1. **线程状态：**
   1. **初始(NEW)：**新创建了一个线程对象，但还没有调用start()方法。
   2. **可运行(RUNNABLE)**：也就是就绪状态，调用start()方法之后线程就进入就绪状态， 但是并不是说只要调用start()方法线程就马上变为当前线程，在变为当前线程之前都是为就绪状态。线程在睡眠和挂起中恢复的时候也会进入就绪状态。线程对象创建后，其他线程(比如main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，等待被线程调度选中，获取cpu的使用权。
   3. **运行(RUNNING)**：可运行状态(runnable)的线程获得了cpu 时间片，执行程序代码。线程被设置为当前线程，开始执行run()方法。就是线程进入运行状态
   4. **阻塞(BLOCKED)：**表示线程阻塞于锁。

* 等待阻塞：运行状态的线程执行wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
* 同步阻塞：运行状态的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
* 其他阻塞：运行状态的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
  1. **死亡(DEAD)：**线程执行结束，线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。



1. **线程创建：**
2. **继承Thread类：**重写run()方法，通过Thread调用Start方法启动线程；
3. **实现Runable接口：**实现run()方法，通过start方法启动线程，与第一种效果一样；
4. **实现Callable接口：**实现call()方法，是Executor框架中的功能类；

* 有返回值
* 可以抛出异常
* 使用Future对象获取异步执行结果，监视目标线程

推荐使用实现接口的方法，原因：Java类不支持多重继承，可实现多个接口；继承Theard类开销较大。

1. **线程操作**
2. **start()与run()的区别**

* start是本地方法，它启动一个新线程，使线程处于就绪状态，然后执行run()方法。
* Run方法完成线程实际操作。
* start方法可以异步调用run方法，如果不通过start方法直接调用run方法，会被当做普通函数执行，无法实现多线程。

1. **sleep()与wait()的区别**
2. 原理不同：sleep是Thread的静态方法，控制线程自身流程，线程会自动苏醒；wait是Object类的方法，用于线程通信，需要notify或notifyAll方法唤醒。
3. 锁处理机制：sleep只是暂停线程，不会释放锁（可能导致思索）；wait用于线程通信，会释放锁。
4. 使用区域：sleep可在任何地方使用；wait只能在同步控制方法或同步语句块使用。
5. 捕获异常：sleep必须捕获异常（可能被中断产生InterruptedException），wait、notify、notifyAll不需要捕获异常。
6. **Thread类常用方法**

* static void start();
* static Thread currentThread()：返回当前线程。
* static void sleep(long millis)：线程休眠，转入阻塞状态。
* static void yield()：线程让步给优先级相同或更高的线程，转入可执行状态。
* join()：将两个线程合并实现同步，t.join就是等待t线程完成。用wait方法实现。

（过时）stop()：强制终止线程，释放锁，可能导致程序不确定性。

（过时）suspend():终止线程，不释放锁，可能导致死锁。建议run方法里自然结束线程。

* setDaemon():守护线程，为其他线程提供服务，优先级低，线程启动之前调用。
* void interrupt():向线程发送中断请求，并将中断状态设为true；如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛出InterruptedException，从而提前结束该线程；但是不能中断 I/O 阻塞和 synchronized 锁阻塞。
* static boolean interrupted():测试线程是否处于中断状态，并重置中断状态为false。

1. **线程同步**
2. **什么是线程安全？**

指某个函数、函数库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。即在多线程场景下，不发生有序性、原子性以及可见性问题。

1. **如何保证线程安全？**

Java中主要通过加锁来控制多个线程对共享资源的互斥访问。共有两种方法：

* **Synchronized关键字（JVM实现）：**

为了协调多个线程之间的共享数据访问，虚拟机给每个对象都分配了一个锁，代表在同一时刻只有一个线程可以拥有这个对象。使用synchronized代码给对象加锁，且它会自动释放不必担心死锁，有两种形式：

* + 同步方法：方法声明前加synchronized关键字，方法体大时影响程序执行效率。
  + 同步代码块：synchronized(object)｛｝，可声明任意代码块或对象，灵活性高。
  + wait()方法与notify()方法：必须在同步方法和同步块中使用，否则会抛出 java.lang.IllegalMonitorStateException 异常。执行synchronized代码期间可用wait和notify完成线程间的协作（如交替打印奇偶数）。
* **Lock（JDK实现）：**

JDK5增加Lock接口及其实现类ReentrantLock（重入锁）。如何使用？生成Lock对象并对代码块调用lock()方法，结束后调用unlock()方法。

* lock()：以阻塞方式获得锁，若得不到就一直阻塞，并且忽略中断异常。
* tryLock()：非阻塞方式获得锁，尝试获得锁，得到则返回true，否则返回false
* tryLock(long timeout, TimeUnit unit)：有限时间等待获得锁。
* lockInterruptibly()：没获得锁时处于休眠状态，直到获得锁或者被中断（收到InterruptedException异常）。

