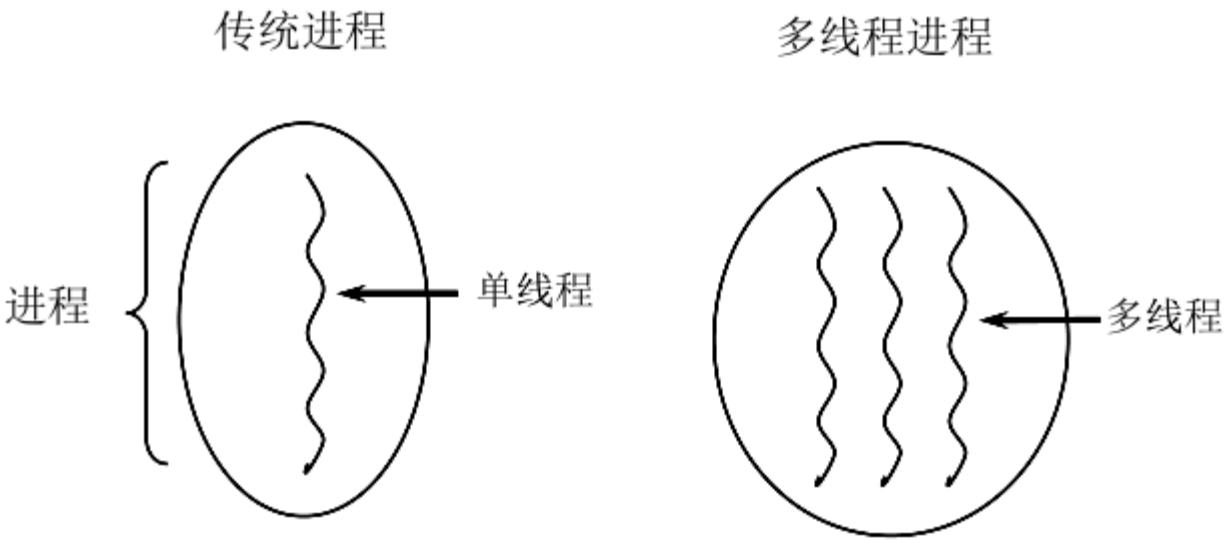
线程

程序、进程、线程

- **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
- **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
  - 如：运行中的QQ，运行中的MP3播放器
  - 程序是静态的，进程是动态的
  - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- **线程(thread)**，进程可进一步细化为线程，是一个程序内部的一条执行路径。
  - 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
  - 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
  - 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。

进程与线程



进程与线程

- **单核CPU和多核CPU的理解**
  - 单核CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来。
  - 如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）
  - 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。
- **并行与并发**
  - **并行：**多个CPU同时执行多个任务。比如：多个人同时做不同的事。
  - **并发：**一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多个人做同一件事。

使用多线程的优点

背景： 以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1. 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。
2. 提高计算机系统CPU的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。

API中创建线程的两种方式

- JDK1.5之前创建新执行线程有两种方法：
  - 继承Thread类的方式
  - 实现Runnable接口的方式

- 方式一：**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。

进程与线程

```java
/**
 * 多线程的创建，方式一：继承于Thread类
 * 1. 创建一个继承于Thread类的子类
 * 2. 重写Thread类的run() --> 将此线程执行的操作声明在run()中
 * 3. 创建Thread类的子类的对象
 * 4. 通过此对象调用start()
 */
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0) {
                System.out.println(Thread.currentThread().getName() + " ----- " + i);
            }
        }
    }
}

public class ThreadTest{
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

注意：1.启动线程必须是调用start方法而不是直接调用run方法
2.如果要再启动一个线程需要重新创建对象

API中创建线程的两种方式

- **方式二：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。

进程与线程

