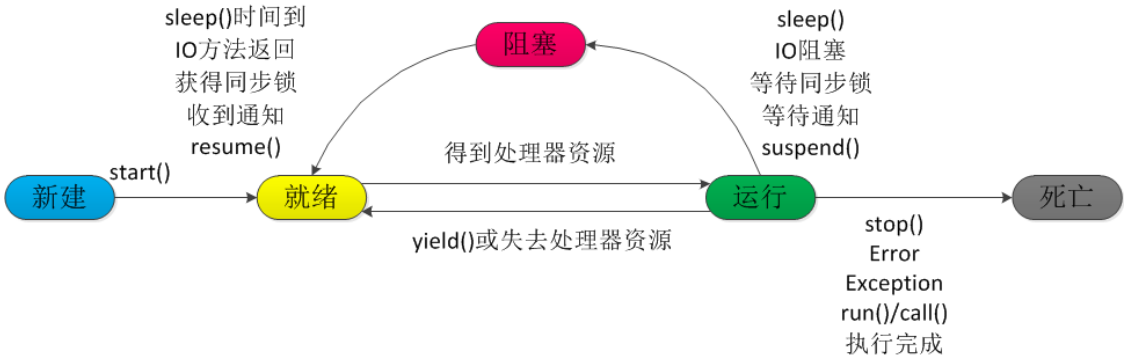
# JAVA多线程并发面试题

## 线程的生命周期？线程有几种状态

- 当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换。
- 


## sleep()、wait()、join()、yield()的区别

**sleep和wait**

- sleep 是 Thread 类的静态本地方法，wait 则是 Object 类的本地方法。
- sleep方法不会释放lock，但是wait会释放，而且会加入到等待队列中。
    - sleep就是把cpu的执行资格和执行权释放出去，不再运行此线程，当定时时间结束再取回cpu资源，参与cpu的调度，获取到cpu资源后就可以继续运行了。而如果sleep时该线程有锁，那么sleep不会释放这个锁，而 是把锁带着进入了冻结状态，也就是说其他需要这个锁的线程根本不可能获取到这个锁。也就是说无法执行程序。如果在睡眠期间其他线程调用了这个线程的interrupt方法，那么这个线程也会抛出interruptexception异常返回，这点和wait是一样的。
- sleep方法不依赖于同步器synchronized，但是wait需要依赖synchronized关键字。
- sleep不需要被唤醒（休眠之后推出阻塞），但是wait需要（不指定时间需要被别人中断）。
- sleep 一般用于当前线程休眠，或者轮循暂停操作，wait 则多用于多线程之间的通信。
- sleep 会让出 CPU 执行时间且强制上下文切换，而 wait 则不一定，wait 后可能还是有机会重新竞争到锁继续执行的。

**yield**

- yield（）执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行

**join**

- join（）执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队列，直到线程A结束或中断线程

## 对线程安全的理解

- 线程安全指的是内存安全，堆是共享内存，可以被所有线程访问。

- 当多个线程访问一个对象时，如果不用进行额外的同步控制或其他的协调操作，调用这个对象的行为都可以获得正确的结果，我们就说这个对象是线程安全的。

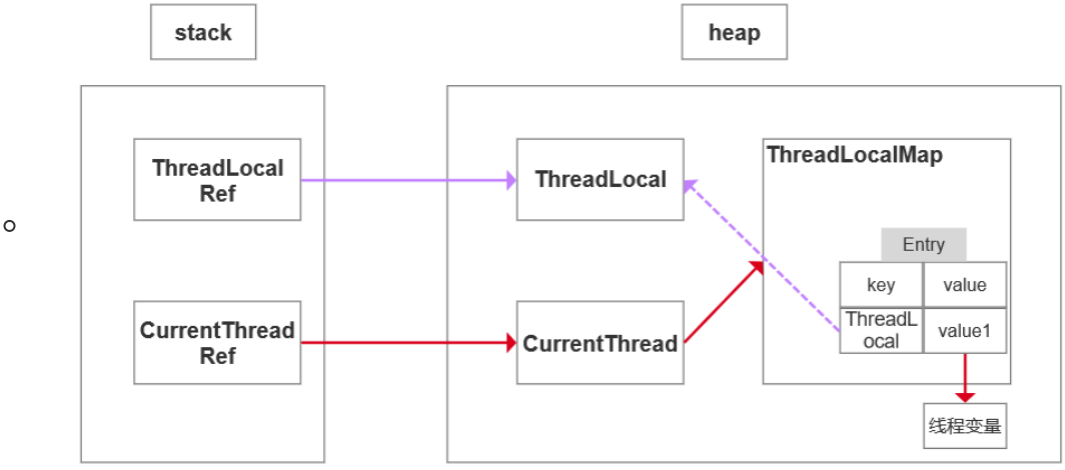
# Thread、Runable的区别

- Thread和Runnable的实质是继承关系，没有可比性。无论使用Runnable还是Thread，都会new Thread，然后执行run方法。用法上，如果有复杂的线程操作需求，那就选择继承Thread，如果只是简 单的执行一个任务，那就实现runnable。

