# TODO

线程问题如何调试

# 进程与线程

同步机制应该遵循的基本准则

· 空闲让进：当无进程处于临界区时，表明临界资源处于空闲状态，允许一个请求进入临界区的进程立即进入临界区，以有效利用临界资源

 · 忙则等待：当已有进程处于临界区时，表明临界资源正在被访问，因而其他试图进入临界区的进程必须等待，以保证对临界资源的互斥访问

 · 有限等待：对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，以免陷入“死等”状态

 · 让权等待：当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态

线程和进程资源比较

|  |  |
| --- | --- |
| 进程占有的资源 | 每个线程内私有资源 |
| 地址空间  全局变量  打开的文件  子进程  信号量  账户信息 | 栈  寄存器  状态  程序计数器 |

线程共享的内容包括：

1.代码段（code segment）

2.数据段（data section）

3.进程打开的文件描述符、

4.信号的处理器、

5.进程的当前目录和

6.进程用户ID与进程组ID

(1)地址空间:进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;

(2)资源拥有:进程是资源分配和拥有的基本单元,同一进程内的线程共享进程的资源，线程几乎不拥有资源，只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈)

(3)CPU执行：线程是CPU处理器调度的基本单元（就绪，执行，阻塞）,但进程不是.

（4）从属关系：线程依赖进程而存在，进程至少有一个线程

实例：服务器的响应，显然为每一个用户的请求一个进程没有必要，采取线程就好

# 线程的状态

先来了解一下线程有哪些状态，这个将会有助于后面对Thread类中的方法的理解。

　　线程从创建到最终的消亡，要经历若干个状态。一般来说，线程包括以下这几个状态：创建(new)、就绪(runnable)、运行(running)、阻塞(blocked)、time waiting、waiting、消亡（dead）。

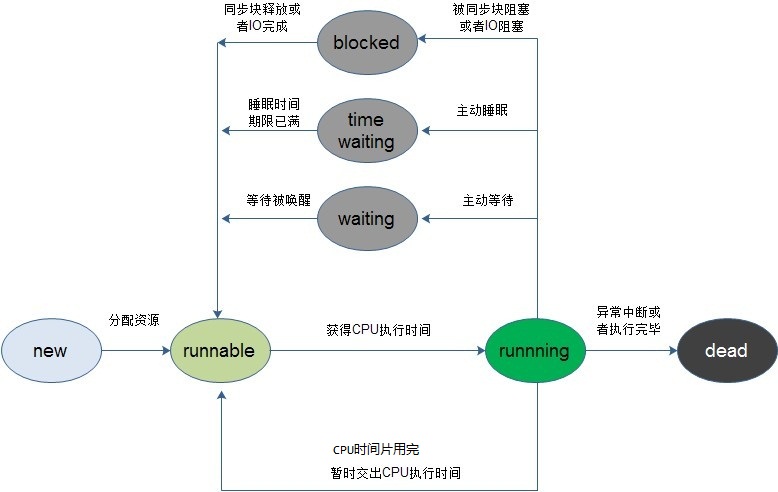
　　当需要新起一个线程来执行某个子任务时，就创建了一个线程。但是线程创建之后，不会立即进入就绪状态，因为线程的运行需要一些条件（比如内存资源，在前面的JVM内存区域划分一篇博文中知道程序计数器、Java栈、本地方法栈都是线程私有的，所以需要为线程分配一定的内存空间），只有线程运行需要的所有条件满足了，才进入就绪状态。

　　当线程进入就绪状态后，不代表立刻就能获取CPU执行时间，也许此时CPU正在执行其他的事情，因此它要等待。当得到CPU执行时间之后，线程便真正进入运行状态。

　　线程在运行状态过程中，可能有多个原因导致当前线程不继续运行下去，比如用户主动让线程睡眠（睡眠一定的时间之后再重新执行）、用户主动让线程等待，或者被同步块给阻塞，此时就对应着多个状态：time waiting（睡眠或等待一定的事件）、waiting（等待被唤醒）、blocked（阻塞）。

　　当由于突然中断或者子任务执行完毕，线程就会被消亡。

　　下面这副图描述了线程从创建到消亡之间的状态：



在有些教程上将blocked、waiting、time waiting统称为阻塞状态，这个也是可以的，只不过这里我想将线程的状态和Java中的方法调用联系起来，所以将waiting和time waiting两个状态分离出来。

# 上下文切换

对于单核CPU来说（对于多核CPU，此处就理解为一个核），CPU在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

由于可能当前线程的任务并没有执行完毕，所以在切换时需要保存线程的运行状态，以便下次重新切换回来时能够继续切换之前的状态运行。举个简单的例子：比如一个线程A正在读取一个文件的内容，正读到文件的一半，此时需要暂停线程A，转去执行线程B，当再次切换回来执行线程A的时候，我们不希望线程A又从文件的开头来读取。

　　因此需要记录线程A的运行状态，那么会记录哪些数据呢？因为下次恢复时需要知道在这之前当前线程已经执行到哪条指令了，所以需要记录程序计数器的值，另外比如说线程正在进行某个计算的时候被挂起了，那么下次继续执行的时候需要知道之前挂起时变量的值时多少，因此需要记录CPU寄存器的状态。所以一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

　　说简单点的：对于线程的上下文切换实际上就是 存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。

虽然多线程可以使得任务执行的效率得到提升，但是由于在线程切换时同样会带来一定的开销代价，并且多个线程会导致系统资源占用的增加，所以在进行多线程编程时要注意这些因素。