```java
/**
 * 创建多线程的方式二：实现Runnable接口
 * 1. 创建一个实现了Runnable接口的类
 * 2. 实现类去实现Runnable中的抽象方法：run()
 * 3. 创建实现类的对象
 * 4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
 * 5. 通过Thread类的对象调用start()
 */
class MyThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 != 0) {
                System.out.println(Thread.currentThread().getName() + " " + i);
            }
        }
    }
}
public class ThreadTest {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        Thread t1 = new Thread(myThread);
        t1.start();
    }
```

进程与线程

比较两种创建线程的方式

1. 实现的方式没有类的单继承性的局限性
2. 实现的方式更适合来处理多个线程有共享数据的情况

联系：public class Thread implements Runnable
相同点：两种方式都需要重写run(),将线程要执行的逻辑声明在run()中。

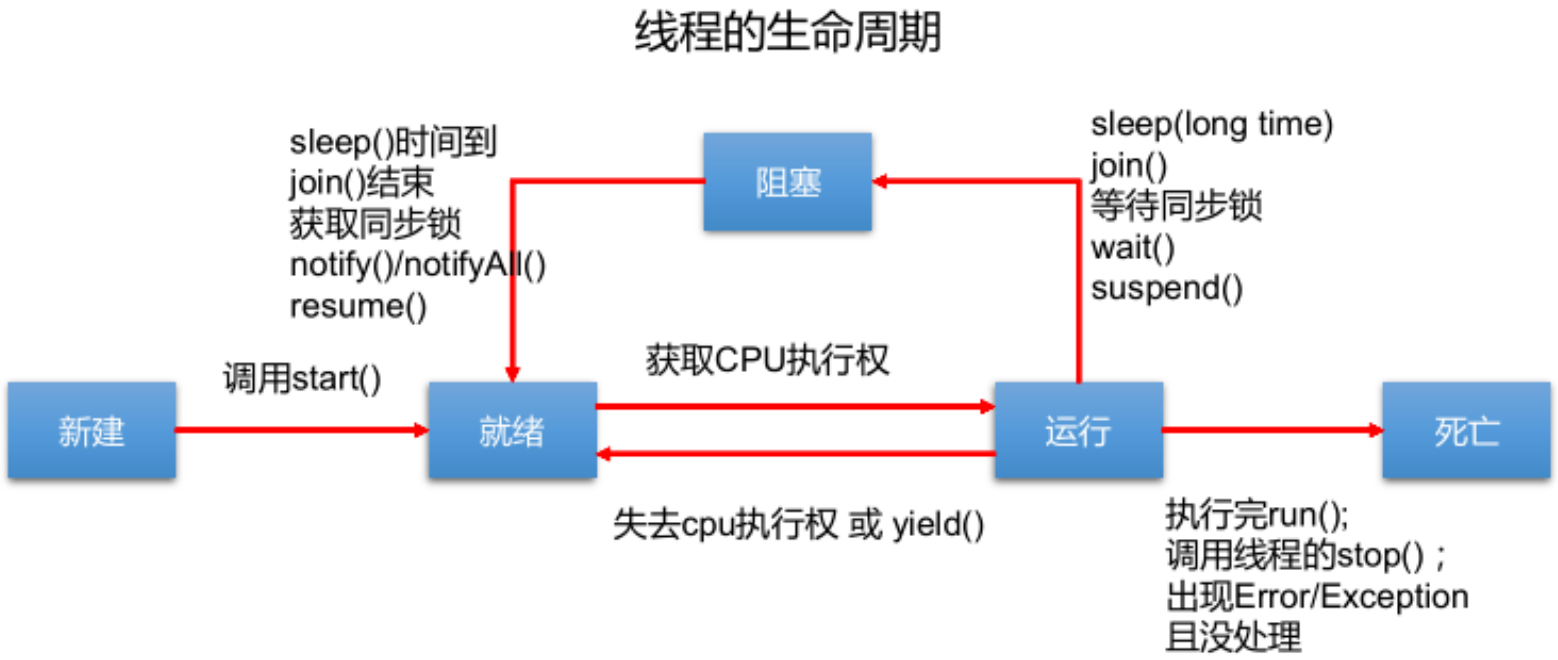
Thread类的有关方法(1)

- **void start():** 启动线程，并执行对象的run()方法
- **run():** 线程在被调度时执行的操作
- **String getName():** 返回线程的名称
- **void setName(String name)**:设置该线程名称
- **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

Thread类的有关方法(2)

- **static void yield()**：线程让步
  - 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
