建线程
      - 什么是线程池
        - 线程池，说的就是提前创建好一批线程，然后保存在线程池中，当有任务需要执行的时候，从线程池中选一个线程来执行任务
        - 在机器资源有限的情况下，使用池化技术可以大大的提高资源的利用率，提升性能等
      - 核心参数
        - acc : 获取调用上下文
        - corePoolSize: 核心线程数量，可以类比正式员工数量，常驻线程数量。
        - maximumPoolSize: 最大的线程数量，公司最多雇佣员工数量。常驻+临时线程数量。
        - workQueue：多余任务等待队列，再多的人都处理不过来了，需要等着，在这个地方等。
        - keepAliveTime：非核心线程空闲时间，就是外包人员等了多久，如果还没有活干，解雇了。
        - threadFactory: 创建线程的工厂，在这个地方可以统一处理创建的线程的属性。每个公司对员工的要求不一样，恩，在这里设置员工的属性。
        - handler：线程池拒绝策略，什么意思呢？就是当任务实在是太多，人也不够，需求池也排满了，还有任务咋办？默认是不处理，抛出异常告诉任务提交者，我这忙不过来了
        - 如何实现参数的动态修改?
          - 自定义线程池：扩展或实现自己的线程池，添加修改参数的功能。
          - 使用定时任务：定期检查并调整线程池参数，但要避免频繁调整以免影响性能。
          - 外部配置：利用配置中心动态更新线程池配置，应用程序读取新配置后相应调整。
          - JMX接口：使用Java的管理和监控接口在运行时修改线程池参数
      - 什么时候会达到最大线程数 — 当工作队列已满，且活动线程数量未达到最大线程数时，线程池会根据最大线程数的设置决定是否创建新的线程来处理任务。如果活动线程数量小于最大线程数，则线程池会创建新的线程来执行任务，直到活动线程数量达到最大线程数。
      - 核心线程数和最大线程数如何来确定
        - 根据任务特性确定：核心线程数可设为满足应用程序基本需求的线程数，最大线程数则根据应用程序的最大负载和处理能力来确定；
        - 根据系统资源限制确定：核心线程数可参考系统的CPU核心数，最大线程数则设为一个合理的上限值，以避免过多线程竞争资源；
        - 根据任务类型确定：对于计算密集型任务，可能需要更多的线程来并行执行；对于IO密集型任务，则可以适当减少线程数。
      - 如何创建线程池
        - executors创建线程池的方法
          - 创建出来的线程池都实现了ExecutorService接口
          - newFixedThreadPool(int Threads)：创建固定数目线程的线程池
          - newCachedThreadPool() — 创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果没有可用的线程，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程
          - newSingleThreadExecutor() — 创建一个单线程化的Executor
          - newScheduledThreadPool(int corePoolSize) — 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类
        - 使用ThreadPoolExecutor直接创建
          - corePoolSize：核心线程数，即使线程是空闲的，线程池也会尽量保持这么多的线程数
          - maximumPoolSize：线程池允许的最大线程数
          - keepAliveTime：当线程数大于核心线程数时，这是超出核心线程数的线程在终止前等待新任务的最长时间
          - unit：keepAliveTime参数的时间单位。
          - workQueue：用于在任务执行前存储任务的队列。常用的有LinkedBlockingQueue、SynchronousQueue等
          - threadFactory：用于设置如何创建新线程。默认情况下使用的是Executors.defaultThreadFactory()。
          - handler：当执行被阻塞时使用的拒绝策略，比如ThreadPoolExecutor.AbortPolicy（抛出RejectedExecutionException）、ThreadPoolExecutor.CallerRunsPolicy（在调用者线程中运行任务）等
          - 在创建的同时，给`BlockQueue`指定容量就可以了
      - 线程池的拒绝策略有哪些
        - AbortPolicy - 这是默认的拒绝策略 — 当线程池无法接受新任务时，会抛出RejectedExecutionException异常。这意味着新任务会被立即拒绝，不会加入到任务队列中，也不会执行
        - DiscardPolicy - 这个策略在任务队列已满时，会丢弃新的任务而且不会抛出异常
        - DiscardOldestPolicy - 这个策略也会丢弃任务，但它会先尝试将任务队列中最早的任务删除，然后再尝试提交新任务
        - CallerRunsPolicy - 这个策略将任务回退给调用线程，而不会抛出异常

```java
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    int corePoolSize,         // 核心线程池大小
    int maximumPoolSize,      // 最大线程池大小
    long keepAliveTime,       // 线程最大空闲时间
    TimeUnit unit,            // 时间单位
    BlockingQueue<Runnable> workQueue, // 线程等待队列
    ThreadFactory threadFactory,       // 线程创建工厂
    RejectedExecutionHandler handler   // 拒绝策略
);
```

    - **Runnable和Callable区别**
      - Runnable接口和Callable接口都可以用来创建新线程，实现Runnable的时候，需要实现run方法；实现Callable接口的话，需要实现call方法。
      - Runnable的run方法无返回值，Callable的call方法有返回值，类型为Object
      - Callable中可以够抛出checked exception,而Runnable不可以。
      - Callable和Runnable都可以应用于executors。而Thread类只支持Runnable。