# 源码分析

package myjava**.**lang**;**

**import** java**.**security**.**AccessControlContext**;**

/\*\*

 \* 1.每个线程都有一个优先级，高优先级线程的执行优先于低优先级线程。

 \* 2.每个线程都可以或不可以标记为一个守护程序。每个线程都有一个标识名，多个线程可以同名。 如果线程创建时没有指定标识名，就会为其生成一个新名称。

 \* 3.当某个线程中运行的代码创建一个新 Thread 对象时，该新线程的初始优先级被设定为创建线程的优先级，

 \* 并且当且仅当创建线程是守护线程时，新线程才是守护程序。 当 Java 虚拟机启动时，通常都会有单个非守护线程（它通常会调用某个指定类的 main 方法）。

 \* 4.Java 虚拟机会继续执行线程，直到下列任一情况出现时为止： 1）调用了 Runtime 类的 exit 方法，并且安全管理器允许退出操作发生。

 \* 2）非守护线程的所有线程都已停止运行，无论是通过从对 run 方法的调用中返回，还是通过抛出一个传播到 run 方法之外的异常。

 \* 5.创建新执行线程有两种方法。 1）将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。 接下来可以分配并启动该子类的实例。

 \* 2）实现 Runnable 接口的类。 该类然后实现 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。

 \*/

public class Thread **implements** Runnable **{**

    /\*

     \* 这个方法及静态代码块在Object类中也存在，可以说是重写了Object类的registerNatives代码

     \*/

    private static native void registerNatives**();**

    static **{**

        registerNatives**();** // 静态代码块，确保registerNatives()方法在类加载的时候就执行，而该方法是native的

**}**

    private char name**[];** // 存线程名字。char[]的效率比String高

    private int priority**;** // 优先级

    private Thread threadQ**;**

    private long eetop**;**

    /\* Whether or not to single\_step this thread. \*/

    private boolean single\_step**;**

    /\* 是否是一个daemon守护线程 \*/

    private boolean daemon **=** **false;**

    /\* JVM state JVM状态 \*/

    private boolean stillborn **=** **false;**

    /\* 哪个线程会被执行，调用run方法时run方法是哪个类里面的 \*/

    private Runnable target**;**

    /\* 线程组 \*/

    private ThreadGroup group**;**

    /\* 该线程的上下文类加载器 \*/

    private ClassLoader contextClassLoader**;**

    /\* 该线程所继承的AccessControlContext环境 \*/

    private AccessControlContext inheritedAccessControlContext**;**

    /\* 为匿名线程命名的自动计数器 \*/

    private static int threadInitNumber**;** // 线程的编号，没有赋初值，所以从0开始。

    private static synchronized int nextThreadNum**()** **{** // 获得下一个线程的编号，使用了synchronized关键字来同步该方法，防止并发

**return** threadInitNumber**++;**

**}**

    /\*

     \* ThreadLocal values pertaining to this thread. This map is maintained by

     \* the ThreadLocal class.

     \*/

    ThreadLocal**.**ThreadLocalMap threadLocals **=** **null;**

    /\*

     \* InheritableThreadLocal values pertaining to this thread. This map is

     \* maintained by the InheritableThreadLocal class.

     \*/

    ThreadLocal**.**ThreadLocalMap inheritableThreadLocals **=** **null;**

    /\*

     \* The requested stack size for this thread, or 0 if the creator did not

     \* specify a stack size. It is up to the VM to do whatever it likes with

     \* this number; some VMs will ignore it.

     \*/

    private long stackSize**;**

    /\*

     \* JVM-private state that persists after native thread termination.

     \*/

    private long nativeParkEventPointer**;**

    /\*

     \* Thread ID

     \*/

    private long tid**;**

    /\* For generating thread ID \*/

    private static long threadSeqNumber**;**

    /\*

     \* Java thread status for tools, initialized to indicate thread 'not yet

     \* started'

     \*/

    private int threadStatus **=** 0**;** // 线程状态变量，默认为0

    private static synchronized long nextThreadID**()** **{**

**return** **++**threadSeqNumber**;**

**}**

    /\*\*

     \* The argument supplied to the current call to

     \* java.util.concurrent.locks.LockSupport.park. Set by (private)

     \* java.util.concurrent.locks.LockSupport.setBlocker Accessed using

     \* java.util.concurrent.locks.LockSupport.getBlocker

     \*/

    volatile Object parkBlocker**;**

    /\*

     \* The object in which this thread is blocked in an interruptible I/O

     \* operation, if any. The blocker's interrupt method should be invoked after

     \* setting this thread's interrupt status.

     \*/

    private volatile Interruptible blocker**;**

    private Object blockerLock **=** **new** Object**();**

    /\*

     \* Set the blocker field; invoked via sun.misc.SharedSecrets from java.nio

     \* code

     \*/

    void blockedOn**(**Interruptible b**)** **{**

        synchronized **(**blockerLock**)** **{**

            blocker **=** b**;**

**}**

**}**

    /\*\*

     \* 线程可以具有的最低优先级。

     \*/

    public final static int MIN\_PRIORITY **=** 1**;**

    /\*\*

     \* 分配给线程的默认优先级。

     \*/

    public final static int NORM\_PRIORITY **=** 5**;**

    /\*\*

     \* 线程可以具有的最高优先级。

     \*/

    public final static int MAX\_PRIORITY **=** 10**;**

    /\* If stop was called before start \*/

    private boolean stopBeforeStart**;** // 线程自己停止，还是用stop()停止线程的。默认值为false;

    /\* Remembered Throwable from stop before start \*/

    private Throwable throwableFromStop**;**

    /\*\* 返回对当前正在执行的线程对象的引用。 \*/

    public static native Thread currentThread**();**

    /\*\*\* 暂停当前正在执行的线程对象，并执行其他线程。 \*/

    public static native void yield**();**

    /\*\*

     \* 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

     \* 该线程不丢失任何监视器的所属权,不释放对象。

     \*

     \* 参数： millis - 以毫秒为单位的休眠时间。 抛出： InterruptedException -

     \* 如果任何线程中断了当前线程。当抛出该异常时，当前线程的中断状态 被清除。 另请参见： Object.notify()

     \*/

    public static native void sleep**(**long millis**)** **throws** InterruptedException**;**

