**08 Java多线程与并发**

一些基础知识点：

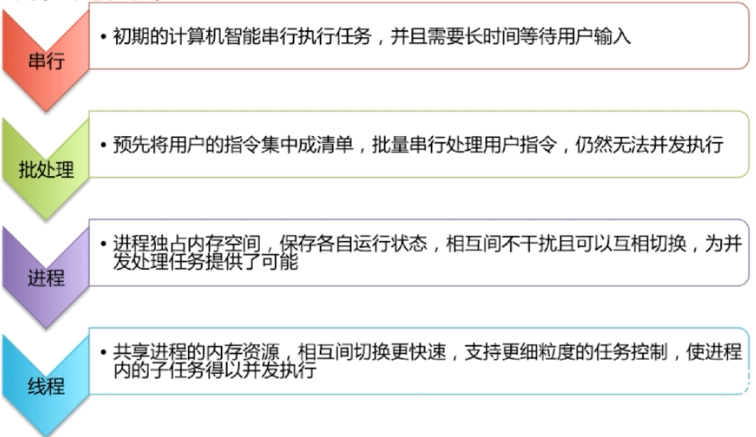
什么是进程？什么是线程？

计算机组成结构？

Linux的用户态和内核态定义？它们之间如何转换？为什么要转换？

什么是系统中断？

进程和线程的由来：

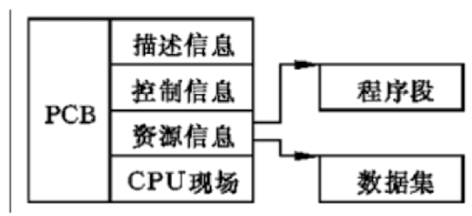


**进程对应一个程序**，程序是可执行文件，进程是执行中程序的实例；每个进程**对应一定的内存地址空间（资源）**，并且只能使用自己的内存空间，各个进程互不干扰，保存了程序每个运行时刻的状态，为进程切换提供了可能。**线程执行进程的子任务**，一个进程可以包含多个线程，每个线程负责一个独立的子任务，以提升实时性。

进程和线程的区别？

答：**进程是资源分配的最小单位，线程是CPU调度的最小单位**。

* 所有与进程有关的的资源，都被记录在PCB（进程控制块）中
* 进程是抢占处理机的调度单位；线程隶属于进程，共享其资源
* 线程只由堆栈寄存器、程序计数器和TCB（线程控制块 存储线程局部变量）组成