    - 实现接口 VS 继承 Thread
      - 实现接口好一些
      - Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口；
      - 类可能只要求可执行就行，继承整个 Thread 类开销过大。
  - 基础线程机制
    - Executor
      - Executor 管理多个异步任务的执行，而无需显式地管理线程的生命周期
      - 这里的异步是指多个任务的执行互不干扰，不需要进行同步操作
      - 主要有三种 Executor
        - CachedThreadPool — 一个任务创建一个线程
        - FixedThreadPool — 所有任务只能使用固定大小的线程
        - SingleThreadExecutor — 相当于大小为 1 的 FixedThreadPool
      - Executors的创建线程池的方法，创建出来的线程池都实现了ExecutorService接口
      - 用它创建线程池有缺点
        - 存在OOM
          - LinkedBlockingQueue是一个用链表实现的有界阻塞队列，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，最大长度为Integer.MAX_VALUE
          - newFixedThreadPool中创建LinkedBlockingQueue时，并未指定容量
          - 对于一个无边界队列来说，是可以不断的向队列中加入任务的，这种情况下就有可能因为任务过多而导致内存溢出问题
        - 如何正确创建线程池
          - 避免使用Executors创建线程池，主要是避免使用其中的默认实现，那么我们可以自己直接调用`ThreadPoolExecutor`的构造函数来自己创建线程池。在创建的同时，给`BlockQueue`指定容量就可以了
          - 还可以自定义线程名称，更加方便的出错的时候溯源
    - Daemon守护线程
      - **守护线程是程序运行时在后台提供服务的线程，不属于程序中不可或缺的部分**
      - **当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程**
      - main() 属于非守护线程 — 使用 setDaemon() 方法将一个线程设置为守护线程
      - **和其他线程的区别：Java虚拟机在所有<用户线程>都结束后就会退出，而不会等<守护线程>执行完。**
    - sleep()
      - 静态方法，Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒
      - sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理。
      - 线程中抛出的其它异常也同样需要在本地进行处理。
      - 自动被唤醒
      - 持有锁时执行不会释放锁资源
    - yield()
      - 对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行
      - 该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。
  - 线程中断 - 一个线程执行完毕之后会自动结束，如果在运行过程中发生异常也会提前结束
    - InterruptedException
      - 如果该线程处于**阻塞**、**限期等待**或者**无限期**等待状态
      - 会提前结束该线程。但是不能中断 I/O 阻塞和 synchronized 锁阻塞
      - 通过调用一个线程的interrupt()来中断线程
    - interrupted()
      - 如果一个线程的 run() 方法执行一个无限循环，并且没有执行 sleep() 等会抛出 InterruptedException 的操作，那么调用线程的 interrupt() 方法就无法使线程提前结束。
      - 但是调用 interrupt() 方法会设置线程的中断标记，此时调用 interrupted() 方法会返回 true。因此可以在循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。
    - Executor 的中断操作
      - 调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭
      - 但是如果调用的是 shutdownNow() 方法，则相当于调用每个线程的 interrupt() 方法。
      - 如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 cancel(true) 方法就可以中断线程
  - 线程同步的方式有哪些
    - synchronized — Java中最基本的线程同步机制，可以修饰代码块或方法，保证同一时间只有一个线程访问该代码块或方法，其他线程需要等待锁的释放。
    - ReentrantLock — 与synchronized关键字类似，也可以保证同一时间只有一个线程访问共享资源，但是更灵活，支持公平锁、可中断锁、多个条件变量等功能。
    - Semaphore — 允许多个线程同时访问共享资源，但是限制访问的线程数量。可以用于控制并发访问的线程数量，避免系统资源被过度占用。
    - CountDownLatch — 允许一个或多个线程等待其他线程执行完毕之后再执行，可以用于线程之间的协调和通信。
    - CyclicBarrier类 — 允许多个线程在一个栅栏处等待，直到所有线程都到达栅栏位置之后，才会继续执行。
    - **CountDownLatch、CyclicBarrier、Semaphore区别**
      - CountDownLatch是一个计数器
        - 它允许一个或多个线程等待其他线程完成操作。 它通常用来实现一个线程等待其他多个线程完成操作之后再继续执行的操作。
        - CountDownLatch适用于一个线程等待多个线程完成操作的情况
      - CyclicBarrier是一个同步屏障
        - 它允许多个线程相互等待，直到到达某个公共屏障点，才能继续执行。 它通常用来实现多个线程在同一个屏障处等待，然后再一起继续执行的操作。
        - CyclicBarrier适用于多个线程在同一个屏障处等待
      - Semaphore是一个计数信号量
        - 它允许多个线程同时访问共享资源，并通过计数器来控制访问数量。 它通常用来实现一个线程需要等待获取一个许可证才能访问共享资源，或者需要释放一个许可证才能完成操作的操作。
        - Semaphore适用于一个线程需要等待获取许可证才能访问共享
    - 如何理解AQS?
