进程：

多线程：

线程的创建，运行，阻塞，死亡

线程阻塞的情况：

1. 线程调用sleep方法主动放弃所暂用的处理器资源
2. 线程调用了一个阻塞式的I/O方法，在改方法返回之前，线程都是阻塞的
3. 线程试图获取一个同步的监视器，改同步监视器正在被其他线程使用
4. 线程正在等待某个通知notify
5. 线程调用了线程的suspend方法，将该线程挂起。该方法容易导致死锁

线程死亡状态：

1.run结束后，线程正常结束

2.线程发生未捕获的异常或者error

3.线程调用了stop方法，该方法容易导致死锁

控制线程：

Join()方法：当某个程序执行流中调用join方法时。调用线程将被阻塞，直到被join线程执行结束后

后台线程：如果所有的前台线程死亡，后台线程也是随着死亡

setDaemon(true)可把指定线程设置为后台线程

前台线程创建的子线程默认为前台线程。后台线程创建的子线程默认为后台线程

线程睡眠sleep 让当前执行线程暂停一段时间，并进入阻塞状态

线程让步yield 让当前执行线程暂停，但不会进入阻塞状态，该线程转入就绪状态

Sleep 方法与yield方法的区别

1.sleep暂停后，会给其他线程机会。而yield只会给你比他优先级高或者同级的线程机会

2.sleep方法时线程进入阻塞状态。只有经过阻塞时间后才会进入就绪状态；而yield是不会进入阻塞状态，只是强制当前线程进入就绪状态。

3.sleep声明抛出InterruptedException ，而yield没有声明任何异常

4.sleep相比yield更有可移植性

线程安全

同步代码块synchronized(obj) 给代码块加上同步监视器obj

同步方法 synchronized修饰方法，this对象就是同步监视器

线程安全的类：

1.该类的对象可以被多个线程安全的访问

2.每个线程调用该类的任何方法后，都将得到正确的结果

3.每个线程调用该类的任何方法后，该类对象状态依然保持合理状态

可变类的线程安全是以降低程序的运行效率为代价的

1.不对可变类的所有方法都使用同步，只对改变竞争资源的方法进行同步

2.如果可变类有两种运行环境：单线层环境和多线程环境，则应该为可变类提供多个版本，线程不安全版本和线程安全版本，单线程环境使用线程不安全版本保证性能

释放同步监视器的锁定

1.同步代码块或者同步方法执行结束，当前线程即释放同步监视器

2.当同步代码块或者同步方法中执行return或者break终止该代码块，方法的执行，当前线程即释放同步监视器

3.发生异常或者error，导致代码块，方法异常结束

4.当线程执行了同步方法快或者同步方法时，程序调用了监视器的wait方法，则当前线程暂停，释放同步监视器

不会释放同时监视器的情况：

1.当执行线程的sleep。Yield方法时来暂停当前线程的执行，当前线程不会释放同步监视器

2.当其他线程调用该线程的suspend方法将该线程挂起时

同步锁（Lock），jdk1.5之后

可重入锁ReentrantLock：线程可以对它已经加锁的对象进行再次加锁

死锁：当两个线程相互等待对方释放同步监视器时

线程通讯：

线程的协调运行：object类的 wait ，notify ，notifyAll（唤醒其他所有线程） 方法 配合Synchronized关键字

使用条件变量来控制协调：Condition 和await signal sgnalAll（唤醒该lock对象对应的其他线程）

管道流：

管道流实现多线程间的通信的步骤：

1.创建管道输出流和管道输入流

2.使用管道输出流或者输入流的connect方法将两个输入流和输出流链接起来

3.将输出流和输入流分别传入两个线程

4.两个线程可以依赖各自的输出流和输入流进行通信

线程组：TreadGroup

未处理异常 uncaughtException

Callable 和future 有返回值线程执行体 jdk1.5之后

实现步骤:

1.创建一个callable的实现类，并实现call方法，该call方法作为执行体，且该方法有返回值

2.创建callable的实例，并使用FutrueTask包装callable对象，该FutrueTask对象封装了该callable对象中国call方法的返回值

3.使用FutureTask对象作为新的线程的target对象，并启动线程

4.调用了FutureTask对象的方法获取子线程的返回值

线程池jdk1.5之后

Executors工厂的实例：

newCachedThreadPool 具有缓存功能的线程池，根据需要创建线程，这些线程都会缓存到线程池中

newFixedThreadPool（int nThreads） 具有可重用，固定线程数的线程池

newSingleThreadExecutor 创建一哥具有单线程的线程池，相当于newFixedThreadPool的线程池的个数是1

newScheduledThreadPool（int corePoolSize） 创建具有指定线程数的线程池，他可以是指延迟后执行线程任务

newSingleThreadScheduledExecutor创建具有指定一条线程的线程池，他可以是指延迟后执行

线程任务

ExecutorService 代表尽快执行的线程池 ，程序只需要将一哥runnable或者callable对象提交给该线程池即可 ；

submit(Runnable a)

submit(Runnable a ,T result)

submit(Callable<T> a)

ScheduledExecutorService 代表在指定延迟或者周期内执行线程任务的线程池

方法：scheduled()

execute和submit方法区别

主要有三个区别：

1、接收的参数不一样。

ExecutorService的submit(Callable<T> task)参数是一个Callable对象,submit(Runnable task)也可以接受Runnable对象；executorService.execute(Runnable command)的参数是一个Runnable对象。

2、submit有返回值，而execute没有。

Method submit extends base method Executor.execute by creating and returning a Future that can be used to cancel execution and/or wait for completion. 用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。然后我就可以把所有失败的原因综合起来发给调用者。个人觉得cancel execution这个用处不大，很少有需要去取消执行的，而最大的用处应该是第二点。

3、submit方便Exception处理。

There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit any thrown exception, checked or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will rethrow this exception, wrapped in an ExecutionException.意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

例如：我有很多更新各种数据的task，我们希望如果其中一个task失败，其它的task就不需要执行了。那我们就需要catch Future.get()抛出的异常，然后终止其它task的执行。代码如下：

ThreadLocaL 线程局部变量：为每一个使用该变量的线程提供一个变量值的副本，每一个线程都可以独立的改变自己的副本，而不会与其他线程冲突

包装线程不安全的集合：

使用Collections的静态方法