    /\*\*

     \* 在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

     \* 该线程不丢失任何监视器的所属权。

     \*

     \* 参数： millis - 以毫秒为单位的休眠时间。 nanos - 要休眠的另外 0-999999 纳秒。 抛出：

     \* IllegalArgumentException - 如果 millis 值为负或 nanos 值不在 0-999999 范围内。

     \* InterruptedException - 如果任何线程中断了当前线程。当抛出该异常时，当前线程的中断状态 被清除。 另请参见：

     \*/

    public static void sleep**(**long millis**,** int nanos**)**

**throws** InterruptedException **{**

**if** **(**millis **<** 0**)** **{** // 如果传入的毫秒时间小于0，则抛非法参数异常

**throw** **new** IllegalArgumentException**(**"timeout value is negative"**);**

**}**

**if** **(**nanos **<** 0 **||** nanos **>** 999999**)** **{** // 如果传入的纳秒时间小于0或者大于999999，则抛非法参数异常

**throw** **new** IllegalArgumentException**(**

                    "nanosecond timeout value out of range"**);**

**}**

        // 就是纳秒大于500000 ，即大于0.5毫秒的时候四舍五入。或者毫秒为0、纳秒不为0时，睡一微秒。

**if** **(**nanos **>=** 500000 **||** **(**nanos **!=** 0 **&&** millis **==** 0**))** **{**

            millis**++;** // 如果纳秒大于等于500000，或者纳秒不等于0且毫秒等于0，则毫秒自增1。即对纳秒四舍五入到毫秒。

**}**

        sleep**(**millis**);**

**}**

    /\*\*

     \* Thread类中的init方法是Thread类中所有构造方法都调用的方法，用于初始化线程的各种信息 初始化一个线程

     \*

     \* @param g

     \*            the Thread group 线程组

     \* @param target

     \*            the object whose run() method gets called 调用谁的run()方法

     \* @param name

     \*            the name of the new Thread 线程名称

     \* @param stackSize

     \*            栈大小 the desired stack size for the new thread, or zero to

     \*            indicate that this parameter is to be ignored.

     \*/

    // 即是把线程类的引用保存到target中。这样，当调用Thread的run方法时，target就不为空了，而是继续调用了target的run方法，

    // 所以我们需要实现Runnable的run方法。这样通过Thread的run方法就调用到了Runnable实现类中的run方法。

    // 这也是Runnable接口实现的线程类需要这样启动的原因。

    private void init**(**ThreadGroup g**,** Runnable target**,** String name**,**

            long stackSize**)** **{**

        Thread parent **=** currentThread**();** // 当先线程

        SecurityManager security **=** System**.**getSecurityManager**();** // 安全管理，查看线程拥有的功能（例如：读写文件，访问网络）。

**if** **(**g **==** **null)** **{** // 判断线程组是否为空

            /\* Determine if it's an applet or not 确定是否为applet \*/

            /\*

             \* If there is a security manager, ask the security manager what to

             \* do. 如果有一个安全管理器，那么询问安全管理器该做什么

             \*/

**if** **(**security **!=** **null)** **{**

                g **=** security**.**getThreadGroup**();** // 如果安全管理器不为空，那么就可以得到线程组对象

**}**

            /\*

             \* If the security doesn't have a strong opinion of the matter use

             \* the parent thread group. 如果安全管理器没有强大的支持，那么就用调用当前线程的线程组

             \*/

**if** **(**g **==** **null)** **{** // 如果线程组还是为空，那么调用currentThread()的getThreadGroup()

                g **=** parent**.**getThreadGroup**();**

**}**

**}**

        /\*

         \* checkAccess regardless of whether or not threadgroup is explicitly

         \* passed in.

         \*/

        g**.**checkAccess**();** // 确定当前运行的线程是否有权修改此线程组。

        /\*

         \* Do we have the required permissions?

         \*/

**if** **(**security **!=** **null)** **{**

**if** **(**isCCLOverridden**(**getClass**()))** **{**// 权限校验，参数实际上是：new

                                                // RuntimePermission("enableContextClassLoaderOverride");

                security**.**checkPermission**(**SUBCLASS\_IMPLEMENTATION\_PERMISSION**);**

**}**

**}**

        g**.**addUnstarted**();** // 记录一个线程到线程组里，好像只是取得一个线程号。其实就是一个计数器自增：

                            // nUnstartedThreads++;

**this.**group **=** g**;** // 初始化确定线程组

**this.**daemon **=** parent**.**isDaemon**();** // 初始化确定线程是否为守护线程

**this.**priority **=** parent**.**getPriority**();** // 初始化确定线程的优先级

**this.**name **=** name**.**toCharArray**();** // 初始化确定线程名字

**if** **(**security **==** **null** **||** isCCLOverridden**(**parent**.**getClass**()))** // 初始化确定上下文的类加载器

**this.**contextClassLoader **=** parent**.**getContextClassLoader**();**

**else**

**this.**contextClassLoader **=** parent**.**contextClassLoader**;**

**this.**inheritedAccessControlContext **=** AccessController**.**getContext**();** // 权限快照？

**this.**target **=** target**;** // 初始化确定目标Runnable对象，即具体执行哪个Runnable对象里的run方法的对象

        setPriority**(**priority**);** // 初始化设置线程优先级

**if** **(**parent**.**inheritableThreadLocals **!=** **null)** // 本地线程？操作系统提供的相关线程？

**this.**inheritableThreadLocals **=** ThreadLocal

**.**createInheritedMap**(**parent**.**inheritableThreadLocals**);** // 创建本地线程

        /\* Stash the specified stack size in case the VM cares 存放特定的栈大小以防虚拟机不测 \*/

**this.**stackSize **=** stackSize**;** // 初始化栈大小

        /\* Set thread ID \*/

        tid **=** nextThreadID**();** // 初始化设置线程的ID

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(null, null, gname) 具有相同的作用，其中 gname

     \* 是一个新生成的名称。自动生成的名称的形式为 "Thread-"+n，其中的 n 为整数。

     \*

     \* 另请参见： Thread(ThreadGroup, Runnable, String)