      - AQS 是很多同步器的基础框架，比如 ReentrantLock、CountDownLatch 、Semaphore、volatile、CAS、FIFO Queue都是基于 AQS 实现的 — AQS只是个接口，具体资源的获取、释放都由自定义的同步器去实现
      - AQS有两条队列
        - 同步队列 — 用于实现锁的获取和释放
        - 条件队列 — 用于在特定条件下管理线程的等待和唤醒
      - AQS是如何实现线程的等待和唤醒的? — AQS中线程等待和唤醒主要依赖park和unpark实现的。
        - 当一个线程尝试获取锁或者同步器时，如果获取失败，AQS会将该线程封装成一个Node并添加到等待队列中，然后通过LockSupport.park()将该线程阻塞
        - 当一个线程释放锁或者同步器时，AQS会通过LockSupport.unpark()方法将等待队列中的第一个线程唤醒，并让其重新尝试获取锁或者同步器
  - 线程间的协作 - 当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调
    - join() — 在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束
    - **唤醒/启用线程方法（需要锁标记）：wait() notify() notifyAll()** — 调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。
    - **wait() 和 sleep() 的区别**
      - wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；
      - wait() 会释放锁，sleep() 不会
    - await() signal() signalAll()一般和Lock接口一起使用，也是唤醒线程的
      - 可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程
      - 相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活
  - 线程互斥同步 - Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问
    - Synchronized Lock — 它提供了一种互斥机制，也就是在同一时刻只允许一个线程访问共享资源 — 可以通过在方法声明中使用synchronized关键字来同步整个方法，使该方法在同一时刻只能被一个线程执行
    - ReentrantLock — ReentrantLock是java.util.concurrent.locks包中提供的一个显式锁 — 需要显式地创建一个ReentrantLock实例，然后在代码前调用lock()方法获取锁，在代码后调用unlock()方法释放锁
  - 线程的实现方式有哪些?
    - 使用内核线程实现
      - 由操作系统内核直接支持。
      - 内核负责线程切换、调度，并映射任务到处理器。
    - 使用用户线程实现
      - 在用户空间建立，通过运行时系统管理。
      - 优点：切换快，可跨操作系统运行。
      - 缺点：需用户程序自行处理线程操作，系统调用可能阻塞所有线程，且多处理器映射是挑战。
    - 使用用户线程加轻量级进程混合实现
      - 线程创建在用户空间，但调度由内核完成。
      - 多个用户线程复用多个内核线程。
  - 什么是ThreadLocal，如何实现的
    - ThreadLocal是java.lang下面的一个类，是用来解决java多线程程序中并发问题的一种途径；通过为每一个线程创建一份共享变量的副本来保证各个线程之间的变量的访问和修改互相不影响
      - 解决并发问题
      - 在线程中传递数据
    - 实现方法
      - ThreadLocal 的实现主要依赖于 ThreadLocalMap 这个内部类，它是 ThreadLocal 类的静态内部类，用于存储每个线程的局部变量
      - 每个 Thread 对象都持有一个 ThreadLocalMap 成员，该成员是用来存储本线程内所有 ThreadLocal 变量的值
    - 使用方法
      - 创建一个 ThreadLocal 实例。
      - 通过 set将此线程局部变量的当前线程副本中的值设置为指定值。许多应用程序不需要这项功能，它们只依赖于 initialValue() 方法来设置线程局部变量的值
      - 通过 get返回此线程局部变量的当前线程副本中的值。如果这是线程第一次调用该方法，则创建并初始化此副本get() 方法获取当前线程的变量值。
      - 使用完 ThreadLocal 后，可以通过 remove() 方法清除当前线程绑定的值，以避免潜在的内存泄漏。
    - ThreadLocal为什么会导致内存泄漏？如何解决的