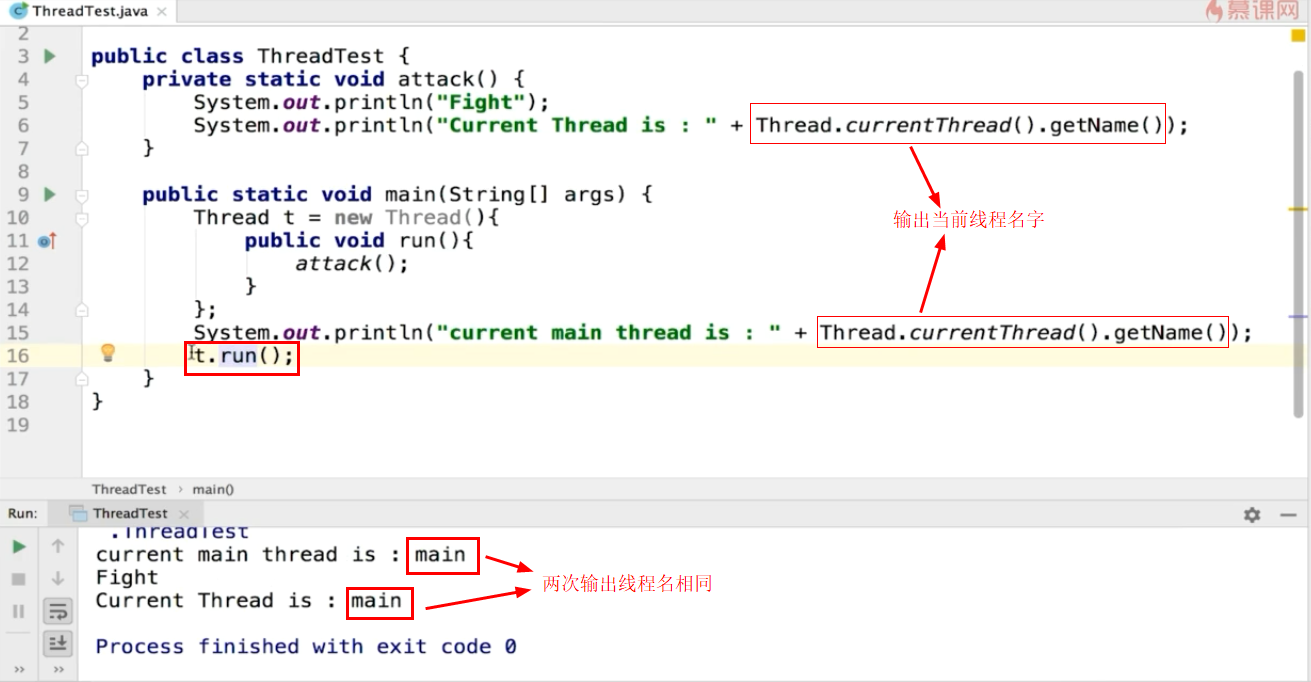
PCB PCB和TCB的关系

* （独立性）进程是独立应用，线程不是
* （内存等资源）进程有独立地址空间等资源，线程没有，它共享进程资源
* （健壮性）因为进程之间互不干扰，多进程比多线程健壮
* （切换开销）进程切换开销大于线程切换开销