  - 若队列中没有同优先级的线程，忽略此方法
- **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
  - 低优先级的线程也可以获得执行
- **static void sleep(long millis)**：(指定时间:毫秒)
  - 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。
  - 抛出InterruptedException异常
- **stop()**: 强制线程生命期结束，不推荐使用
- **boolean isAlive()**：返回boolean，判断线程是否还活着

进程与线程


线程的生命周期
sleep()时间到
join()结束
获取同步锁
notify()/notifyAll()
resume()
阻塞
sleep(long time)
join()
等待同步锁
wait()
suspend()
调用start()
获取CPU执行权
新建
就绪
运行
死亡
失去cpu执行权 或 yield()
执行完run();
调用线程的stop()；
出现Error/Exception
且没处理

线程同步

线程安全的问题的提出

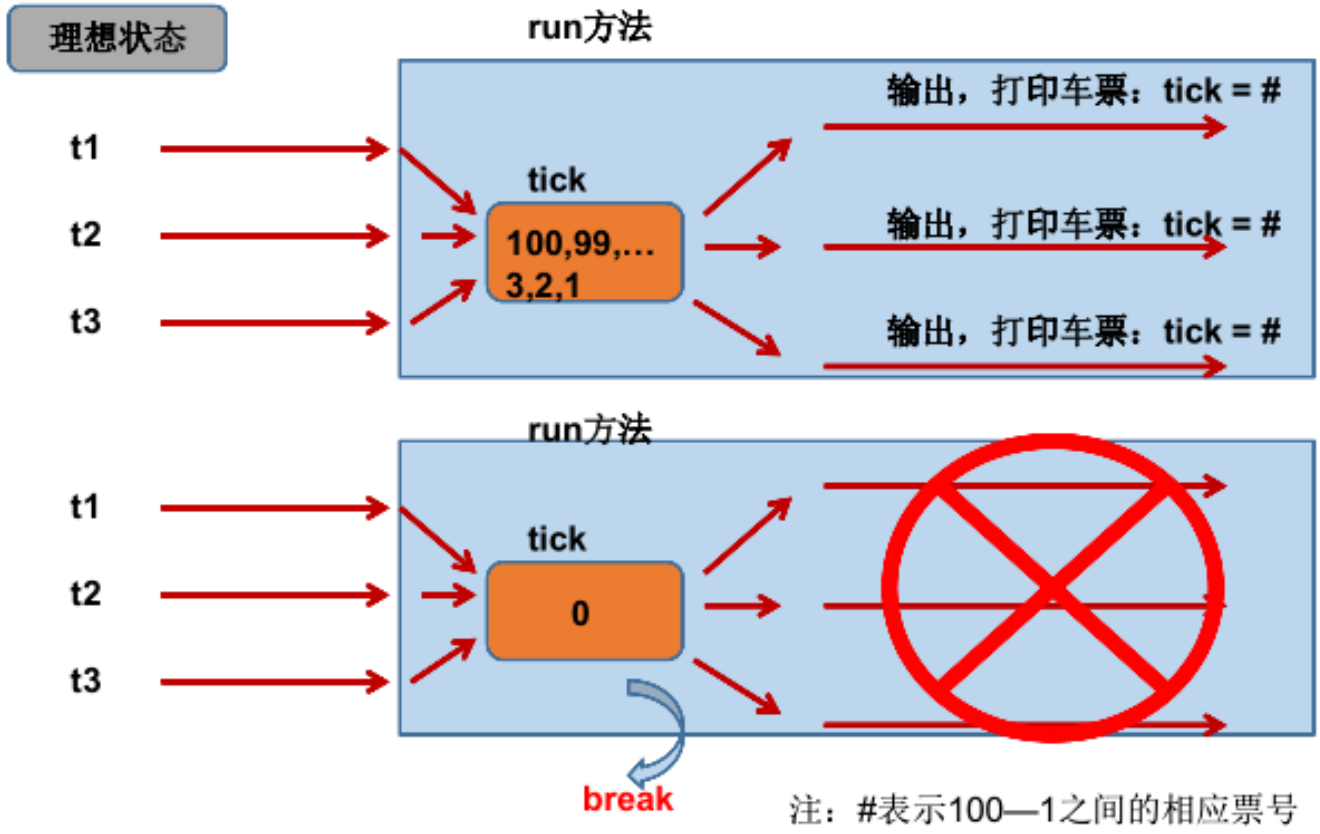
1.多个线程执行的不确定性引起执行结果的不稳定性

2.多个线程数据共享，会破坏数据

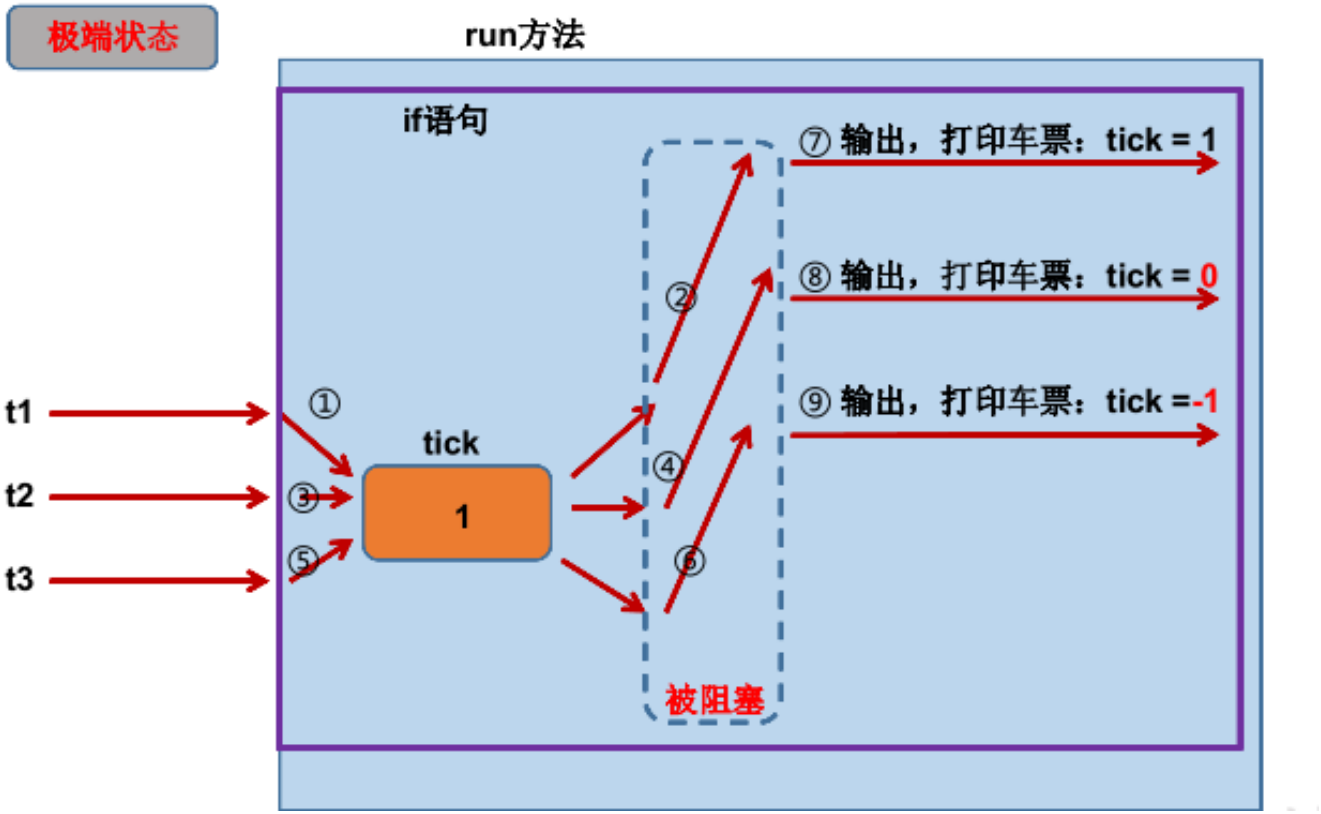
```
class Window implements Runnable {
    private int tickets = 100;
    @Override
    public void run() {
        while (true) {
            if (tickets > 0) {
                System.out.println(Thread.currentThread().getName() + ":卖票，票号为" + tickets);
                tickets --;
            } else { break; }
        }
    }
}