      - 栈上的ThreadLocal Ref引用不在使用了，即方法结束后这个对象引用就不再用了，ThreadLocal对象因为还有一条引用链在，所以就会导致他无法被回收
      - Thread对象如果一直在被使用，会导致ThreadLocalMap无法被回收
      - 使用了弱引用
      - 手动清理ThreadLocal
    - Thread.sleep(0)的作用
      - Thread的sleep会让线程暂时释放CPU资源，然后进入到TIMED_WAITING状态，等到指定时间之后会再尝试获取CPU时间片。
      - 其实就是让当前线程释放一下CPU时间片，然后重新开始争抢
    - ThreadLocalMap结构
      - 如何解决哈希冲突 — ThreadLocalMap 使用开放定址法（线性探测）来解决哈希冲突。当发生哈希冲突时，它会在数组中寻找下一个可用的槽位
      - 扩容机制 — 当 ThreadLocalMap 中的元素数量达到数组长度的三分之二时，会触发扩容。扩容时，数组的长度会加倍，并且所有的元素都会重新散列到新的数组中
    - 线程本地变量介绍 — 线程本地变量是线程私有的，它们的值对于每个线程都是独立的。这意味着，如果你在一个线程中修改了 ThreadLocal 变量的值，它不会影响其他线程中该变量的值
    - 父子线程如何实现共享数据? — 虽然 ThreadLocal 主要是用于存储线程局部变量，但有时我们可能需要在父子线程之间共享数据，ThreadLocal 本身不支持这种功能，但是可以使用 InheritableThreadLocal，它是 ThreadLocal 的一个扩展，可以实现在父线程中设置的值被子线程继承
    - ThreadLocal和同步有什么不同 — ThreadLocal和同步（synchronization）是解决多线程并发问题的两种不同机制。
      - ThreadLocal为每个线程提供了一个独立的变量副本，以解决线程间的数据隔离问题，它侧重于数据的隔离性。
      - 而同步机制（如synchronized关键字或Lock接口）则是用来控制多个线程对共享资源的访问，以避免数据不一致和其他并发问题，它侧重于数据的共享和一致性保护。

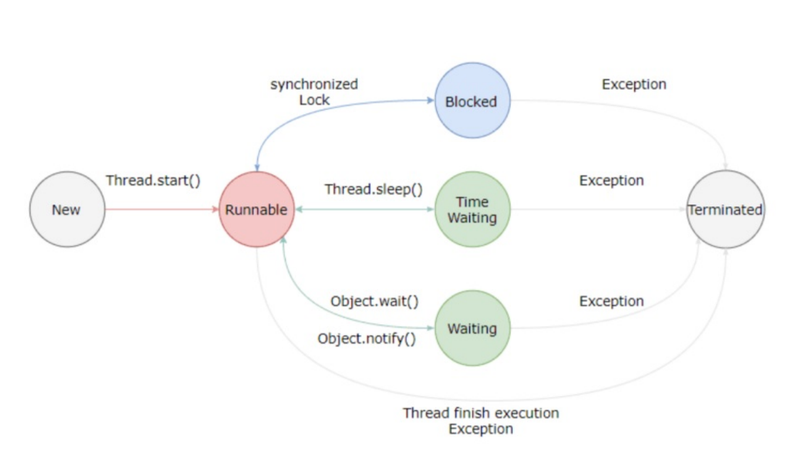
为什么要选择阻塞队列

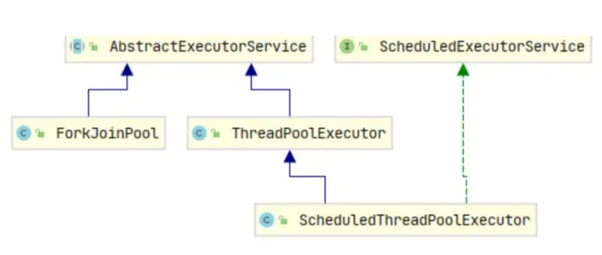
- 线程安全：阻塞队列是线程安全的，它可以在多线程环境中安全地进行入队（enqueue）和出队（dequeue）操作，而无需额外的同步措施。这大大简化了多线程编程的复杂性，减少了出错的可能性。
- 自动阻塞与唤醒：当队列为空时，从队列中获取元素的线程将会被阻塞，直到有其他线程向队列中插入了新的元素；同样，当队列已满时，尝试向队列中添加元素的线程将会被阻塞，直到队列中有空闲空间。这种机制可以有效地协调生产者和消费者的速度，避免了资源的浪费和过度的CPU占用。
- 容量限制：阻塞队列通常提供了容量的限制，这有助于防止内存溢出。当队列达到其容量上限时，生产者线程将被阻塞，直到消费者线程从队列中取出一些元素，为新的元素腾出空间。
- 简化并发编程模型：使用阻塞队列可以大大简化并发编程的模型。生产者只需将元素放入队列，而消费者只需从队列中取出元素，无需关心对方的状态。这种解耦使得生产者和消费者可以独立地工作和扩展。
- 提高系统响应能力：通过使用阻塞队列，系统可以在高负载情况下更加平稳地运行。当系统繁忙时，生产者线程可以被阻塞，从而避免无休止地消耗CPU资源；同样，当队列为空时，消费者线程也可以被阻塞，从而避免无效的轮询和CPU占用。
- 易于实现生产者-消费者模式：阻塞队列是生产者-消费者模式的自然实现方式。在这种模式中，生产者负责生成数据并放入队列，而消费者负责从队列中取出数据进行处理。阻塞队列的阻塞和唤醒机制使得这种模式更加高效和可靠。

进程的生命周期

- 新建（New）：进程被创建，如执行一个程序。
- 就绪（Ready）：进程已准备好运行并等待CPU分配时间片。
- 运行（Running）：进程正在CPU上执行其指令。
- 等待/阻塞（Waiting/Blocked）：进程因为某些事件（如I/O操作完成）而等待。
- 就绪挂起（Ready Suspended）：进程在就绪状态下被挂起（如被换出到磁盘）。
- 等待挂起（Waiting Suspended）：进程在等待/阻塞状态下被挂起。
- 终止（Terminated）：进程完成执行或被终止

线程状态转换





Java中线程池的继承关系

一些问题

- 并发出现问题的根源 — 并发三要素
  - 可见性: CPU缓存引起 — 一个线程对共享变量的修改，另外一个线程立刻看到
  - 原子性: 分时复用引起
    - 即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
    - 只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作
  - 有序性: 重排序引起 — 即程序执行的顺序按照代码的先后顺序执行
- 什么是并发，什么是并行?
