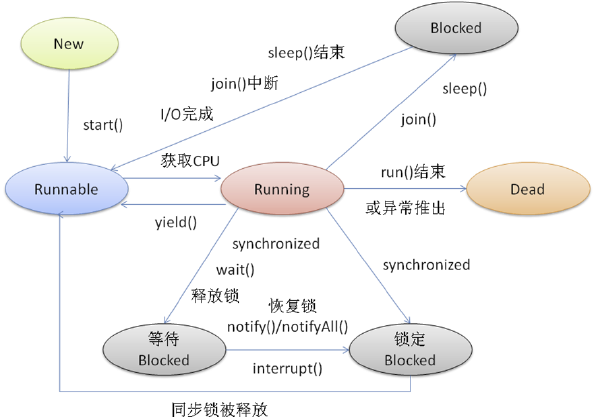
## 线程状态转换



线程在Running的过程中可能会遇到阻塞(Blocked)情况

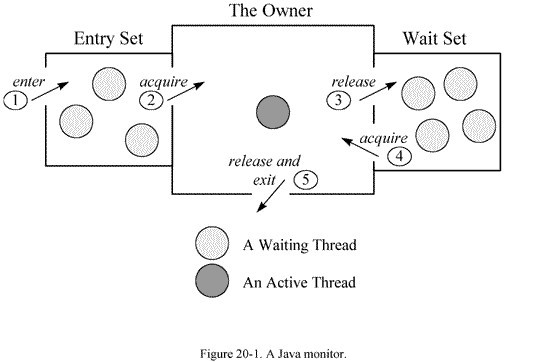
调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。

调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）

对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

## 1、线程同步

synchronized, wait, notify 是任何对象都具有的同步工具。让我们先来了解他们



**monitor:**

他们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时（如图中5的状态，即脱离同步块）去wait或notify，会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出此异常。

### 1、synchronized:

代码块：如下，在多线程环境下，synchronized块中的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容。

public class Thread1 implements Runnable {

Object lock;

public void run() {

synchronized(lock){

..do something

}

}

}

直接用于方法： 相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例。

public class Thread1 implements Runnable {

public synchronized void run() {

..do something

}

}

### 2、synchronized, wait, notify结合

典型场景生产者消费者问题

/\*\*

\* 生产者生产出来的产品交给店员

\*/

public synchronized void produce()

{

if(this.product >= MAX\_PRODUCT)

{

try

{

wait();

System.out.println("产品已满,请稍候再生产");

}

catch(InterruptedException e)

{

e.printStackTrace();

}

return;

}

this.product++;

System.out.println("生产者生产第" + this.product + "个产品.");

notifyAll(); //通知等待区的消费者可以取出产品了

}

/\*\*

\* 消费者从店员取产品

\*/

public synchronized void consume()

{

if(this.product <= MIN\_PRODUCT)

{

try

{

wait();

System.out.println("缺货,稍候再取");

}

catch (InterruptedException e)

{

e.printStackTrace();

}

return;

}

System.out.println("消费者取走了第" + this.product + "个产品.");

this.product--;

notifyAll(); //通知等待去的生产者可以生产产品了

}

### 3、volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。

针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

### 4、管道通信

管道通信就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

## 2、线程类

基本线程类指的是Thread类，Runnable接口，Callable接口

### Thread

**Thread 类实现了Runnable接口，启动一个线程的方法：**

MyThread my = new MyThread();

　　my.start();

**Thread类相关方法：**

//当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）

public static Thread.yield()

//暂停一段时间

public static Thread.sleep()

//在一个线程中调用other.join(),将等待other执行完后才继续本线程。

public join()

//后两个函数皆可以被打断

public interrupte()

**关于中断**：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。终端只会影响到wait状态、sleep状态和join状态。被打断的线程会抛出InterruptedException。  
Thread.interrupted()检查当前线程是否发生中断，返回boolean  
synchronized在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

**Thread类最佳实践**：  
写的时候最好要设置线程名称 Thread.name，并设置线程组 ThreadGroup，目的是方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

### Runnable

与Thread类似，实现Runnable接口必须重写其run方法，调用Thread.start()启动线程。Thread类是实现了Runnable接口的。

### Callable

实现Callable接口必须重写其call方法，调用Thread.start()启动线程。

Runnable和Callable的区别是：

(1)Callable规定的方法是call()，Runnable规定的方法是run()。其中Runnable可以提交给Thread来包装下，直接启动一个线程来执行，而Callable则一般都是提交给ExecuteService来执行。