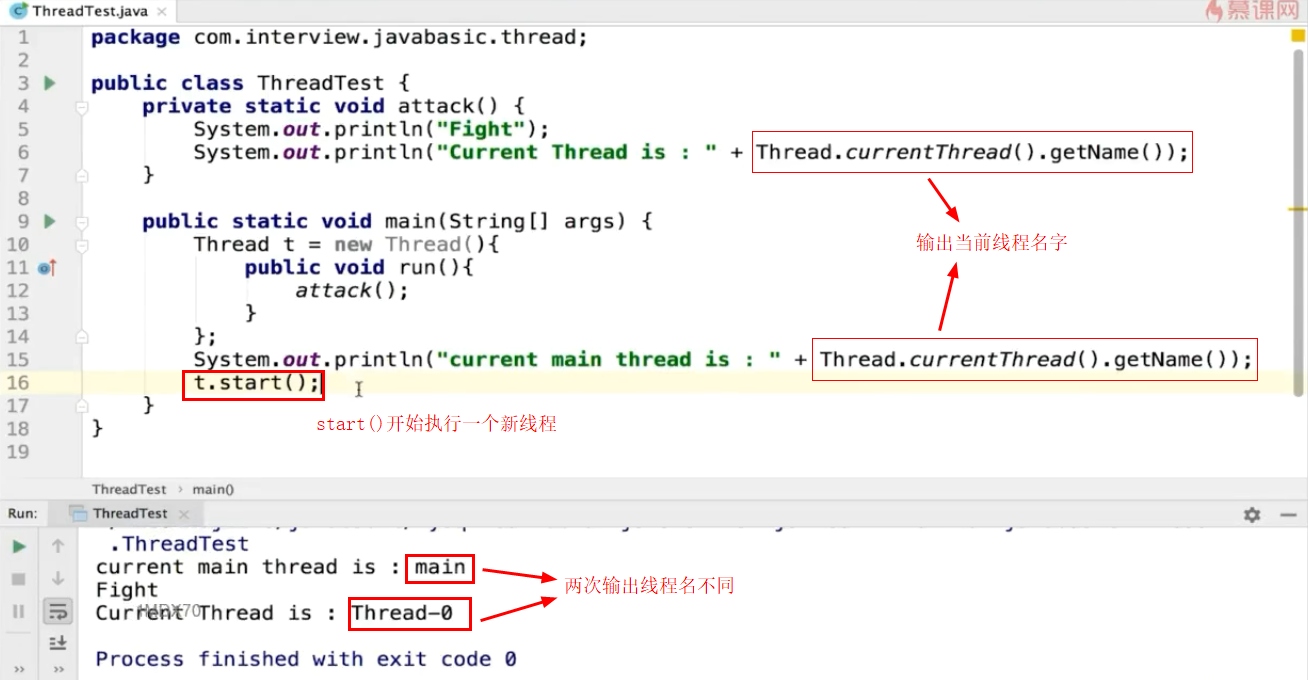
Java进程和线程的关系？

1. 由于跨平台性，Java对操作系统提供的功能进行封装，包括进程和线程
2. 运行一个Java程序会产生一个Java进程，每个进程至少包含一个线程（主线程）
3. 每个Java进程对应一个JVM实例，多个线程共享JVM的堆（堆是线程共享的）
4. Java程序会自动创建一个主线程
5. 主线程可以创建子线程，原则上后语所有子线程完成执行

Thread中start和run方法的区别？



调用 线程.run()方法 两次输出均为主线程



调用 线程.start()方法执行一个新线程 两次输出线程名不同

* 调用start()方法对创建一个新的子线程并启动
* 直接调用run()只是在同一线程中执行它，没有新的线程被创建

Thread和Runnable的关系？

答：Thread是一个类，可以创建一个线程；Runnable是一个接口，实现了它的类是一个可执行任务。可以通过Override重写Thread中的run()方法或向Thread传入Runnable变量将任务装入线程。