# 对守护线程的理解

- 守护线程：为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；
- 守护线程类似于整个进程的一个默默无闻的小喽喽；它的生死无关重要，它却依赖整个进程而运行；哪天其他线程结束了，没有要执行的了，程序就结束了，理都没理守护线程，就把它中断了；
- 注意： 由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

# ThreadLocal的原理和使用场景

- 每一个 Thread 对象均含有一个 ThreadLocalMap 类型的成员变量 threadLocals ， 它存储本线程中所有ThreadLocal对象及其对应的值
- ThreadLocalMap 由一个个 Entry 对象构成 。
- Entry 继承自 WeakReference<ThreadLocal<?>> ，一个 Entry 由 ThreadLocal 对象和 Object 构成。由此可见， Entry 的key是ThreadLocal对象，并且是一个弱引用。当没指向key的强引用后，该key就会被垃圾收集器回收 。
- 当执行set方法时，ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap对象。再以当前ThreadLocal对象为key，将值存储进ThreadLocalMap对象中。
- get方法执行过程类似。ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap对象。再以当前ThreadLocal对象为key，获取对应的value。
- 由于每一条线程均含有各自**私有的**ThreadLocalMap容器，这些容器相互独立互不影响，因此不会存在线程安全性问题，从而也无需使用同步机制来保证多条线程访问容器的互斥性。
- **使用场景：**
  - 在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。
  - 线程间数据隔离
  - 进行事务操作，用于存储线程事务信息。
  - 数据库连接，Session会话管理。
  - 

# 并发、并行、串行的区别

- 串行在时间上不可能发生重叠，前一个任务没搞定，下一个任务就只能等着
- 并行在时间上是重叠的，两个任务在**同一时刻互不干扰**的同时执行。
- 并发允许两个任务彼此干扰。统一时间点、只有一个任务运行，交替执行

# 并发的三大特性

- **原子性**
- 原子性是指在一个操作中cpu不可以在中途暂停然后再调度，即不被中断操作，要不全部执行完成，要 不都不执行。就好比转账，从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。2个操作必须全部完成。
- **可见性**
- 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。
- 若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。
- **有序性**
- 虚拟机在进行代码编译时，对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按照我们写的代码的顺序来执行，有可能将他们重排序。实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题。