(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得

(3)call方法可以抛出异常，run方法不可以

(4)运行Callable任务可以拿到一个Future对象，c表示异步计算的结果。

## 3、线程调度

线程的调度

1、调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX\_PRIORITY

线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

## 4、并发工具类concurrent

Executor                  ：具体Runnable任务的执行者。

ExecutorService           ：一个线程池管理者，其实现类有多种，我会介绍一部分。我们能把Runnable,Callable提交到池中让其调度。

Semaphore                 ：一个计数信号量

ReentrantLock             ：一个可重入的互斥锁定 Lock，功能类似synchronized，但要强大的多。

Future                    ：是与Runnable,Callable进行交互的接口，比如一个线程执行结束后取返回的结果等等，还提供了cancel终止线程。

**BlockingQueue**             ：

阻塞队列。

CompletionService         : ExecutorService的扩展，可以获得线程执行结果的

CountDownLatch            ：一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

CyclicBarrier             ：一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点

Future                    ：Future 表示异步计算的结果。

ScheduledExecutorService ：一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

**ConcurrentHashMap** :

ConcurrentHashMap 和 java.util.HashTable 类很相似，但 ConcurrentHashMap 能够提供比 HashTable 更好的并发性能。在你从中读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住。

此外，在你向其中写入对象的时候，ConcurrentHashMap 也不会锁住整个 Map。它的内部只是把 Map 中正在被写入的部分进行锁定。

另外一个不同点是，在被遍历的时候，即使是 ConcurrentHashMap 被改动，它也不会抛 ConcurrentModificationException。尽管 Iterator 的设计不是为多个线程的同时使用。更多关于 ConcurrentMap 和 ConcurrentHashMap 的细节请参考官方文档。

**CyclicBarrier** :

两个线程在栅栏旁等待对方。通过调用 CyclicBarrier 对象的 await() 方法，两个线程可以实现互相等待。一旦 N 个线程在等待 CyclicBarrier 达成，所有线程将被释放掉去继续运行。

**Semaphore** :

java.util.concurrent.Semaphore 类是一个计数信号量。这就意味着它具备两个主要方法：

acquire()

release()

计数信号量由一个指定数量的 “许可” 初始化。每调用一次 acquire()，一个许可会被调用线程取走。每调用一次 release()，一个许可会被返还给信号量。因此，在没有任何 release() 调用时，最多有 N 个线程能够通过 acquire() 方法，N 是该信号量初始化时的许可的指定数量。这些许可只是一个简单的计数器。这里没啥奇特的地方。

Semaphore 用法

信号量主要有两种用途：

保护一个重要(代码)部分防止一次超过 N 个线程进入。

在两个线程之间发送信号。

保护重要部分

如果你将信号量用于保护一个重要部分，试图进入这一部分的代码通常会首先尝试获得一个许可，然后才能进入重要部分(代码块)，执行完之后，再把许可释放掉。

比如这样：

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

**ThreadPoolExecutor** :

java.util.concurrent.ThreadPoolExecutor 是 ExecutorService 接口的一个实现。ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。

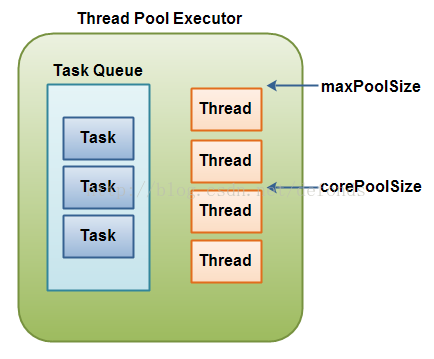
ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定：

corePoolSize

maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。

ThreadPoolExecutor 图解：



创建一个 ThreadPoolExecutor

ThreadPoolExecutor 有若干个可用构造子。比如：

int corePoolSize = 5; //核心线程数，默认情况下核心线程会一直存活

int maxPoolSize = 10; //线程池所能容纳的最大线程数

long keepAliveTime = 5000; //非核心线程的闲置超时时间，超过这个时间就会被回收

TimeUnit unit = TimeUnit.MILLISECOND; //指定keepAliveTime的单位

workQueue //线程池中的任务队列,常用的有三种队列:SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