如何给run()方法传参？（run()方法没有参数表也没有返回值）

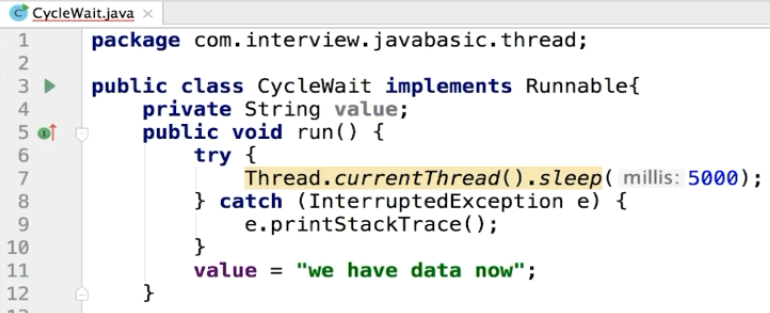
答：

1. 构造函数传参——向Thread或Runnable的构造函数传参
2. 成员变量传参——Thread或Runnable的成员变量
3. 回调函数传参

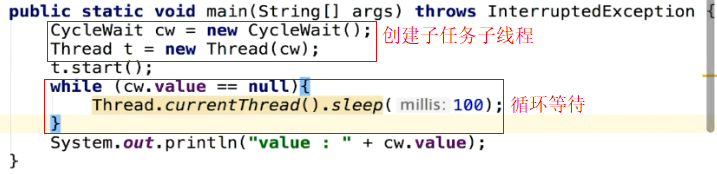
如何实现处理线程的返回值？

1. 主线程等待法（忙等待）——主线程循环等待，直到目标子线程返回值

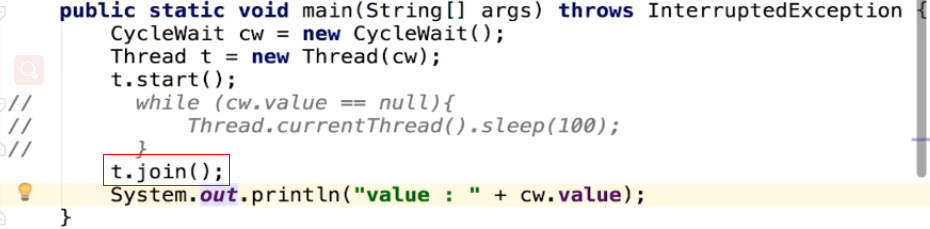
子任务：



循环等待的主线程：



1. 使用Thread类的join()阻塞当前线程以等待子线程处理完毕



**另一个使用join()方法的例子见《Java语言程序设计-进阶篇》P319-320**

1. 通过Callable接口实现：实现Callable接口的任务有返回值，执行Callable任务之后可获取Future类型的对象，在该对象向调用get()可获取Callable任务返回的Object；通过FutureTask 或线程池获取

