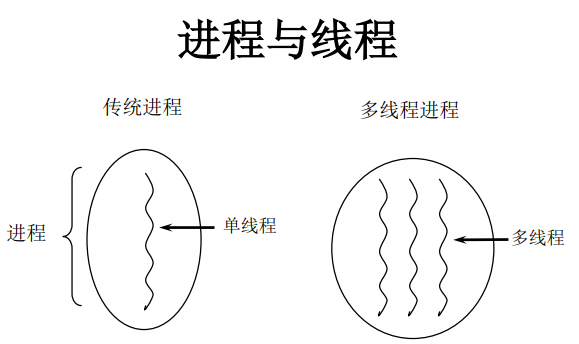
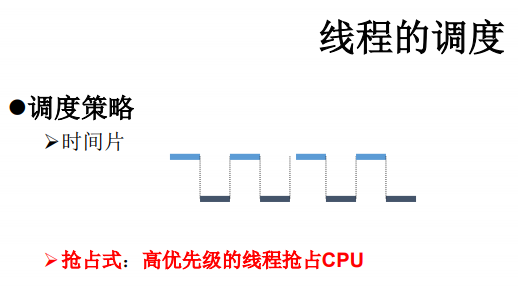
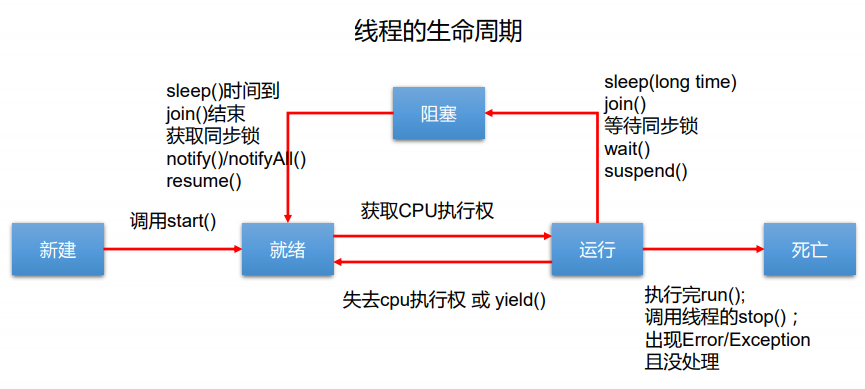
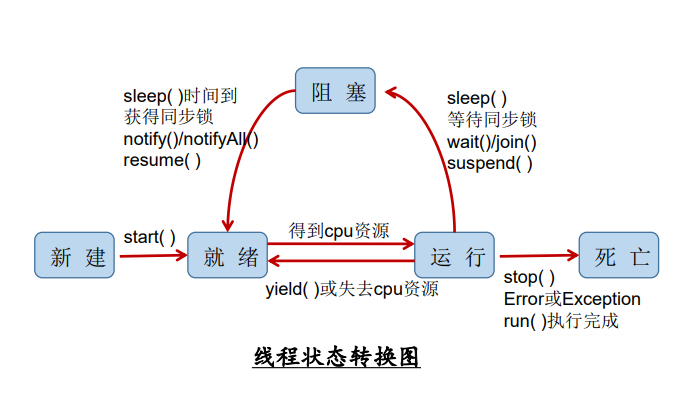
1. 创建多线程有3种方式，分别是1、[继承线程类](https://how2j.cn/k/thread/thread-start/353.html" \l "step778)2、[实现Runnable接口](https://how2j.cn/k/thread/thread-start/353.html" \l "step779)3、[匿名类](https://how2j.cn/k/thread/thread-start/353.html" \l "step780)
2. 匿名类：使用[匿名类](https://how2j.cn/k/interface-inheritance/interface-inheritance-inner-class/322.html#step687)，继承Thread,重写run方法，直接在run方法中写业务代码，匿名类的一个好处是可以很方便的访问外部的局部变量。
3. 注意： 启动线程是start()方法，run()并不能启动一个新的线程
4. 守护线程的概念是： 当一个进程里，所有的线程都是守护线程的时候，结束当前进程。守护线程通常会被用来做日志，性能统计等工作。
5. 如果一个类，其方法都是有synchronized修饰的，那么该类就叫做线程安全的类
6. Synchronzied
   1. Synchronzied 修饰非静态方法==》对象锁，该对象内所有的加锁的非静态方法共用这一把锁, 一个加锁非静态方法执行, 另一个加锁非静态方法不能执行,要等持有锁的线程释放锁, 不同对象之间的方法不互相作用
   2. Synchronzied 修饰静态方法==》其实是类锁，因为是静态方法，它把整个类锁起来了；该类内所有的加锁的静态方法共用这一把锁, 一个加锁静态方法执行, 同类另一个加锁静态方法不能执行,要等持有锁的线程释放锁
   3. 注意：静态与非静态方法不共用锁
7. Synchronized修饰非静态方法，实际上是对 调用该方法的对象 加锁，俗称“对象锁”。
   1. 解释：Java中每个对象都有一个锁，并且是唯一的。假设分配的一个对象空间，里面有多个方法，相当于空间里面有多个小房间，如果我们把所有的小房间都加锁，因为这个对象只有一把钥匙，因此同一时间只能有一个人打开一个小房间，然后用完了还回去，再由JVM 去分配下一个获得钥匙的人。
8. 情况1：同一个对象在两个线程中分别访问该对象的两个同步方法
   1. 结果：会产生互斥。
   2. 解释：因为锁针对的是对象，当对象调用一个synchronized方法时，其他同步方法需要等待其执行结束并释放锁后才能执行。
9. 情况2：不同对象在两个线程中调用同一个同步方法
   1. 结果：不会产生互斥。
   2. 解释：因为是两个对象，锁针对的是对象，并不是方法，所以可以并发执行，不会互斥。形象的来说就是因为我们每个线程在调用方法的时候都是new 一个对象，那么就会出现两个空间，两把钥匙。
10. Synchronized修饰静态方法，实际上是对该类对象加锁，俗称“类锁”。
    1. 情况1：用类直接在两个线程中调用两个不同的同步方法
       1. 结果：会产生互斥。
       2. 解释：因为对静态对象加锁实际上对类（.class）加锁，类对象只有一个，可以理解为任何时候都只有一个空间，里面有N个房间，一把锁，因此房间（同步方法）之间一定是互斥的。
       3. 注意：上述情况和用单例模式声明一个对象来调用非静态方法的情况是一样的，因为永远就只有这一个对象。所以访问同步方法之间一定是互斥的。