public class SellTicket {
    public static void main(String[] args) {
        Window window = new Window();
        Thread t1 = new Thread(window); Thread t2 = new Thread(window); Thread t3 = new Thread(window);
        t1.setName("窗口一"); t2.setName("窗口二");  t3.setName("窗口三");
        t1.start(); t2.start(); t3.start();
    }
}
```

线程同步


理想状态
run方法
t1
t2
t3
tick
100,99,…
3,2,1
输出，打印车票：tick = #
输出，打印车票：tick = #
输出，打印车票：tick = #
run方法
t1
t2
t3
tick
0
break
注：#表示100—1之间的相应票号

线程同步



1. 多线程出现了安全问题
2. 问题 的原因：

当多线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

3. 解决办法：

操作共享数据的语句，只能让一个线程都执行完，在执行过程中，即使阻塞，其他线程也不可以参与执行。

线程同步

Java中解决线程安全问题的方式：通过同步机制

方式一：同步代码块

```
synchronized(同步监视器){
    //需要被同步的代码
}
```

说明：

1.操作共享数据的代码，即为需要被同步的代码。 -->不能包含代码多了，也不能包含代码少了。

2.共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。

3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。

要求：多个线程必须要共用同一把锁。

补充：在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器。

线程同步

Java中解决线程安全问题的方式：通过同步机制

方式二：同步方法。

如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的。

* 关于同步方法的总结：
* 1. 同步方法仍然涉及到同步监视器，只是不需要我们显式的声明。
* 2. 非静态的同步方法，同步监视器是：this
* 静态的同步方法，同步监视器是：当前类本身

同步的方式，解决了线程的安全问题。---好处

操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。

线程的死锁问题

- 死锁
  - 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
  - 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
- 解决方法
  - 专门的算法、原则
  - 尽量减少同步资源的定义
  - 尽量避免嵌套同步

线程同步

- 从JDK 5.0开始，Java提供了更强大的线程同步机制——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。
- 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。
- 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。

线程同步