**详解见“剑指Java 8-4 如何实现处理线程的返回值”**

线程的状态：

* 一般意义上的线程状态：1.新建 2.就绪 3.运行 4.阻塞（和等待）5.结束
* Thread.State描述的线程状态：

新建（New）：创建后尚未启动的线程

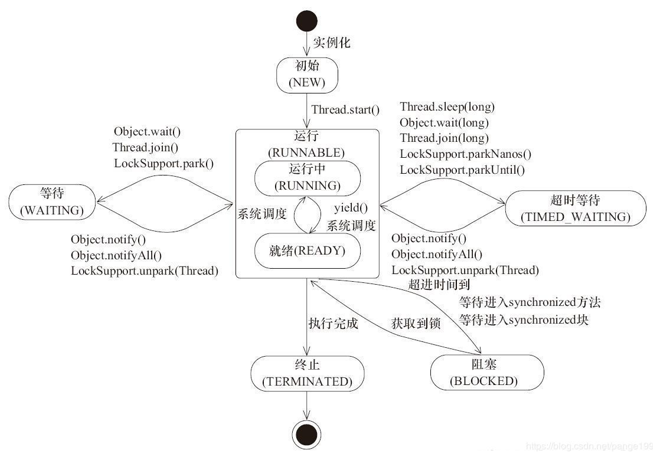
可运行（Runnable）：包含Running和Ready

无限等待（Waiting）：不会被分配CPU执行时间，需要显式的被唤醒

限期等待（Timed Waiting）：在一定时间后会由系统自动唤醒

阻塞（Blocked）：等待获取排它锁

结束（Terminated）：已终止线程的状态，线程已经结束执行（子线程run()或主线程main()完成时）



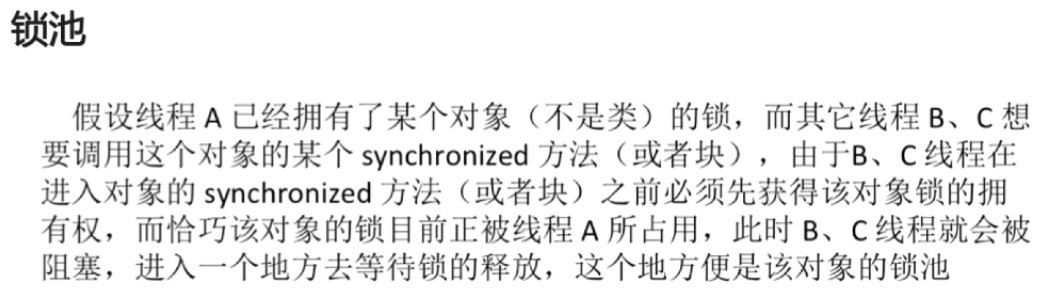
sleep和wait的区别？

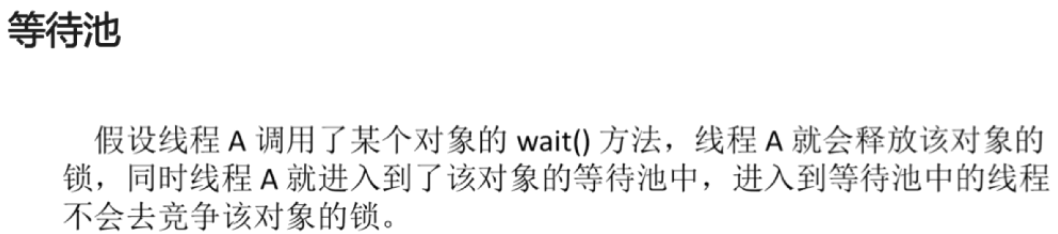
答：

1. sleep是Thread类的静态方法，通过Thread.sleep()调用；wait是Object类中定义的方法
2. sleep()方法可以在任何地方使用；wait()方法只能在synchronized方法或synchronized语句块中使用——因为作用是让线程释放锁并进入等待态，因此在加锁语句块中使用
3. **本质区别**：Thread.sleep只会让出CPU，不会导致锁行为改变；Object.wait不仅让出CPU，还会释放已经占有的同步资源锁，使等待其他同类资源的线程获得资源并运行

**本质区别验证代码见“剑指Java 8-6 sleep和wait的区别”**

两个概念：（1）锁池EntryList （2）等待池WaitSet





notify和notifyAll的区别？

答：

* notifyAll会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会
* notify只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会

**验证详解见“剑指Java 8-7 notify和notifyAll的区别”**

yield函数：是Thread类的静态方法，当调用Thread.yield()函数时，会给线程调度器一个当前线程愿意让出CPU的暗示，但线程调度器可以会停止该线程，也可能让该线程继续执行

**yield函数作用代码演示见“剑指Java 8-8 yield函数”**

**09 多线程与并发-原理**

线程安全问题的的主要诱因：

* 存在共享数据（临界资源）
* 存在多条线程共同操作这些共享数据
* 解决问题的根本方法：使同一时刻有却只有一个线程在操作共享数据

**synchronized关键字**获取两种锁——对象锁和类锁

* 获取对象锁的两种方法：

1. 同步非静态方法（synchronized method）,给调用该方法的对象加锁
2. 同步代码块（synchronized(this) 或 synchronized(对象名)）,给this或指定对象名加锁

* 获取类锁的两种方法：

1. 同步静态方法（synchronized static method）,给这个静态方法的类加锁
2. 同步静态代码块（synchronized (类.class)）,给这个指定类加锁——也就是给 类.class 这个对象加锁

Synchronized的一些锁优化技术：自适应自旋、锁消除、锁粗化、轻量锁、偏向锁…

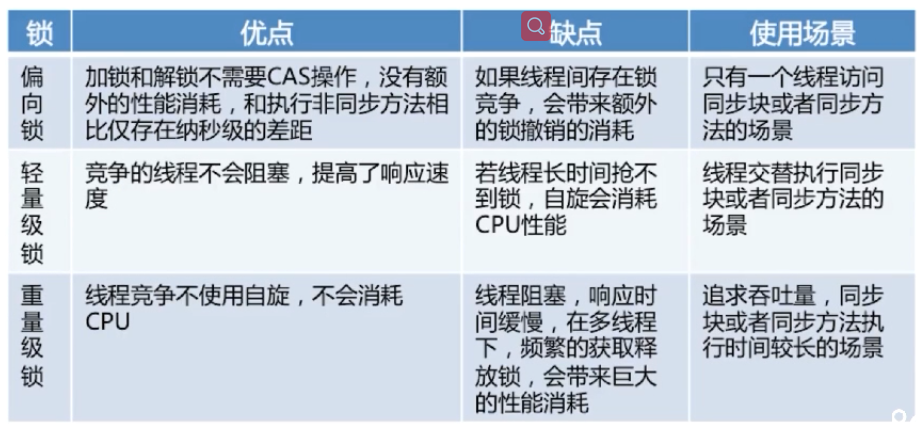
自旋锁：许多情况下，共享数据的锁定状态持续时间较短，为了这段时间挂起、恢复阻塞线程并不值得，可以让没有获取到锁的线程不放弃CPU时间的循环等待（忙等待）一会。但如果自旋等待时间较长，会增加CPU开销，降低性能