     \*/

    public Thread**()** **{**

        init**(null,** **null,** "Thread-" **+** nextThreadNum**(),** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(null, target,gname) 具有相同的作用，其中的 gname

     \* 是一个新生成的名称。 自动生成的名称的形式为 “Thread-”+n，其中的 n 为整数。

     \*

     \* 参数： target - 其 run 方法被调用的对象。 另请参见： Thread(ThreadGroup, Runnable, String)

     \*/

    public Thread**(**Runnable target**)** **{**

        init**(null,** target**,** "Thread-" **+** nextThreadNum**(),** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(group, target, gname) 具有相同的作用，其中的 gname

     \* 是一个新生成的名称。自动生成的名称的形式为 "Thread-"+n ，其中的 n 为整数。

     \*

     \* 参数： group - 线程组。 target - 其 run 方法被调用的对象。 抛出： SecurityException -

     \* 如果当前线程无法在指定的线程组中创建线程。 另请参见： Thread(ThreadGroup, Runnable, String)

     \*/

    public Thread**(**ThreadGroup group**,** Runnable target**)** **{**

        init**(**group**,** target**,** "Thread-" **+** nextThreadNum**(),** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(null, null, name) 具有相同的作用。

     \*

     \* 参数： name - 新线程的名称。 另请参见： Thread(ThreadGroup, Runnable, String)

     \*/

    public Thread**(**String name**)** **{**

        init**(null,** **null,** name**,** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(group, null, name) 具有相同的作用。

     \*

     \* 参数： group - 线程组。 name - 新线程的名称。 抛出： SecurityException -

     \* 如果当前线程无法在指定的线程组中创建线程。 另请参见： Thread(ThreadGroup, Runnable, String)

     \*/

    public Thread**(**ThreadGroup group**,** String name**)** **{**

        init**(**group**,** **null,** name**,** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象。这种构造方法与 Thread(null, target, name) 具有相同的作用。

     \*

     \* 参数： target - 其 run 方法被调用的对象。 name - 新线程的名称。 另请参见： Thread(ThreadGroup,

     \* Runnable, String)

     \*/

    public Thread**(**Runnable target**,** String name**)** **{**

        init**(null,** target**,** name**,** 0**);**

**}**

    /\*\*

     \* 分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，并作为 group 所引用的线程组的一员。

     \* 如果 group 为 null，并且有安全管理器，则该组由安全管理器的 getThreadGroup 方法确定。 如果 group 为

     \* null，并且没有安全管理器，或安全管理器的 getThreadGroup 方法返回 null， 则该组与创建新线程的线程被设定为相同的

     \* ThreadGroup。 如果有安全管理器，则其 checkAccess 方法通过 ThreadGroup 作为其参数被调用。 此外，当被重写

     \* getContextClassLoader 或 setContextClassLoader 方法的子类构造方法直接或间接调用时， 其

     \* checkPermission 方法通过

     \* RuntimePermission("enableContextClassLoaderOverride") 权限调用。其结果可能是

     \* SecurityException。

     \*

     \* 如果 target 参数不是 null，则 target 的 run 方法在启动该线程时调用。如果 target 参数为 null，则该线程的

     \* run 方法在该线程启动时调用。 新创建线程的优先级被设定为创建该线程的线程的优先级，即当前正在运行的线程的优先级。方法 setPriority

     \* 可用于将优先级更改为一个新值。 当且仅当创建新线程的线程当前被标记为守护线程时，新创建的线程才被标记为守护线程。方法 setDaemon

     \* 可用于改变线程是否为守护线程。

     \* 参数： group - 线程组。 target - 其 run 方法被调用的对象。 name - 新线程的名称。 抛出：

     \* SecurityException - 如果当前线程无法在指定的线程组中创建线程，或者无法重写上下文类加载器方法。

     \*/

    public Thread**(**ThreadGroup group**,** Runnable target**,** String name**)** **{**

        init**(**group**,** target**,** name**,** 0**);**

**}**

    /\*\*

     \* 除了允许指定线程堆栈大小以外，这种构造方法与 Thread(ThreadGroup,Runnable,String) 完全一样。

     \* 堆栈大小是虚拟机要为该线程堆栈分配的地址空间的近似字节数。 stackSize 参数（如果有）的作用具有高度的平台依赖性。

     \* 在某些平台上，指定一个较高的 stackSize 参数值可能使线程在抛出 StackOverflowError 之前达到较大的递归深度。

     \* 同样，指定一个较低的值将允许较多的线程并发地存在，且不会抛出 OutOfMemoryError（或其他内部错误）。 stackSize

     \* 参数的值与最大递归深度和并发程度之间的关系细节与平台有关。在某些平台上，stackSize 参数的值无论如何不会起任何作用。

     \* 作为建议，可以让虚拟机自由处理 stackSize 参数。如果指定值对于平台来说过低，则虚拟机可能使用某些特定于平台的最小值；

     \* 如果指定值过高，则虚拟机可能使用某些特定于平台的最大值。同样，虚拟机还会视情况自由地舍入指定值（或完全忽略它）。

     \*

     \* 将 stackSize 参数值指定为零将使这种构造方法与 Thread(ThreadGroup, Runnable, String)

     \* 构造方法具有完全相同的作用。

     \*

     \* 由于这种构造方法的行为具有平台依赖性，因此在使用它时要非常小心。执行特定计算所必需的线程堆栈大小可能会因 JRE 实现的不同而不同。

     \* 鉴于这种不同，仔细调整堆栈大小参数可能是必需的，而且可能要在支持应用程序运行的 JRE 实现上反复调整。

     \*

     \* 参数： stackSize - 新线程的预期堆栈大小，为零时表示忽略该参数。 抛出： SecurityException -

     \* 如果当前线程无法在指定的线程组中创建线程。

     \*/

    public Thread**(**ThreadGroup group**,** Runnable target**,** String name**,**

            long stackSize**)** **{**

        init**(**group**,** target**,** name**,** stackSize**);**

**}**

    /\*\*

     \* 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。 结果是两个线程并发地运行；当前线程（从调用返回给 start

     \* 方法）和另一个线程（执行其 run 方法）。 多次启动一个线程是非法的。特别是当线程已经结束执行后，不能再重新启动。 抛出：

     \* IllegalThreadStateException - 如果线程已经启动。

     \*/

    public synchronized void start**()** **{**

        /\*\*

         \* This method is not invoked for the main method thread or "system"

         \* group threads created/set up by the VM. Any new functionality added

         \* to this method in the future may have to also be added to the VM.