```
class A{
        private final ReentrantLock lock = new ReenTrantLock();
        public void m(){
                lock.lock();
                try{
                        //保证线程安全的代码;
                }
                finally{
                        lock.unlock();
                }
        }
}
```

注意：如果同步代码有异常，要将unlock()写入finally语句块

线程同步

1. 面试题：synchronized 与 Lock的异同？
   相同：二者都可以解决线程安全问题
   不同：synchronized机制在执行完相应的同步代码以后，自动的释放同步监视器；Lock需要手动的启动同步(lock())，同时结束同步也需要手动的实现unlock()

2.优先使用顺序：
   Lock
   同步代码块（已经进入了方法体，分配了相应资源）
   同步方法（在方法体之外）

线程通信

- **wait() 与 notify() 和 notifyAll()**
  - **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调用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。
  - **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
  - **notifyAll ()**：唤醒正在排队等待资源的所有线程结束等待.
- 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。
- 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。

* 说明：
 * 1.wait()：一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。
 * 2.notify()：一旦执行此方法，就会唤醒被wait的一个线程。如果有多个线程被wait()，就唤醒优先级高的那个。
* 3.notifyAll():一旦执行此方法，就会唤醒所有被wait的线程。

线程通信

面试题：sleep() 和 wait()的异同？
* 1.相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。
* 2.不同点：
1）两个方法声明的位置不同：Thread类中声明sleep() , Object类中声明wait()
2）调用的要求不同：sleep()可以在任何需要的场景下调用。 wait()必须使用在同步代码块或同步方法中
3）关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。
*

经典问题：生产者/消费者 大家自己实现

线程通信

JDK5 新增创建线程的方式

1.实现Callable接口

与使用Runnable相比， Callable功能更强大些

相比run()方法，可以有返回值

方法可以抛出异常

支持泛型的返回值

需要借助FutureTask类，比如获取返回结果

Future接口

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。

FutrueTask是Futrue接口的唯一的实现类

FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

线程通信

JDK5 新增创建线程的方式

2.使用线程池

使用线程池的好处：池的思想，类比数据库连接池。

- **背景：**经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- **思路：**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。类似生活中的公共交通工具。
- **好处：**
  - 提高响应速度（减少了创建新线程的时间）
  - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  - 便于线程管理
    - ✓ corePoolSize：核心池的大小
    - ✓ maximumPoolSize：最大线程数
    - ✓ keepAliveTime：线程没有任务时最多保持多长时间后会终止
    - ✓ …

线程通信

JDK5 新增创建线程的方式

2.使用线程池

线程池相关API

- JDK 5.0起提供了线程池相关API：**ExecutorService** 和 **Executors**
- ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor
  - void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable
  - <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
  - void shutdown() ：关闭连接池
- 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
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(n); 创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池
  - 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。