  - 并发（Concurrent） — 在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行。操作系统是把CPU的时间划分成长短基本相同的时间区间,即”时间片”，通过操作系统的管理，把这些时间片依次轮流地分配给各个用户使用
  - 并行（Parallel） — 当系统有一个以上CPU时，当一个CPU执行一个进程时，另一个CPU可以执行另一个进程，两个进程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行
  - 并发是两个队伍交替使用一台设备。并行是两个队伍同时使用两台咖啡机
- 并发编程中的原子性和数据库ACID的原子性**不一样**
  - 在数据库中，事务的ACID中原子性指的是"要么都执行要么都回滚"。
  - 在并发编程中，原子性指的是"操作不可拆分、不被中断"。所以在并发编程中，我们要保证原子性指的就是一段代码需要不可拆分，不被中断。 — 比如 i++
  - 为什么Lua脚本可以保证原子性
    - Redis既是一个数据库，又是一个支持并发编程的系统，所以，他的原子性有两种。所以，我们需要明确清楚，在问"Lua脚本保证Redis原子性"的时候，指的到底是哪个原子性
    - Lua脚本可以保证原子性，因为Redis会将Lua脚本封装成一个单独的事务，而这个单独的事务会在Redis客户端运行时，由Redis服务器自行处理并完成整个事务
      - 如果在这个进程中有其他客户端请求的时候，Redis将会把它暂存起来，等到 Lua 脚本处理完毕后，才会再把被暂存的请求恢复
      - 这样就可以保证整个脚本是作为一个整体执行的，中间不会被其他命令插入。但是，如果命令执行过程中命令产生错误，事务是不会回滚的，将会影响后续命令的执行。
      - 也就是说，Redis保证以原子方式执行Lua脚本，但是不保证脚本中所有操作要么都执行或者都回滚。
      - 意味着Redis中Lua脚本的执行，可以保证并发编程中不可拆分、不被中断的这个原子性，但是没有保证数据库ACID中要么都执行要么都回滚的这个原子性
- 什么是Java内存模型（JMM）
  - Java程序是需要运行在Java虚拟机上面的，Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，**保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。**
  - 规定了所有的共享变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。
- JAVA是怎么解决并发问题的: JMM(Java内存模型)，它是规范了 JVM 如何提供按需禁用缓存和编译优化的方法
  - volatile、synchronized 和 final 三个关键字
    - Java提供了volatile关键字来保证可见性
      - 当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。
      - 而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。
    - 通过volatile关键字来保证一定的“有序性”
    - 另外可以通过synchronized和Lock来保证有序性， — 很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。
  - Happens-Before 规则
    - JMM是通过Happens-Before 规则来保证有序性的 — **适用于问题多线程下如何保证可见性和有序性**
    - 它的概念是：如果一个操作 A “happen-before” 另一个操作 B，那么 A 的结果对 B 是可见的
  - As-if-serial 语义
    - **是一种保证单线程程序的行为就像代码按顺序执行一样的概念**
    - 保证了单线程中，指令重排是有一定的限制的，而只要编译器和处理器都遵守了这个语义，那么就可以认为单线程程序是按照顺序执行的 — 所以由于synchronized修饰的代码，同一时间只能被同一线程访问。那么也就是单线程执行的。所以，可以保证其有序性
- 有三个线程T1,T2,T3如何保证顺序执行?