# volatile

- 保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知。
- 禁止指令重排序优化。

# 为什么用线程池?

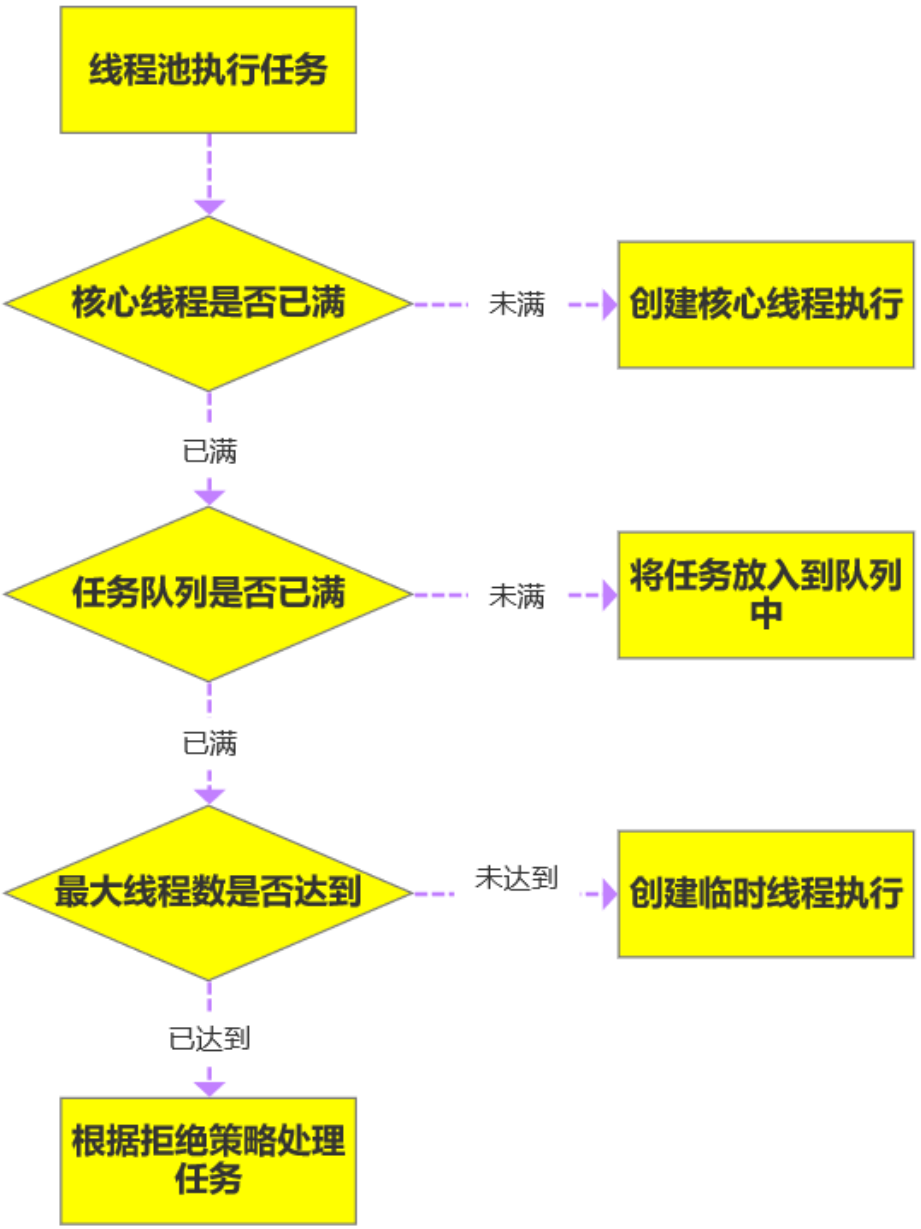
- 降低资源消耗；提高线程利用率，降低创建和销毁线程的消耗。
- 提高响应速度；任务来了，直接有线程可用可执行，而不是先创建线程，再执行。
- 提高线程的可管理性；线程是稀缺资源，使用线程池可以统一分配调优监控。

# 解释下线程池参数?

- corePoolSize 代表核心线程数，也就是正常情况下创建工作的线程数，这些线程创建后并不会消除，而是一种常驻线程
- maxinumPoolSize 代表的是最大线程数，它与核心线程数相对应，表示最大允许被创建的线程数，比如当前任务较多，将核心线程数都用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数
- keepAliveTime 、 unit 表示超出核心线程数之外的线程的空闲存活时间，也就是核心线程不会消除，但是超出核心线程数的部分线程如果空闲一定的时间则会被消除,我们可以通过
- setKeepAliveTime 来设置空闲时间
- workQueue 用来存放待执行的任务，假设我们现在核心线程都已被使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还再持续进入则会开始创建新的线程
- ThreadFactory 实际上是一个线程工厂，用来生产线程执行任务。我们可以选择使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂
- Handler 任务拒绝策略，有两种情况，第一种是当我们调用 shutdown 等方法关闭线程池后，这时候即使线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续想线程

池提交任务就会遭到拒绝。另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，这是也就拒绝

## 简述线程池处理流程




## 线程池中阻塞队列的作用？为什么是先添加列队而不是先创建最大线程?

- 一般的队列只能保证作为一个有限长度的缓冲区，如果超出了缓冲长度，就无法保留当前的任务了，阻塞队列通过阻塞可以保留住当前想要继续入队的任务。
- 阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。
- 阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时,线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活、不至于一直占用cpu资源
- 在创建新线程的时候，是要获取全局锁的，这个时候其它的就得阻塞，影响了整体效率。
- 就好比一个企业里面有10个（core）正式工的名额，最多招10个正式工，要是任务超过正式工人数（task > core）的情况下，工厂领导（线程池）不是首先扩招工人，还是这10人，但是任务可以稍微积压一下，即先放到队列去（代价低）。10个正式工慢慢干，迟早会干完的，要是任务还

在继续增加，超过正式工的加班忍耐极限了（队列满了），就的招外包帮忙了（注意是临时工）要是正式工加上外包还是不能完成任务，那新来的任务就会被领导拒绝了（线程池的拒绝策略）。

# 线程池中线程复用原理

- 线程池将线程和任务进行解耦，线程是线程，任务是任务，摆脱了之前通过 Thread 创建线程时的一个线程必须对应一个任务的限制。
- 在线程池中，同一个线程可以从阻塞队列中不断获取新任务来执行，其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的 run 方法，将 run 方法当成一个普通的方法执行，通过这种方式只使用固定的线程就将所有任务的 run 方法串联起来。