合理的使用线程池能够带来3个很明显的好处：

1.降低资源消耗：通过重用已经创建的线程来降低线程创建和销毁的消耗

2.提高响应速度：任务到达时不需要等待线程创建就可以立即执行。

3.提高线程的可管理性：线程池可以统一管理、分配、调优和监控。

**Lock** :

既然 Lock 是一个接口，在你的程序里需要使用它的实现类之一来使用它。以下是一个简单示例：

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

首先创建了一个 Lock 对象。之后调用了它的 lock() 方法。这时候这个 lock 实例就被锁住啦。任何其他再过来调用 lock() 方法的线程将会被阻塞住，直到锁定 lock 实例的线程调用了 unlock() 方法。最后 unlock() 被调用了，lock 对象解锁了，其他线程可以对它进行锁定了。

Java Lock 实现

java.util.concurrent.locks 包提供了以下对 Lock 接口的实现类：

ReentrantLock

Lock 和 synchronized 代码块的主要不同点

一个 Lock 对象和一个 synchronized 代码块之间的主要不同点是：

synchronized 代码块不能够保证进入访问等待的线程的先后顺序。

你不能够传递任何参数给一个 synchronized 代码块的入口。因此，对于 synchronized 代码块的访问等待设置超时时间是不可能的事情。

synchronized 块必须被完整地包含在单个方法里。而一个 Lock 对象可以把它的 lock() 和 unlock() 方法的调用放在不同的方法里。

Lock 的方法

Lock 接口具有以下主要方法：

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

lock() 将 Lock 实例锁定。如果该 Lock 实例已被锁定，调用 lock() 方法的线程将会阻塞，直到 Lock 实例解锁。

lockInterruptibly() 方法将会被调用线程锁定，除非该线程被打断。此外，如果一个线程在通过这个方法来锁定 Lock 对象时进入阻塞等待，而它被打断了的话，该线程将会退出这个方法调用。

tryLock() 方法试图立即锁定 Lock 实例。如果锁定成功，它将返回 true，如果 Lock 实例已被锁定该方法返回 false。这一方法永不阻塞。tryLock(long timeout, TimeUnit timeUnit) 的工作类似于 tryLock() 方法，除了它在放弃锁定 Lock 之前等待一个给定的超时时间之外。

unlock() 方法对 Lock 实例解锁。一个 Lock 实现将只允许锁定了该对象的线程来调用此方法。其他(没有锁定该 Lock 对象的线程)线程对 unlock() 方法的调用将会抛一个未检查异常(RuntimeException)。

**ReadWriteLock** :

java.util.concurrent.locks.ReadWriteLock 读写锁是一种先进的线程锁机制。它能够允许多个线程在同一时间对某特定资源进行读取，但同一时间内只能有一个线程对其进行写入。

读写锁的理念在于多个线程能够对一个共享资源进行读取，而不会导致并发问题。并发问题的发生场景在于对一个共享资源的读和写操作的同时进行，或者多个写操作并发进行。

**读锁**：如果没有任何写操作线程锁定 ReadWriteLock，并且没有任何写操作线程要求一个写锁(但还没有获得该锁)。因此，可以有多个读操作线程对该锁进行锁定。

**写锁**：如果没有任何读操作或者写操作。因此，在写操作的时候，只能有一个线程对该锁进行锁定。

java.util.concurrent.locks 包提供了 ReadWriteLock 接口的以下实现类：

ReentrantReadWriteLock。

ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

// multiple readers can enter this section

// if not locked for writing, and not writers waiting

// to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

// only one writer can enter this section,

// and only if no threads are currently reading.

readWriteLock.writeLock().unlock();

注意如何使用 ReadWriteLock 对两种锁实例的持有。一个对读访问进行保护，一个队写访问进行保护。

**AtomicInteger** :

AtomicInteger 类为我们提供了一个可以进行原子性读和写操作的 int 变量，它还包含一系列先进的原子性操作，比如 compareAndSet()。AtomicInteger 类位于 java.util.concurrent.atomic 包，因此其完整类名为 java.util.concurrent.atomic.AtomicInteger。

AtomicInteger 类包含有一些方法，通过它们你可以增加 AtomicInteger 的值，并获取其值。这些方法如下：

addAndGet()

getAndAdd()

getAndIncrement()

incrementAndGet()