         \*

         \* A zero status value corresponds to state "NEW".

         \* threadStatus线程状态变量为0意味着状态为NEW

         \*/

**if** **(**threadStatus **!=** 0**)** // 如果线程状态不为0，则抛出线程状态非法的异常。这也意味着重复调用start()方法会抛出异常。所以一个线程要启动时只能调用一次start()方法。

**throw** **new** IllegalThreadStateException**();**

        group**.**add**(this);** // 否则，就把这个线程对象加到线程组中

        start0**();** // 然后调用native的start0()方法来启动新线程

**if** **(**stopBeforeStart**)** **{**

            stop0**(**throwableFromStop**);**

**}**

**}**

    private native void start0**();**

    /\*\*

     \* 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。

     \* Thread 的子类应该重写该方法。

     \*/

    public void run**()** **{**

**if** **(**target **!=** **null)** **{** // 如果有目标Runnable对象，那么执行目标Runnable对象的run方法，否则什么都不做。

            target**.**run**();**

**}**

**}**

    /\*\*

     \* This method is called by the system to give a Thread a chance to clean up

     \* before it actually exits.

     \*/

    private void exit**()** **{**

**if** **(**group **!=** **null)** **{**

            group**.**remove**(this);**

            group **=** **null;**

**}**

        /\* Aggressively null out all reference fields: see bug 4006245 \*/

        target **=** **null;**

        /\* Speed the release of some of these resources \*/

        threadLocals **=** **null;**

        inheritableThreadLocals **=** **null;**

        inheritedAccessControlContext **=** **null;**

        blocker **=** **null;**

        uncaughtExceptionHandler **=** **null;**

**}**

    /\*\*

     \* 已过时。 该方法具有固有的不安全性。用 Thread.stop 来终止线程将释放它已经锁定的所有监视器 （作为沿堆栈向上传播的未检查

     \* ThreadDeath 异常的一个自然后果）。

     \* 如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。 stop

     \* 的许多使用都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。

     \* 目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。 如果目标线程等待很长时间（例如基于一个条件变量），则应使用

     \* interrupt 方法来中断该等待。 有关更多信息，请参阅为何不赞成使用 Thread.stop、Thread.suspend 和

     \* Thread.resume？。

     \*

     \* 强迫线程停止执行。 如果安装了安全管理器，则以 this 作为其参数调用 checkAccess 方法。这可能引发

     \* SecurityException（在当前线程中）。

     \*

     \* 如果该线程不同于当前线程（即当前线程试图终止除它本身以外的某一线程），则安全管理器的 checkPermission 方法（带有

     \* RuntimePermission("stopThread") 参数）也会被调用。这会再次抛出

     \* SecurityException（在当前线程中）。

     \*

     \* 无论该线程在做些什么，它所代表的线程都被迫异常停止，并抛出一个新创建的 ThreadDeath 对象，作为异常。

     \*

     \* 停止一个尚未启动的线程是允许的。如果最后启动了该线程，它会立即终止。

     \*

     \* 应用程序通常不应试图捕获 ThreadDeath，除非它必须执行某些异常的清除操作（注意，抛出 ThreadDeath 将导致 try 语句的

     \* finally 子句在线程正式终止前执行）。如果 catch 子句捕获了一个 ThreadDeath

     \* 对象，则重新抛出该对象很重要，因为这样该线程才会真正终止。

     \*

     \* 对其他未捕获的异常作出反应的顶级错误处理程序不会打印输出消息，或者另外通知应用程序未捕获到的异常是否为 ThreadDeath 的一个实例。

     \*/

    @Deprecated

    public final void stop**()** **{**

        // If the thread is already dead, return.

        // A zero status value corresponds to "NEW".

**if** **((**threadStatus **!=** 0**)** **&&** **!**isAlive**())** **{**

**return;**

**}**

        stop1**(new** ThreadDeath**());**

**}**

    /\*\*

     \* 已过时。 该方法具有固有的不安全性。有关详细信息，请参阅

     \* stop()。该方法的附加危险是它可用于生成目标线程未准备处理的异常（包括若没有该方法该线程不太可能抛出的已检查的异常

     \* ）。有关更多信息，请参阅为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume？。

     \*

     \* 强迫线程停止执行。 如果安装了安全管理器，则调用该线程的 checkAccess 方法，这可能引发

     \* SecurityException（在当前线程中）。

     \*

     \* 如果该线程不同于当前线程（即当前线程试图终止除它本身以外的某一线程），或者 obj 不是 ThreadDeath 的一个实例，则安全管理器的

     \* checkPermission 方法（带有 RuntimePermission("stopThread") 参数）也会被调用。此外，这可能抛出

     \* SecurityException（在当前线程中）。

     \*

     \* 如果参数 obj 为 null，则抛出 NullPointerException（在当前线程中）。

     \*

     \* 无论该线程在做些什么，它所代表的线程都被迫异常停止，并抛出 Throwable 对象

     \* obj，作为一个异常。这是一种不正常的操作，通常情况下，应使用不带任何参数的 stop 方法。

     \*

     \* 停止一个尚未启动的线程是允许的。如果最后启动了该线程，它会立即终止。

     \*

     \*

     \* 参数： obj - 要抛出的可抛出对象。 抛出： SecurityException - 如果当前线程不能修改该线程。

     \* NullPointerException - 如果 obj 为 null。 另请参见： interrupt(), checkAccess(),

     \* run(), start(), stop(), SecurityManager.checkAccess(Thread),

     \* SecurityManager.checkPermission(java.security.Permission)

     \*/

    @Deprecated

    public final synchronized void stop**(**Throwable obj**)** **{**

        stop1**(**obj**);**

**}**

    /\*\*

     \* Common impl for stop() and stop(Throwable).