    2. 情况2：用一个类的静态对象在两个线程中调用静态方法或非静态方法
       1. 结果：会产生互斥。
       2. 解释：因为是一个对象调用，同上。
    3. 情况3：一个对象在两个线程中分别调用一个静态同步方法和一个非静态同步方法
       1. 结果：不会产生互斥。
       2. 解释：因为虽然是一个对象调用，但是两个方法的锁类型不同，调用的静态方法实际上是类对象在调用，即这两个方法产生的并不是同一个对象锁，因此不会互斥，会并发执行。
11. Synchronized方法与Synchronized代码块：
    1. synchronized methods() {} 与 synchronized (this) {} 之间并没有什么区别。只是前者便于阅读理解，而后者可以更精确的控制冲突限制访问区域，有时候表现得更加高效。
12. Synchronized（非this对象）：
    1. 这个"非this对象"大多数是实例变量及方法的参数，可以是任意的
    2. 其实同理，锁住的不是当前实例对象，而是放入synchronized(非this对象)中的非this对象，即对该对象进行加锁。
    3. 优点：synchronized(非this对象)代码块中的程序与同步方法是异步的，不与其他锁this同步方法争抢this锁，大大提高了运行效率。
    4. 注意：非this对象，这个对象如果是实例变量的话，指的是对象的引用，只要对象的引用不变，即使改变了对象的属性，运行结果依然是同步的。
13. 对象锁钥匙只能有一把才能互斥，才能保证共享变量的唯一性
14. 在静态方法上的锁，和 实例方法上的锁，默认不是同样的，如果同步需要制定两把锁一样。
15. 关于同一个类的方法上的锁，来自于调用该方法的对象，如果调用该方法的对象是相同的，那么锁必然相同，否则就不相同。比如 new A().x() 和 new A().x(),对象不同，锁不同，如果A的单例的，就能互斥。
16. 静态方法加锁，能和所有其他静态方法加锁的 进行互斥
17. 静态方法加锁，和xx.class 锁效果一样，直接属于类的
18. 借助Collections.synchronizedList，可以把ArrayList转换为线程安全的List。
19. 调用wait是有前提条件的，一定是在synchronized块里，否则就会出错。
20. 首先通过lock对象得到一个Condition对象，然后分别调用这个Condition对象的：await, signal,signalAll 方法。
21. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现，Lock是代码层面的实现。
22. Lock可以选择性的获取锁，如果一段时间获取不到，可以放弃。synchronized不行，会一根筋一直获取下去。 借助Lock的这个特性，就能够规避死锁，synchronized必须通过谨慎和良好的设计，才能减少死锁的发生。
23. synchronized在发生异常和同步块结束的时候，会自动释放锁。而Lock必须手动释放， 所以如果忘记了释放锁，一样会造成死锁。
24. 程序：完成特定任务的指令集合，一段静态的代码，静态对象
25. 进程：程序的一次执行过程，或正在运行的一个程序，是一个动态过程，有产生，存在，消亡的过程
26. 线程：进程进一步细化为线程，是一个程序内部的一条执行路径
    1. 若一个进程同一时间并行执行多个线程，就是支持多线程的
    2. 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小
    3. 一个进程中的多个线程共享相同的内存单元/内存地址空间，它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。
27. 