自适应自旋锁：自旋的次数不再固定，由上一次自旋的情况动态确定。使JVM的性能更高

锁消除：JIT运行时，对运行的上下文进行扫描，去除不可能存在的竞争的锁，节省无意义的申请锁时间

锁粗化：一串操作对同一个对象反复加锁解锁，甚至加锁操作是在循环体中，频繁锁操作会降低性能。因此扩大加锁范围，避免反复加锁解锁

锁膨胀方向：无锁（共享数据没有被任何线程占用） → 偏向锁（只有一个线程占用） → 轻量级锁（线程交替占用） → 重量级锁（多线程同时访问锁）

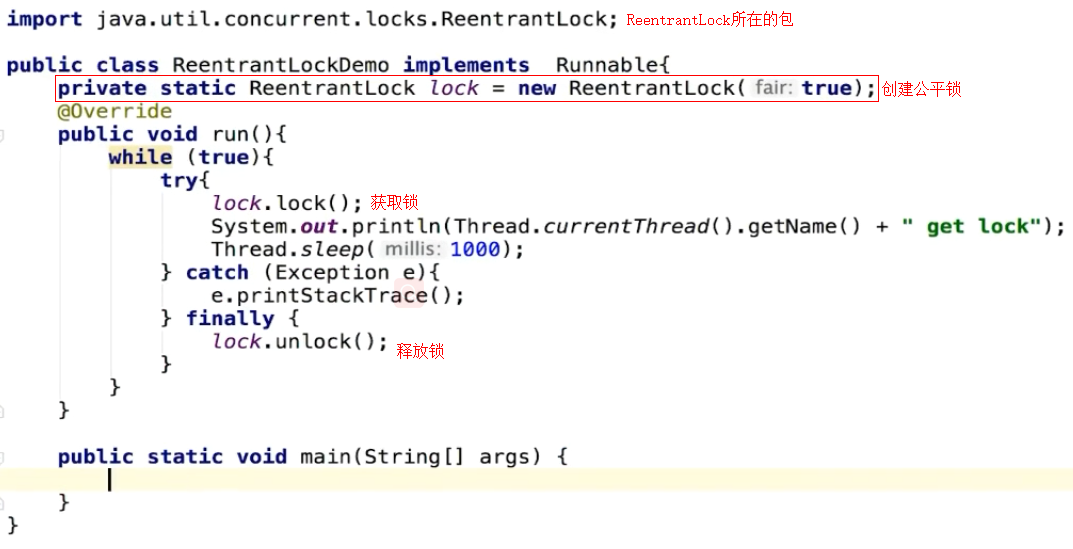


**ReentrantLock（再入锁）：**

1. 位于java.util.concurrent.locks包
2. 基于AQS实现
3. 能够实现比synchronized关键字更细粒度的控制，如控制fairness（公平策略）
4. 调用lock()之后，必须调用unlock()释放锁

ReentrantLock公平性设置：ReentrantLock fairLock=new ReentrantLock(true);

* 公平锁：参数为true时，采取公平策略，将锁赋给等待时间最久的的线程（慎用）
* 非公平锁：无参数或参数为false时，锁随机赋给一个线程
* synchronized是非公平锁



使用ReentrantLock代码示例——对lock()调用后紧随try-catch块并在finally中lock.unlock()释放锁是一个好习惯

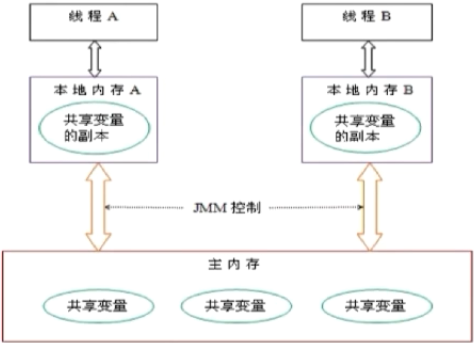
Synchronized关键字和ReentrantLock的区别：

答：

1. synchronized是关键字，ReentrantLock是实现了Lock接口的类
2. synchronized更简单，ReentrantLock显式锁更加直观灵活