     \*/

    private final synchronized void stop1**(**Throwable th**)** **{**

        SecurityManager security **=** System**.**getSecurityManager**();**

**if** **(**security **!=** **null)** **{**

            checkAccess**();**

**if** **((this** **!=** Thread**.**currentThread**())**

**||** **(!(**th **instanceof** ThreadDeath**)))** **{**

                security**.**checkPermission**(**SecurityConstants**.**STOP\_THREAD\_PERMISSION**);**

**}**

**}**

        // A zero status value corresponds to "NEW"

**if** **(**threadStatus **!=** 0**)** **{**

            resume**();** // Wake up thread if it was suspended; no-op otherwise

            stop0**(**th**);**

**}** **else** **{**

            // Must do the null arg check that the VM would do with stop0

**if** **(**th **==** **null)** **{**

**throw** **new** NullPointerException**();**

**}**

            // Remember this stop attempt for if/when start is used

            stopBeforeStart **=** **true;**

            throwableFromStop **=** th**;**

**}**

**}**

    /\*\*

     \* 中断线程。 1)如果当前线程没有中断它自己（这在任何情况下都是允许的），则该线程的 checkAccess 方法就会被调用，这可能抛出

     \* SecurityException。

     \*

     \* 2).如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法， 或者该类的

     \* join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int)

     \* 方法过程中受阻，则其中断状态将被清除，它还将收到一个 InterruptedException。

     \*

     \* 如果该线程在可中断的通道上的 I/O 操作中受阻，则该通道将被关闭，该线程的中断状态将被设置并且该线程将收到一个

     \* ClosedByInterruptException。

     \*

     \* 如果该线程在一个 Selector 中受阻，则该线程的中断状态将被设置，它将立即从选择操作返回，并可能带有一个非零值，就好像调用了选择器的

     \* wakeup 方法一样。

     \*

     \* 如果以前的条件都没有保存，则该线程的中断状态将被设置。

     \*

     \* 中断一个不处于活动状态的线程不需要任何作用。

     \*

     \*

     \* 抛出： SecurityException - 如果当前线程无法修改该线程

     \*/

    public void interrupt**()** **{** // 使用起来很简单，直接使用t.interrupt(),即停止，但是如果停止时线程内部抛出异常的时候，线程内部还要进行一次this.interrupt()

**if** **(this** **!=** Thread**.**currentThread**())** // 如果当前对象不为当前线程，则校验权限。

            checkAccess**();**

        synchronized **(**blockerLock**)** **{** // 同步中断锁

            Interruptible b **=** blocker**;**

**if** **(**b **!=** **null)** **{**

                interrupt0**();** // Just to set the interrupt flag

                b**.**interrupt**(this);**

**return;**

**}**

**}**

        interrupt0**();**

**}**

    /\*\*

     \* 测试当前线程是否已经中断。线程的中断状态 由该方法清除。 换句话说，如果连续两次调用该方法，则第二次调用将返回

     \* false（在第一次调用已清除了其中断状态之后，且第二次调用检验完中断状态前，当前线程再次中断的情况除外）。

     \* 线程中断被忽略，因为在中断时不处于活动状态的线程将由此返回 false 的方法反映出来。

     \*

     \*

     \* 返回： 如果当前线程已经中断，则返回 true；否则返回 false。

     \*/

    public static boolean interrupted**()** **{**

**return** currentThread**().**isInterrupted**(true);**

**}**

    /\*\*

     \* 测试线程是否已经中断。线程的中断状态 不受该方法的影响。 线程中断被忽略，因为在中断时不处于活动状态的线程将由此返回 false 的方法反映出来。

     \*

     \*

     \* 返回： 如果该线程已经中断，则返回 true；否则返回 false。 另请参见： interrupted()

     \*/

    public boolean isInterrupted**()** **{**

**return** isInterrupted**(false);**

**}**

    /\*\*

     \* Tests if some Thread has been interrupted. The interrupted state is reset

     \* or not based on the value of ClearInterrupted that is passed.

     \*/

    private native boolean isInterrupted**(**boolean ClearInterrupted**);**

    /\*\*

     \* 已过时。 该方法最初用于破坏该线程，但不作任何清除。它所保持的任何监视器都会保持锁定状态。 不过，该方法决不会被实现。即使要实现，它也极有可能以

     \* suspend() 方式被死锁。 如果目标线程被破坏时保持一个保护关键系统资源的锁，则任何线程在任何时候都无法再次访问该资源。

     \* 如果另一个线程曾试图锁定该资源，则会出现死锁。这类死锁通常会证明它们自己是“冻结”的进程。 有关更多信息，请参阅为何不赞成使用

     \* Thread.stop、Thread.suspend 和 Thread.resume？。

     \*

     \* 抛出 NoSuchMethodError。

     \*

     \* 抛出： NoSuchMethodError - 始终

     \*/

    @Deprecated

    public void destroy**()** **{**

**throw** **new** NoSuchMethodError**();**

**}**

    /\*\*

     \* 测试线程是否处于活动状态。如果线程已经启动且尚未终止，则为活动状态。

     \*

     \* 返回： 如果该线程处于活动状态，则返回 true；否则返回 false。

     \*/

    public final native boolean isAlive**();**

    /\*\*

     \* 已过时。 该方法已经遭到反对，因为它具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，

     \* 则在目标线程重新开始以前任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume

     \* 之前锁定该监视器，则会发生死锁。这类死锁通常会证明自己是“冻结”的进程。有关更多信息，请参阅为何不赞成使用

     \* Thread.stop、Thread.suspend 和 Thread.resume？。

     \*

     \* 挂起线程。 首先，调用线程的 checkAccess 方法，且不带任何参数。这可能抛出 SecurityException（在当前线程中）。

     \*

     \* 如果线程处于活动状态则被挂起，且不再有进一步的活动，除非重新开始。

     \*

     \*

     \* 抛出： SecurityException - 如果当前线程不能修改该线程。 另请参见： checkAccess()