  - 想要让三个线程依次执行，并且严格按照T1,T2,T3的顺序的话，主要就是想办法让三个线程之间可以通信、或者可以排队。
  - 想让多个线程之间可以通信，可以通过join方法实现，还可以通过CountDownLatch、CyclicBarrier和Semaphore来实现通信。
  - 想要让线程之间排队的话，可以通过线程池或者CompletableFuture的方式来实现。
  - 不能依次执行start方法 — 数据结果不固定
- 三个线程分别顺序打印0-100

Synchronized

我们可以结合Sync和Object#notifyAll来完成，如下所示

Java 复制代码

```java
public class SortTest {

    private static final Object LOCK = new Object();
    private static volatile int count = 0;
    private static final int MAX = 100;

    public static void main(String[] args) {
        Thread thread = new Thread(new Seq(0));
        Thread thread1 = new Thread(new Seq(1));
        Thread thread2 = new Thread(new Seq(2));
        thread.start();
        thread1.start();
        thread2.start();
    }

    static class Seq implements Runnable {

        private final int index;

        public Seq(int index) {
            this.index = index;
        }

        @Override
        public void run() {
            while (count < MAX) {
                synchronized (LOCK) {
                    try {
                        while (count % 3 != index) {
                            LOCK.wait();
                        }
                        if(count <=MAX){
                            System.out.println("Thread-" + index + ": " + count);
                        }
                        count++;
                        LOCK.notifyAll();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

锁

- JDK 实现的 ReentrantLock
  - 一般优先使用Synchronized
  - ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件。 — 如何实现可重入
    - 可重入锁指的是同一个线程中可以多次获取同一把锁
    - 加锁的时候，看下当前持有锁的线程和当前请求的线程是否是同一个，一样就可重入
    - 一个可重入锁，重入了多少次，就得解锁多少次
  - ReentrantLock基于什么实现
    - 基于AQS：ReentrantLock的实现基础是AbstractQueuedSynchronizer（AQS），这是一个用于构建同步组件的基础框架。
    - 可重入性：ReentrantLock是一种可重入锁，意味着同一个线程可以多次获取同一把锁而不会造成死锁。这是通过内部维护一个计数器来实现的，每次获取锁时计数器增加，释放锁时计数器减少。
    - 公平锁与非公平锁：ReentrantLock提供了公平锁和非公平锁两种模式。公平锁会按照线程请求锁的顺序来分配锁，而非公平锁则不保证这种顺序。
    - 功能丰富：与内置的synchronized关键字相比，ReentrantLock提供了更多的功能，如可中断锁获取、尝试锁获取（可设置超时时间）等。
- synchronized和reentrantLock区别
  - 相同点 — 都是可重入锁
  - 不同点
    - synchronized是Java内置特性，而ReentrantLock是通过Java代码实现的。
    - synchronized是可以自动获取/释放锁的，但是ReentrantLock需要手动获取/释放锁。
    - ReentrantLock还具有响应中断、超时等待等特性。
    - ReentrantLock可以实现公平锁和非公平锁，而synchronized只是非公平锁
      - new ReentrantLock() 默认创建的为非公平锁，如果要创建公平锁可以使用 new ReentrantLock(true)
      - 非公平锁：多个线程不按照申请锁的顺序去获得锁，而是直接去尝试获取锁，获取不到，再进入队列等待，如果能获取到，就直接获取到锁。 — 可以减少CPU唤醒线程的开销，整体的吞吐效率会高点，CPU也不必去唤醒所有线程，会减少唤起线程的数量。但是他可能会导致队列中排队的线程一直获取不到锁或者长时间获取不到锁，活活饿死的情况。
      - 公平锁：多个线程按照申请锁的顺序去获得锁，所有线程都在队列里排队，这样就保证了队列中的第一个先得到锁 — 优点是所有的线程都能得到资源，不会饿死在队列中。但是他存在着吞吐量会下降很多，队列里面除了第一个线程，其他的线程都会阻塞，cpu唤醒阻塞线程的开销会很大的缺点
- JVM实现的synchronized
  - synchronized是怎么实现的
    - JVM 实现的 synchronized — synchronized 的使用方法比较简单，主要可以用来修饰方法和代码块。根据其锁定的对象不同，可以用来定义同步方法和同步代码块。