**Java内存模型JMM(Java Memory Model):**

JMM本身是一种抽象概念，并不真实存在，描述了一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式



JMM中的主内存——**对应JDK8的JVM内存模型的堆和方法区**：

1. 存储Java实例对象，所有线程创建的对象实例都存放在主内存中
2. 包括成员变量、类信息、常量静态变量等
3. 由于是数据共享的区域（线程共享），多线程并发操作会引发线程安全问题

JMM中的工作内存——**对应JDK8的JVM内存模型的程序计数器PCR、栈和本地方法栈**：

1. 存储当前方法的所有本地变量信息，本地变量对其他线程不可见
2. 字节码行号指示器、Native方法信息
3. 属于线程私有区域，不存在线程安全问题

主内存与工作内存的数据存储类型及操作方式——对于一个实例对象：

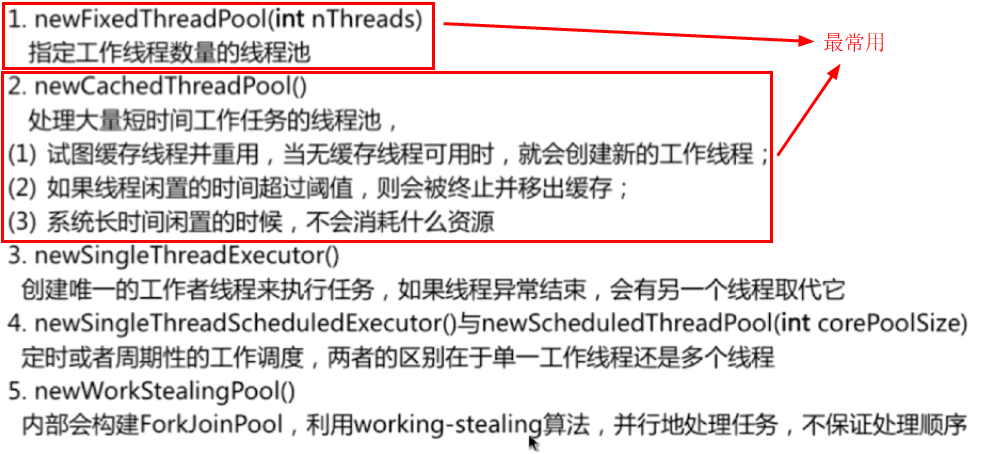
1. 它的成员方法中的基本数据类型（int、boolean、double、char等）本地变量存储在工作内存的栈帧结构中
2. 成员方法中引用类型的本地变量，引用存储在工作内存的栈帧中，实例存储在主内存（堆）中
3. 成员变量、static变量、类信息均存储在主内存中
4. 主内存共享的方式是线程各拷贝一份数据到工作内存，操作完成后刷新回主内存

**指令重排序、happens-before、volatile关键字、CAS等内容暂略**

**线程池的背景：**

服务器给接收到的每个请求创建一个线程。若并发的请求很多，但每个线程执行时间很短，就会**频繁地创建和销毁线程。会大大降低系统效率**，甚至服务器创建销毁线程消耗的时间和资源超过实际处理需要所需。因此引入**线程池重复利用线程完成新的任务**。

**Executors类可以创建线程池：**



**使用Executors创建线程池的简单方法见《Java语言程序设计-进阶篇》 P322-324**

**30.6线程池**

**Fork/Join框架：**

把大任务分割成若干个小任务并并行执行（Fork过程），最终汇总每个小任务结果后得到大任务结果（Join过程）

伪代码：

if(问题规模小)