     \*/

    @Deprecated

    public final void suspend**()** **{**

        checkAccess**();**

        suspend0**();**

**}**

    /\*\*

     \* 已过时。 该方法只与 suspend() 一起使用，但 suspend() 已经遭到反对，因为它具有死锁倾向。有关更多信息，请参阅为何不赞成使用

     \* Thread.stop、Thread.suspend 和 Thread.resume？。

     \*

     \* 重新开始挂起的进程。 首先，调用线程的 checkAccess 方法，且不带任何参数。这可能抛出

     \* SecurityException（在当前线程中）。

     \*

     \* 如果线程处于活动状态但被挂起，则它会在执行过程中重新开始并允许继续活动。

     \*

     \*

     \* 抛出： SecurityException - 如果当前线程不能修改该线程。 另请参见： checkAccess(), suspend()

     \*/

    @Deprecated

    public final void resume**()** **{**

        checkAccess**();**

        resume0**();**

**}**

    /\*\*

     \* 更改线程的优先级。 首先调用线程的 checkAccess 方法，且不带任何参数。这可能抛出 SecurityException。

     \*

     \* 在其他情况下，线程优先级被设定为指定的 newPriority 和该线程的线程组的最大允许优先级相比较小的一个。

     \*

     \*

     \* 参数： newPriority - 要为线程设定的优先级 抛出： IllegalArgumentException - 如果优先级不在

     \* MIN\_PRIORITY 到 MAX\_PRIORITY 范围内。 SecurityException - 如果当前线程无法修改该线程。 另请参见：

     \* getPriority(), checkAccess(), getThreadGroup(), MAX\_PRIORITY,

     \* MIN\_PRIORITY, ThreadGroup.getMaxPriority()

     \*/

    public final void setPriority**(**int newPriority**)** **{**

        ThreadGroup g**;**

        checkAccess**();**

**if** **(**newPriority **>** MAX\_PRIORITY **||** newPriority **<** MIN\_PRIORITY**)** **{** // 传入的优先级变量必须在1-10之间，否则抛异常。

**throw** **new** IllegalArgumentException**();**

**}**

        // 当前线程的权限大于线程组的权限，则赋予线程组的最大权限。

**if** **((**g **=** getThreadGroup**())** **!=** **null)** **{** // 使用线程组中最大的优先级变量，其实传入的优先级变量不一定是真正的优先级。

**if** **(**newPriority **>** g**.**getMaxPriority**())** **{**

                newPriority **=** g**.**getMaxPriority**();**

**}**

            setPriority0**(**priority **=** newPriority**);**

**}**

**}**

    /\*\*

     \* 返回线程的优先级。

     \*

     \* 返回： 该线程的优先级。 另请参见： setPriority(int)

     \*/

    public final int getPriority**()** **{**

**return** priority**;**

**}**

    /\*\*

     \* 改变线程名称，使之与参数 name 相同。 首先调用线程的 checkAccess 方法，且不带任何参数。这可能抛出

     \* SecurityException。

     \*

     \*

     \* 参数： name - 该线程的新名称。 抛出： SecurityException - 如果当前线程不能修改该线程。 另请参见：

     \* getName(), checkAccess()

     \*/

    public final void setName**(**String name**)** **{**

        checkAccess**();** // 判定当前运行的线程是否有权修改该线程.

**this.**name **=** name**.**toCharArray**();**

**}**

    /\*\*

     \* 返回该线程的名称。

     \*

     \* 返回： 该线程的名称。 另请参见： setName(String)

     \*/

    public final String getName**()** **{**

**return** String**.**valueOf**(**name**);**

**}**

    /\*\*

     \* 返回该线程所属的线程组。 如果该线程已经终止（停止运行），该方法则返回 null。

     \*

     \* 返回： 该线程的线程组。

     \*/

    public final ThreadGroup getThreadGroup**()** **{**

**return** group**;**

**}**

    /\*\*

     \* 返回当前线程的线程组中活动线程的数目。

     \*

     \* 返回： 当前线程的线程组中活动线程的数目。

     \*/

    public static int activeCount**()** **{**

**return** currentThread**().**getThreadGroup**().**activeCount**();**

**}**

    /\*\*

     \* 将当前线程的线程组及其子组中的每一个活动线程复制到指定的数组中。该方法只调用当前线程的线程组的 enumerate 方法，且带有数组参数。

     \* 首先，如果有安全管理器，则 enumerate 方法调用安全管理器的 checkAccess 方法，并将线程组作为其参数。这可能导致抛出

     \* SecurityException。

     \*

     \*

     \* 参数： tarray - 要复制到的线程对象数组 返回： 放入该数组的线程数 抛出： SecurityException -

     \* 如果安全管理器存在，并且其 checkAccess 方法不允许该操作。 另请参见：

     \* ThreadGroup.enumerate(Thread[]), SecurityManager.checkAccess(ThreadGroup)