      - 方法级的同步是隐式的（同步方法）
        - 同步方法的常量池中会有一个 ACC_SYNCHRONIZED 标志
        - 当某个线程要访问某个方法的时候，会检查是否有 ACC_SYNCHRONIZED，如果有设置，则需要先获得监视器锁，然后开始执行方法，方法执行之后再释放监视器锁
        - 这时如果其他线程来请求执行方法，会因为无法获得监视器锁而被阻断住。值得注意的是，如果在方法执行过程中，发生了异常，并且方法内部并没有处理该异常，那么在异常被抛到方法外面之前监视器锁会被自动释放
      - 同步代码块使用 monitorenter 和 monitorexit 两个指令实现
        - 可以把执行 monitorenter 指令理解为加锁，执行 monitorexit 理解为释放锁。
        - 每个对象维护着一个记录着被锁次数的计数器。未被锁定的对象的该计数器为 0，当一个线程获得锁（执行 monitorenter ）后，该计数器自增变为 1 ，当同一个线程再次获得该对象的锁的时候，计数器再次自增。
        - 当同一个线程释放锁（执行 monitorexit 指令）的时候，计数器再自减。当计数器为 0 的时候。锁将被释放，其他线程便可以获得锁。
    - 特点
      - 互斥性 — 同一时间点，只有一个线程可以获得锁，获得锁的线程才可以处理被 synchronized 修饰的代码片段
      - 阻塞性 — 只有获得锁的线程才可以执行被 synchronized 修饰的代码片段，未获得锁的线程只能阻塞，等待锁释放
      - 可重入性 — 如果一个线程已经获得锁，在锁未释放之前，再次请求锁的时候，是必然可以获得锁的
    - synchronized锁的是什么 — synchronized最终锁的都是对象！Java中一切皆对象，类最终也是通过对象的方式呈现的。所以，synchronized的不同用法，只不过锁的对象不同而已。
    - synchronized实现机制
      - Java对象头：在JVM中，每个对象在内存中的布局都包括一个对象头。对象头中包含了两部分数据，其中一部分是标记字段（Mark Word），用于存储对象的运行时数据，如锁状态信息。
      - Monitor（监视器）：每个对象都有一个与之关联的Monitor，它可以被看作是一种同步工具或一种锁。当一个线程尝试进入synchronized块或方法时，它需要获取这个Monitor的锁。
      - 线程状态与锁：当一个线程成功获取Monitor锁时，它可以执行synchronized块或方法中的代码。其他尝试获取该锁的线程将被阻塞，直到锁被释放。
  - synchronized如何保证操作原子性、可见性、有序性
    - 原子性 — 通过 monitorenter 和 monitorexit 这两个字节码指令实现 — 在未释放之前，其他线程是无法再次获得锁的
    - 有序性
      - Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的
      - as-if-serial语义的意思指：不管怎么重排序，单线程程序的执行结果都不能被改变
      - 由于synchronized修饰的代码，同一时间只能被同一线程访问。那么也就是单线程执行的，可保证其有序性
    - 可见性
      - 可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值
      - 对一个变量解锁之前，必须先把此变量同步回主存中。这样解锁后，后续线程就可以访问到被修改后的值。
  - **可重入锁** — 可重入锁是一种特殊的锁机制，它允许同一个线程多次获取同一把锁而不会产生死锁。在Java中，synchronized关键字和ReentrantLock类都支持可重入锁。其主要特点和实现机制如下：
    - 实现机制：通常是通过给每个锁关联一个持有锁的线程和一个计数器来实现。线程首次获得锁时，计数器设为1；每次重入锁时，计数器递增；锁释放时，计数器递减；计数器归零时，锁完全释放。
    - 特点：允许同一个线程多次获取同一把锁，每次获取锁时，锁的计数器会递增；只有当计数器归零时，锁才会被完全释放。
  - synchronized的**锁升级（锁膨胀）**过程是怎样的? 1.6之后
    - 无锁
    - 偏向锁（Biased Locking）jdk15后被删除 — 当一个synchronized块被线程首次进入时，锁对象会进入偏向模式。
    - 轻量级锁（Lightweight Locking） — 当有另一个线程尝试获取已被偏向的锁时，偏向锁会被撤销，锁会升级为轻量级锁。
    - 重量级锁（Heavyweight Locking） — 当轻量级锁的CAS操作失败，即出现了实际的竞争，锁会进一步升级为重量级锁
  - **CAS**
    - 什么是CAS?
      - CAS是一项乐观锁技术，是Compare And Swap的简称，顾名思义就是先比较再替换
      - CAS的主要应用就是实现乐观锁和**锁自旋**。
    - CAS存在什么问题?
      - ABA问题
        - CAS通过比较预期值与内存值来判断数据是否被修改
        - 但如果内存值由A变为B再变回A，CAS会误认为数据未变，这称为ABA问题
      - 循环开销时间大 — CAS通常使用自旋锁不断检查条件，如果条件一直不满足，会持续消耗CPU资源
      - 只能保证一个共享变量的原子操作 — CAS只能保证对一个变量的操作是原子的，对于需要同时操作多个变量的场景不适用
    - CAS在操作系统层面是如何保证原子性的
      - CAS是一种基本的原子操作，用于解决并发问题
      - CAS 操作通常使用 cmpxchg 指令实现
        - cmpxchg 指令是一条原子指令。在 CPU 执行 cmpxchg 指令时，处理器会自动锁定总线，防止其他 CPU 访问共享变量，然后执行比较和交换操作，最后释放总线。
        - cmpxchg 指令在执行期间，CPU 会自动禁止中断。这样可以确保 CAS 操作的原子性，避免中断或其他干扰对操作的影响。
