java.lang.Thread 1.0

static void sleep(long millis)

休眠给定的毫秒数

Thread(Runnable target)

构造一个新线程，用于调用给定target的run()方法

void start()

启动这个线程，将引发调用run()方法，这个方法将立即返回，并且新线程将并行运行。

void run()

调用关联Runnable的run方法

java.lang.Runnable 1.0

void run()

必须覆盖这个方法，并在这个方法中提供所要执行的任务指令

中断线程

java.lang.Thread 1.0

void interrupt()

向线程发送中断请求。线程的中断状态将被设置为true。如果目前该线程被一个sleep调用阻塞，那么interruptedException异常被抛出。

static boolean interrupted()

测试当前线程（即正在执行这一命令的线程）是否被中断。注意，这是个静态方法。这一调用会产生副作用——它将当前线程中的中断状态重置为false

boolean isInterrupted()

测试线程是否被终止。不像静态的中断方法，这一调用不改变线程的中断状态。

static Thread currentThread()

返回代表当前执行线程的Thread对象。

线程状态

New 新创建

Runnable 可运行

Blocked 被阻塞

Waiting 等待

Timed Waiting 计时等待

Terminated 被终止



java.lang.Thread 1.0

void join()

等待终止指定的线程

void join(long millis)

等待指定的线程死亡或者经过指定的毫秒数

Thread.State getState() 5.0

得到这一线程的状态

void stop()

停止该线程。Deprecated

void suspend()

暂停这一线程的执行 Deprecated

void resume()

恢复线程，仅仅在调用suspend后使用。 Deprecated

线程属性

线程优先级

java.lang.Thread 1.0

void setPriority(int newPriority)

设置线程的优先级。优先级必须在Thread.MIN\_PRIORITY(在Thread中定义为1)与Thread.MAX.\_PRIORITY(在Thread中定义为10)之间。一般使用Thread.NORM\_PRIORITY优先级(在Thread中定义为5)

static void yield()

导致当前执行线程处于让步状态。如果其他的可运行线程具有至少与此线程同样高的优先级，那么这些线程接下来会被调度。

守护线程

void setDaemon(boolean isDaemon)

标识该线程为守护线程或用户线程。这一方法必须在线程启动之前调用

未捕获异常处理器

线程的run方法不能抛出任何被检测的异常，但是，不被检测的异常会导致线程终止。这样线程就死亡了。

Thread.UncaughtExceptionHandler

void uncaughtException(Thread t,Throwable e)

当一个线程因未捕获异常而终止，按规定要将客户报告记录到日志中。

参数：t 由于未捕获异常而终止的线程

e 未捕获的异常对象

同步

锁对象 LOCK

java.util.concurrent.locks.Lock 5.0

void lock()

获取这个锁，如果锁同时被另一个线程拥有则发生阻塞

void unlock()

释放这个锁

java.util.concurrent.locks.ReentrantLock 5.0

ReentrantLock()

构建一个可以被用来保护临界区的可重入锁

ReentrantLock(boolean fair)

构建一个带有公平策略的锁。一个公平锁偏爱等待时间最长的线程。但是，这一公平的保证将大大降低性能。所以，默认情况下，锁没有被强制为公平的。

条件对象 Condition

java.until.concurrent.locks.Lock 5.0

Condition newCondition()

返回一个与该锁相关的条件对象。

java.util.concurrent.locks.Condition 5.0

void await()

将该线程放到条件的等候集中

void signalAll()

解除该条件的等待集中的所有线程的阻塞状态

void signal()

从该条件的等候集中随机地选择一个线程，解除其阻塞状态

synchronized关键字

锁和条件的关键之处：

锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码。

锁可以管理试图进入被保护代码段的线程。

锁可以拥有一个或多个相关的条件对象。

每个条件对象管理那些已经进入被保护的代码段但还不能运行的线程。

Lock 和 Condition接口为程序设计人员提供了高度的锁定控制。

内部锁，如果一个方法用synchronized关键字声明，那么对象的锁将保护整个方法。也就是说，要调用该方法，线程必须获得内部的对象锁。

public synchronized void method(){

method body

}

public void method()

{

this.intrinsicLock.lock();

try{

method body

}

finally{

this.intrinsicLock.unlock();

}

}

内部对象锁只有一个相关条件，wait方法添加一个线程到等待集中，notifyAll/notify方法解除等待线程的阻塞状态。换句话说，调用wait或notifyAll等价于

intrinsicCondition.await();

intrinsicCondition.signalAll();

PS: wait、notifyAll以及notify方法时Object类的final方法。Condition方法必须被命名为await、signalAll和signal以便它们不会与那些方法发生冲突。

内部锁和条件存在一些局限。包括：

不能中断一个正在试图获得锁的线程

试图获得锁时不能设定超时

每个所仅有单一的条件，可能是不够的。

最好既不使用Lock/Condition也不使用synchronized关键字。可以使用java.util.concurrent包中的机制，它会为你处理所有的加锁。

如果synchronized关键字适合你的程序，那么请尽量使用它，这样可以减少编写的代码数量，减少出错的几率。

如果特别需要Lock/Condition结构提供的独有特性时，才使用Lock/Condition

java.lang.Object 1.0

void notifyAll()

解除那些在该对象上调用wait方法的线程的阻塞状态。该方法只能在同步方法或同步块内部调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常

void notify()

随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态。该方法只能在一个同步方法或同步块中调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常

void wait()

导致线程进入等待状态知道它被通知。该方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常

void wait(long millis)

void wait(long millis,int nanos)

导致线程进入等待状态直到它被通知或者经过指定的时间。这些方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者该方法抛出一个IllegalMonitorStateException异常

millis 毫秒数

nanos 纳秒数 ，<1000000

同步阻塞

每一个java对象有一个锁。线程可以通过调用同步方法获得锁。还有另一种机制可以获得锁，通过进入一个同步阻塞。

监视器概念

监视器是只包含私有域的类

每个监视器类的对象有一个相关的锁

使用该锁对所有的方法进行加锁

该锁可以有任意多个相关条件

Volatile 域

volatile关键字为实例域的同步访问提供了一种免锁机制。

如果声明一个域为volatile，那么编译器和虚拟机就知道该域是可能被另一个线程并发更新的。

volatile变量不能提供原子性

final变量

安全的访问一个共享域，这个域声明为final时也可以

原子性

死锁

线程局部变量

锁测试和超时

读写锁

为什么弃用stop和suspend方法

stop方法用来终止一个线程

当线程被终止，立即释放被它锁住的所有对象的锁，这回导致对象处于不一致的状态

suspend方法用来阻塞一个线程知道另一个线程调用resume

会导致死锁

阻塞队列