# 简述线程、程序、进程的基本概念。以及他们之间关系是什么?

- **线程**与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。
- **程序**是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。
- **进程**是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。 线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而 各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

# 说说Java中实现多线程有几种方法

- 继承Thread类
- 实现Runnable接口
- 实现Callable接口（JDK1.5>=）
- 线程池方式创建

# 如何停止一个正在运行的线程

- 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。
- 使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
- 使用interrupt方法中断线程。

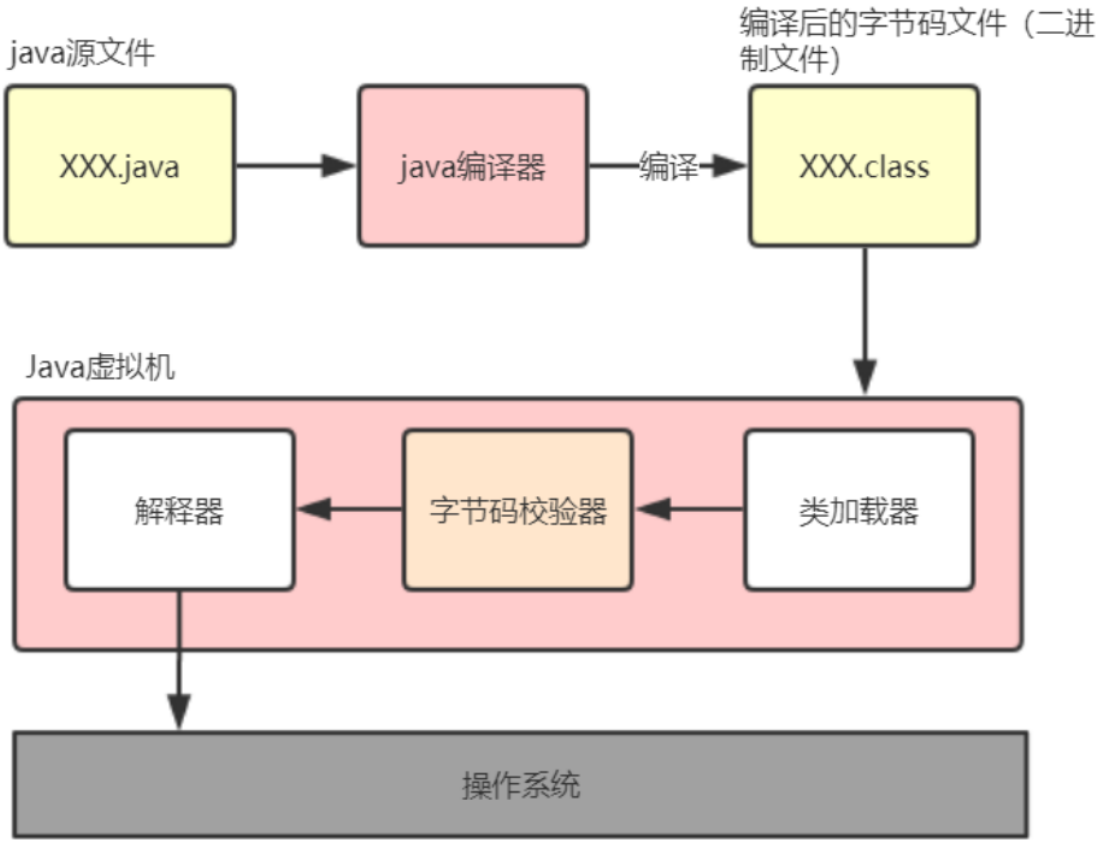
# Thread类中的start()和run()法有什么区别?

- start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

# 常用的线程池有哪些?

- newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线 程达到线程池的最大大小。
- newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线 程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

# Java程序是如何执行的




# 说一下线程之间是如何通信的?

- 线程之间的通信有两种方式：共享内存和消息传递
- **共享内存**
- 在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。典型的共享内存通信方式，就是通过共享对象进行通信。
- 例如上图线程 A 与 线程 B 之间如果要通信的话，那么就必须经历下面两个步骤：
  - 线程 A 把本地内存 A 更新过得共享变量刷新到主内存中去。
  - 线程 B 到主内存中去读取线程 A 之前更新过的共享变量。
- **消息传递**
- 在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。在 Java 中典型的消息传递方式，就是 wait() 和 notify() ，或者 BlockingQueue 。

# 线程池的拒绝策略有哪些?

- AbortPolicy：直接丢弃任务，抛出异常，这是默认策略
- CallerRunsPolicy：只用调用者所在的线程来处理任务
- DiscardOldestPolicy：丢弃等待队列中最旧的任务，并执行当前任务
- DiscardPolicy：直接丢弃任务，也不抛出异常