        - cmpxchg 指令是硬件实现的，可以保证其原子性和正确性。CPU 中的硬件电路确保了 cmpxchg 指令的正确执行，以及对共享变量的访问是原子的
  - synchronized的锁优化是怎样的
    - 锁消除（Lock Elision）
    - 锁粗化（Lock Coarsening）
    - 偏向锁（Biased Locking）
    - 适应性自旋锁（Adaptive Locking）
  - synchronized升级过程中有几次自旋? jdk8
    - 第一次自旋 — 第一次自旋发生在 synchronized 获取轻量级锁时，即当一个线程尝试获取一个被其他线程持有的轻量级锁时，它会自旋等待锁的持有者释放锁。
    - 第二次自旋 — 第二次自旋发生在 synchronized 轻量级锁升级到重量级锁的过程中。即当一个线程尝试获取一个被其他线程持有的重量级锁时，它会自旋等待锁的持有者释放锁。
  - synchronized和volatile的区别
    - 作用范围：
      - synchronized可以用于方法、代码块或整个类，实现对共享资源的互斥访问。
      - volatile则作用于变量，用于确保变量的可见性和有序性。
    - 可见性与原子性：
      - volatile关键字确保了一个线程修改了变量的值后，其他线程能立即看到这个修改（可见性）。但它并不能保证原子性，即volatile不能替代synchronized进行锁定的功能。
      - synchronized既能确保可见性，也能确保原子性。因为它通过锁定机制来防止多个线程同时访问和修改共享资源。
    - 线程安全性：
      - synchronized通过锁定机制确保线程安全，即同一时刻只有一个线程可以执行某个方法或代码块，从而防止多个线程同时访问和修改共享资源。
      - volatile并不能保证复合操作的原子性，因此它更多地被用于声明简单类型的变量（如boolean、int等），以确保这些变量的可见性和禁止指令重排序。
    - 性能影响：
      - synchronized会导致线程阻塞和唤醒，这可能带来相对较大的性能开销，尤其是在高并发场景下。
      - volatile则是一种轻量级的同步机制，它不会造成线程的阻塞和唤醒，因此对性能的影响相对较小。
    - 指令重排序：
      - volatile还提供了禁止指令重排序的语义，这有助于确保多线程环境下的程序正确性。
      - synchronized则没有直接提供禁止指令重排序的功能，但它通过锁定机制间接地防止了指令重排序可能导致的线程安全问题。
  - volatile能保证原子性吗? 为什么?
    - 和synchronized不同，volatile是一个变量修饰符，只能用来修饰变量。无法修饰方法及代码块等。
    - volatile在线程安全方面，可以保证有序性和可见性，但是是不能保证原子性的 。
    - 因为他不是锁，他没做任何可以保证原子性的处理。当然就不能保证原子性了。
  - volatile是如何保证可见性和有序性的?
    - 对于volatile变量，当对volatile变量进行写操作的时候，JVM会向处理器发送一条lock前缀的指令，将这个缓存中的变量回写到系统主存中。 — 所以，如果一个变量被volatile所修饰的话，在每次数据变化之后，其值都会被强制刷入主存。而其他处理器的缓存由于遵守了缓存一致性协议，也会把这个变量的值从主存加载到自己的缓存中。这就保证了一个volatile在并发编程中，其值在多个缓存中是可见的。
    - volatile是通过内存屏障来禁止指令重排的，这就保证了代码的程序会严格按照代码的先后顺序执行。
  - 有了synchronized为什么还需要volatile?
    - Synchronized有缺点
      - 产生阻塞 — 无论是同步方法还是同步代码块，无论是ACC_SYNCHRONIZED还是monitorenter、monitorexit都是基于Monitor实现的。基于Monitor对象，当多个线程同时访问一段同步代码时，首先会进入Entry Set，当有一个线程获取到对象的锁之后，才能进行The Owner区域，其他线程还会继续在Entry Set等待。并且当某个线程调用了wait方法后，会释放锁并进入Wait Set等待。所以，synchronize实现的锁本质上是一种阻塞锁。
      - 在同步操作之前还是要进行加锁，同步操作之后需要进行解锁，存在性能损耗
    - volatile借助了内存屏障来帮助其解决可见性和有序性问题
    - volatile还可以禁止指令重排
- **什么是死锁，如何解决**
  - 死锁是指两个或两个以上的进程（或线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。
  - 产生死锁的四个必要条件
    - 互斥条件 — 一个资源每次只能被一个进程使用。
    - 占有且等待 — 一个进程因请求资源而阻塞时，对已获得的资源保持不放。
    - 不可强行占有 — 进程已获得的资源，在末使用完之前，不能强行剥夺。
    - 循环等待条件 — 若干进程之间形成一种头尾相接的循环等待资源关系。
  - 如何解除和预防死锁 — 避免4个条件同时发生
    - 破坏不可抢占：设置优先级，使优先级高的可以抢占资源
    - 破坏循环等待：保证多个进程（线程）的执行顺序相同即可避免循环等待。
  - 避免死锁的方法 — 具体算法或策略：
    - 避免嵌套锁：当需要加锁多个对象时，应该统一它们的锁顺序，尽量避免出现嵌套锁的情况。
    - 使用tryLock()方法：在Java中，可以使用ReentrantLock类的tryLock()方法，并设置超时时间，以避免长时间等待而产生的死锁。
    - 避免无限期等待：为获取锁的操作设置一个超时时间，超时后如果仍未获取到锁，则放弃任务执行。
    - 使用不同的锁：如果可能的话，尝试使用不同的锁来替代原有的锁，以减少锁的竞争和死锁的可能性。
    - 尽量减少锁的持有时间：长时间持有锁会增加死锁的风险，因此应尽量缩短锁的持有时间。
    - 死锁检测工具：利用一些工具（如Java探针）来检测和解决死锁问题。