**Q．synchronized与Lock的区别**

1. **锁的实现：**synchronized由JVM实现；lock由JDK实现。
2. **使用方法：** synchronized可以锁方法和代码块，是隐式锁，同步操作执行完会自动释放；Lock一般锁代码块，是显式锁，需要手动开启和关闭锁。
3. **等待可中断：**synchronized不能中断，lock可中断。
4. **公平锁：**（指多个线程等待同一个锁，按照时间顺序获得锁）synchronized中的锁是非公平的，lock默认非公平，但也可以公平。
5. **锁绑定多个条件**：lock可以同时绑定多个 Condition 对象。？
6. **Volatile关键字**
7. 定义与使用？

“轻量级synchronized”，只能修饰变量。

* **保证了变量可见性**：被volatile修饰的变量在被修改后会立即同步到主内存（JMM中的说法），当线程需要读取该变量时需要从主内存中刷新。
* **禁止指令重排优化保证有序性：**指令重排会引发有序性问题，被volatile修饰的变量会禁止指令重排，严格按照代码顺序执行。
* **不保证原子性，**要是说能保证，也只是对单个volatile变量的读/写具有原子性，但是对于类似volatile++这样的复合操作就不行了。

1. 如何实现？（内存屏障）

volatile通过内存屏障来实现禁止指令重排和内存可见性问题。

内存屏障（Memory Barrier）：是CPU或编译器在对内存随机访问的操作中的一个同步点，使该点之前的所有读写操作执行之后才可以开始执行该点之后的操作。

* **禁止指令重排**：volatile变量的操作前后插入内存屏障。
* **内存可见性**：在遇到内存屏障的时候，缓存数据会和主存进行同步，或者把缓存数据写入主存、或者从主存把数据读取到缓存。

1. 在哪里会用到volatile？

* 不具备原子性，所以只适用于变量的有效值独立于任何程序的多个变量之间或者某个变量的当前值与修改后值之间没有约束，如可以用于状态标志量，不能用作线程安全计数器。
* 单例模式的实现，典型的双重检查锁定（DCL）

1. 底层实现机制？暂时不看。
2. Synchronized与volatile区别

* synchronized是一种锁机制，存在阻塞问题（多个线程要排队访问同一个共享对象）和性能问题（性能损耗大）。并且synchronized无法禁止指令重排和处理器优化的，它保证的是多个线程之间的有序性，单线程内会产生重排，但编译器的as-if-serial语义保证了单线程的重排不会改变执行结果。
* volatile不是锁，不存在阻塞和性能问题。可以通过内存屏障来禁止指令重排。

1. synchronized实现原理及锁优化
2. synchronized实现原理
3. 锁优化
4. Lock实现原理
5. **线程池（还没看）**
   1. 为什么要使用线程池
6. 重用存在的线程，减少线程对象创建、消亡的开销，性能佳。
7. 可有效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。
8. 提供定时执行、定期执行、单线程、并发数控制等功能。
   1. Java 线程池：Java通过Executors提供四种线程池
      1. newCachedThreadPool创建一个可缓存线程池，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。
      2. newFixedThreadPool 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中等待。
      3. newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
      4. 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。
   2. **java.uitl.concurrent.ThreadPoolExecutor**类是线程池中最核心的一个类，继承了AbstractExecutorService类，并提供了四个构造器。
      1. **corePoolSize**：核心池的大小，在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
      2. **maximumPoolSize**：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
      3. **keepAliveTime**：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
      4. **unit**：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：
         1. TimeUnit.DAYS; //天
         2. TimeUnit.HOURS; //小时
         3. TimeUnit.MINUTES; //分钟
         4. TimeUnit.SECONDS; //秒
         5. TimeUnit.MILLISECONDS; //毫秒
         6. TimeUnit.MICROSECONDS; //微妙
         7. TimeUnit.NANOSECONDS; //纳秒
      5. **workQueue**：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：
         1. ArrayBlockingQueue；
         2. LinkedBlockingQueue；
         3. SynchronousQueue；
      6. **threadFactory**：线程工厂，主要用来创建线程；
      7. **handler**：表示当拒绝处理任务时的策略，有以下四种取值：
         1. ThreadPoolExecutor.AbortPolicy：丢弃任务并抛出RejectedExecutionException异常。
         2. ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
         3. ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
         4. ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务
      8. **execute()**：通过这个方法可以向线程池提交一个任务，交由线程池去执行。
      9. **submit()**：用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，submit()方法的实现，还是调用的execute()方法，但利用Future来获取任务执行结果。
      10. **shutdown()**：关闭线程池
      11. **shutdownNow()**：关闭线程池
   3. 线程池的具体实现原理，将从下面几个方面讲解：
      1. 线程池状态：RUNNING、SHUTDOWN、STOP、TERMINATED
         1. volatile int runState：用来表示线程池的状态
         2. volatile类型修饰符，修饰被不同线程访问和修改的变量。系统每次使用该变量时，直接从内存中读取，不使用缓存。避免缓存与存储不一致的情况。不能保证操作的原子性，影响程序的执行效率。
      2. 任务的执行
      3. 线程池中的线程初始化
      4. 任务缓存队列及排队策略
      5. 任务拒绝策略
      6. 线程池的关闭
      7. 线程池容量的动态调整
   4. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？
      1. 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
      2. 并发不高、任务执行时间长的业务要区分开看：

a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（i）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些是否能做数据缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（ii）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。