28. 一个Java应用程序java.exe，其实至少有三个线程：main()主线程，gc()垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。
29. 并行与并发
    1. 并行：多个CPU同时执行多个任务
    2. 并发：一个CPU(采用时间片)同时执行多个任务。
30. Thread类构造器：
    1. Thread()：创建新的Thread对象
    2. Thread(String threadname)：创建线程并指定线程实例名
    3. Thread(Runnable target)：指定创建线程的目标对象，它实现了Runnable接中的run方法
    4. Thread(Runnable target, String name)：创建新的Thread对象
31. 创建线程的俩种方式：
    1. 继承Thread类
       1. 定义子类继承Thread类
       2. 子类中重写Thread类中的run方法。
       3. 创建Thread子类对象，即创建了线程对象。
       4. 调用线程对象start方法：启动线程，调用run方法。
32. 注意点：
    1. run()方法由JVM调用，什么时候调用，执行的过程控制都有操作系统的CPU调度决定。
    2. 想要启动多线程，必须调用start方法。
    3. 一个线程对象只能调用一次start()方法启动，如果重复调用了，则将抛出以上的异常“IllegalThreadStateException”。
33. 实现Runnable接口
    1. 定义子类，实现Runnable接口。
    2. 子类中重写Runnable接口中的run方法。
    3. 通过Thread类含参构造器创建线程对象。
    4. 将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。
    5. 调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。
    6. 好处：
       1. 避免了单继承的局限性
       2. 多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。
34. Thread类的有关方法
    1. void start(): 启动线程，并执行对象的run()方法
    2. run(): 线程在被调度时执行的操作
    3. String getName(): 返回线程的名称
    4. void setName(String name):设置该线程名称
    5. static Thread currentThread(): 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类
    6. static void yield()：线程让步
       1. 暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程
       2. 若队列中没有同优先级的线程，忽略此方法
    7. join() ：当某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
       1. 低优先级的线程也可以获得执行
    8. static void sleep(long millis)：(指定时间:毫秒)
       1. 令当前活动线程在指定时间段内放弃对CPU控制,使其他线程有机会被执行,时间到后重排队。
       2. 抛出InterruptedException异常
    9. boolean isAlive()：返回boolean，判断线程是否还活着
35. 