顺序解决

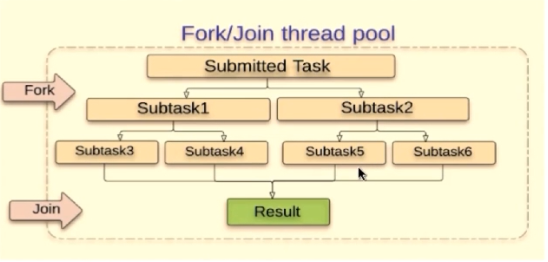
else{

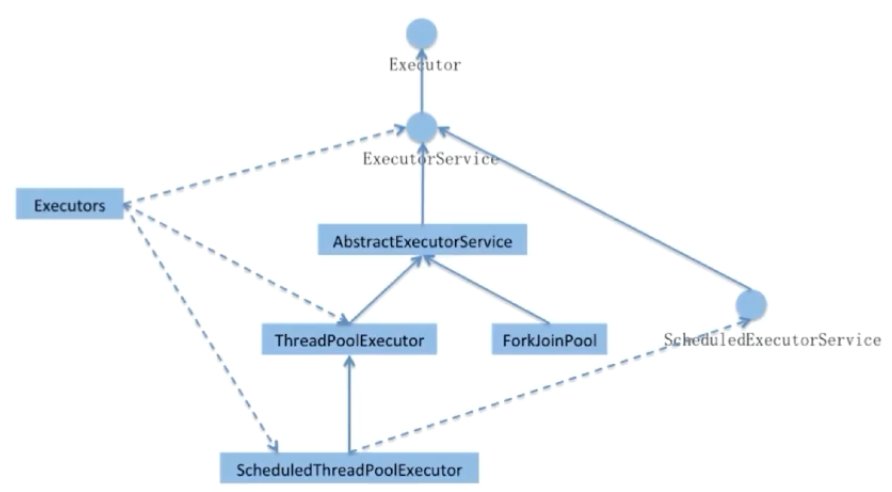
分解成不重叠的子问题（fork）

并发的解决子问题（递归）

合并子问题结果，得到最终解（join）

}



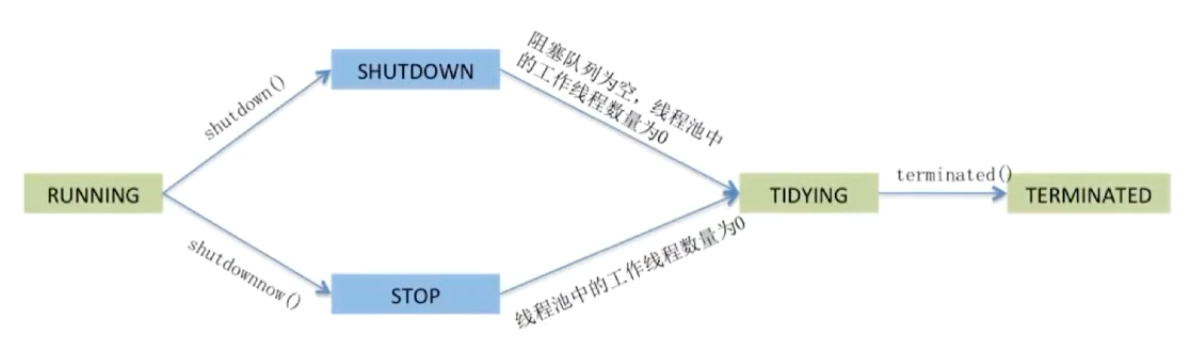


Executor接口的被继承、实现框架关系

**ThreadPoolExecutor类相关整理暂略**

**线程的状态：**

* RUNNING：能接受新提交的的任务，并且也能够处理阻塞队列中的任务
* SHUTDOWN：不再接受新提交的任务，但可以处理存量任务
* STOP：不再接受新提交的任务，也不处理存量任务
* TIDYING：所有任务都已终止
* TERMINATED：terminated()方法执行完后进入该状态



线程状态转换图

**线程池的大小如何选定？**

* 计算密集型任务——CPU算力是稀缺资源，避免线程过多引起切换操作过多造成CPU算力浪费：

线程数 = CPU核数 或 CPU核数+1

* I/O密集型任务——有较多等待操作：

线程数 = CPU核数\*（1+平均等待时间/平均工作时间）