36. Java的调度方法
    1. 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
    2. 对高优先级，使用优先调度的抢占式策略
37. 线程优先级：
    1. MAX\_PRIORITY：10
    2. MIN \_PRIORITY：1
    3. NORM\_PRIORITY：5
    4. 涉及方法：
       1. getPriority() ：返回线程优先值
       2. setPriority(int newPriority) ：改变线程的优先级
38. 注意：
    1. 线程创建时继承父线程的优先级
    2. 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用
39. 线程分类：守护线程、用户线程
    1. 守护线程是用来服务用户线程的，通过在start()方法前调用thread.setDaemon(true)可以把一个用户线程变成一个守护线程。
    2. Java垃圾回收就是一个典型的守护线程。
40. Thread.State类定义了线程的几种状态
    1. 新建： 当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
    2. 就绪：处于新建状态的线程被start()后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
    3. 运行：当就绪的线程被调度并获得CPU资源时,便进入运行状态， run()方法定义了线程的操作和功能
    4. 阻塞：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU并临时中止自己的执行，进入阻塞状态
    5. 死亡：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束
    6. 
    7. 
41. 线程同步机制之Synchronized
    1. 同步代码块：synchronized (对象){ // 需要被同步的代码；}
    2. 同步方法：public synchronized void show (String name){ …. }
    3. Synchronized的锁是什么？
       1. 任意对象都可以作为同步锁。所有对象都自动含有单一的锁（监视器）。
       2. 同步方法的锁：静态方法（类名.class）、非静态方法（this）
       3. 同步代码块：自己指定，很多时候也是指定为this或类名.class
42. 注意：
    1. 必须确保使用同一个资源的多个线程共用一把锁，这个非常重要，否则就无法保证共享资源的安全
    2. 一个线程类中的所有静态方法共用同一把锁（类名.class），所有非静态方法共用同一把锁（this），同步代码块（指定需谨慎）
43. 释放锁操作：
    1. 当前线程的同步方法、同步代码块执行结束。
    2. 当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。
    3. 当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束。
    4. 当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。
44. 不会释放锁的操作：
    1. 线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行
    2. 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。
       1. 应尽量避免使用suspend()和resume()来控制线程
45. 死锁问题：
    1. 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
    2. 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续
46. Lock锁
    1. 通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。
    2. java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。
    3. ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。
47. 
48. Synchronized 与 Lock 的对比
    1. Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁，出了作用域自动释放
    2. Lock只有代码块锁，synchronized有代码块锁和方法锁
    3. 使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）
49. 线程通信：
    1. wait() 与 notify() 和 notifyAll()
       1. wait()：令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程notify()或notifyAll()方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行。
       2. notify()：唤醒正在排队等待同步资源的线程中优先级最高者结束等待
       3. notifyAll ()：唤醒正在排队等待资源的所有线程结束等待.
    2. 这三个方法只有在synchronized方法或synchronized代码块中才能使用，否则会报java.lang.IllegalMonitorStateException异常。
    3. 因为这三个方法必须有锁对象调用，而任意对象都可以作为synchronized的同步锁，因此这三个方法只能在Object类中声明。
50. wait() 方法：
    1. 在当前线程中调用方法： 对象名.wait()
    2. 使当前线程进入等待（某对象）状态 ，直到另一线程对该对象发出 notify (或notifyAll) 为止。
    3. 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
    4. 调用此方法后，当前线程将释放对象监控权 ，然后进入等待
    5. 在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。
51. notify()/notifyAll()：
    1. 在当前线程中调用方法： 对象名.notify()
    2. 功能：唤醒等待该对象监控权的一个/所有线程。
    3. 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
52. 新增的线程创建方式一：实现Callable接口
    1. 相比run()方法，可以有返回值
    2. 方法可以抛出异常
    3. 支持泛型的返回值
    4. 需要借助FutureTask类，比如获取返回结果
53. Future接口：
    1. 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
    2. FutrueTask是Futrue接口的唯一的实现类
    3. FutureTask 同时实现了Runnable, Future接口。它既可以作为
    4. Runnable被线程执行，又可以作为Future得到Callable的返回值
54. 新增的线程创建方式二：
    1. 提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。
    2. ExecutorService 和 Executors
55. ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor
    1. void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable
    2. <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
    3. void shutdown() ：关闭连接池
56. Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池
    1. Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
    2. Executors.newFixedThreadPool(n); 创建一个可重用固定线程数的线程池
    3. Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池
    4. Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。