     \*/

    public static int enumerate**(**Thread tarray**[])** **{**

**return** currentThread**().**getThreadGroup**().**enumerate**(**tarray**);**

**}**

    /\*\*

     \* 已过时。 该调用的定义依赖于 suspend()，但它遭到了反对。此外，该调用的结果从来都不是意义明确的。

     \*

     \* 计算该线程中的堆栈帧数。线程必须挂起。

     \*

     \* 返回： 该线程中的堆栈帧数。 抛出： IllegalThreadStateException - 如果该线程未挂起。

     \*/

    @Deprecated

    public native int countStackFrames**();**

    /\*\*

     \* 等待该线程终止的时间最长为 millis 毫秒。超时为 0 意味着要一直等下去。

     \*

     \* 参数： millis - 以毫秒为单位的等待时间。 抛出： InterruptedException -

     \* 如果任何线程中断了当前线程。当抛出该异常时，当前线程的中断状态 被清除。

     \*/

    public final synchronized void join**(**long millis**)**

**throws** InterruptedException **{**

        long base **=** System**.**currentTimeMillis**();**

        long now **=** 0**;**

**if** **(**millis **<** 0**)** **{**

**throw** **new** IllegalArgumentException**(**"timeout value is negative"**);**

**}**

**if** **(**millis **==** 0**)** **{** // 如果传入的等待时间为0，那判断线程是否活着，如果活着，那么一直等待。即传入等待时间为0时，就是一直等待下去。

**while** **(**isAlive**())** **{**

                wait**(**0**);**

**}**

**}** **else** **{**

**while** **(**isAlive**())** **{**

                long delay **=** millis **-** now**;** // 等待的具体时间

**if** **(**delay **<=** 0**)** **{**

**break;**

**}**

                wait**(**delay**);**

                now **=** System**.**currentTimeMillis**()** **-** base**;**

**}**

**}**

**}**

    /\*\*

     \* 等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

     \*

     \* 参数： millis - 以毫秒为单位的等待时间。 nanos - 要等待的 0-999999 附加纳秒。 抛出：

     \* IllegalArgumentException - 如果 millis 值为负，则 nanos 的值不在 0-999999 范围内。

     \* InterruptedException - 如果任何线程中断了当前线程。当抛出该异常时，当前线程的中断状态 被清除。

     \*/

    public final synchronized void join**(**long millis**,** int nanos**)**

**throws** InterruptedException **{**

**if** **(**millis **<** 0**)** **{**

**throw** **new** IllegalArgumentException**(**"timeout value is negative"**);**

**}**

**if** **(**nanos **<** 0 **||** nanos **>** 999999**)** **{**

**throw** **new** IllegalArgumentException**(**

                    "nanosecond timeout value out of range"**);**

**}**

**if** **(**nanos **>=** 500000 **||** **(**nanos **!=** 0 **&&** millis **==** 0**))** **{**

            millis**++;**

**}**

        join**(**millis**);**

**}**

    /\*\*

     \* 等待该线程终止。

     \*

     \* 抛出： InterruptedException - 如果任何线程中断了当前线程。当抛出该异常时，当前线程的中断状态 被清除。

     \*/

    public final void join**()** **throws** InterruptedException **{**

        join**(**0**);**

**}**

    /\*\*

     \* 将当前线程的堆栈跟踪打印至标准错误流。该方法仅用于调试。

     \*

     \* 另请参见： Throwable.printStackTrace()

     \*/

    public static void dumpStack**()** **{**

**new** Exception**(**"Stack trace"**).**printStackTrace**();**

**}**

    /\*\*

     \* 将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。 该方法必须在启动线程前调用。

     \*

     \* 该方法首先调用该线程的 checkAccess 方法，且不带任何参数。这可能抛出 SecurityException（在当前线程中）。

     \*

     \*

     \* 参数： on - 如果为 true，则将该线程标记为守护线程。 抛出： IllegalThreadStateException -

     \* 如果该线程处于活动状态。 SecurityException - 如果当前线程无法修改该线程。

     \*/

    public final void setDaemon**(**boolean on**)** **{**

        checkAccess**();**

**if** **(**isAlive**())** **{**

**throw** **new** IllegalThreadStateException**();**

**}**

        daemon **=** on**;**

**}**

    /\*\*

     \* 测试该线程是否为守护线程。

     \*/

    public final boolean isDaemon**()** **{**

**return** daemon**;**

**}**

    /\*\*

     \* 判定当前运行的线程是否有权修改该线程。 如果有安全管理器，则调用其 checkAccess 方法，并将该线程作为其参数。这可能导致抛出

     \* SecurityException。

     \*/

    public final void checkAccess**()** **{** // 判定当前运行的线程是否有权修改该线程. 对线程的许多操作都要调用该方法。

        SecurityManager security **=** System**.**getSecurityManager**();** // 获取安全管理器，如果安全管理器不为空，则用安全管理器去判断当前运行的线程是否有权修改该线程.

**if** **(**security **!=** **null)** **{**

            security**.**checkAccess**(this);**

**}**

**}**

    /\*\*

     \* 返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

     \*/

    public String toString**()** **{**

        ThreadGroup group **=** getThreadGroup**();**

**if** **(**group **!=** **null)** **{**

**return** "Thread[" **+** getName**()** **+** "," **+** getPriority**()** **+** ","

**+** group**.**getName**()** **+** "]"**;**

**}** **else** **{**

**return** "Thread[" **+** getName**()** **+** "," **+** getPriority**()** **+** "," **+** "" **+** "]"**;**

**}**

**}**

    /\*\*

     \* 返回该线程的上下文 ClassLoader。上下文 ClassLoader 由线程创建者提供，供运行于该线程中的代码在加载类和资源时使用。

     \* 如果未设定，则默认为父线程的 ClassLoader 上下文。原始线程的上下文 ClassLoader 通常设定为用于加载应用程序的类加载器。

     \* 首先，如果有安全管理器，并且调用者的类加载器不是 null，也不同于其上下文类加载器正在被请求的线程上下文类加载器的祖先， 则通过

     \* RuntimePermission("getClassLoader") 权限调用该安全管理器的 checkPermission

     \* 方法，查看是否可以获取上下文 ClassLoader。

     \*

     \*

     \* 返回： 该线程的上下文 ClassLoader 从以下版本开始： 1.2 另请参见：

     \* setContextClassLoader(java.lang.ClassLoader),

     \* SecurityManager.checkPermission(java.security.Permission),

     \* RuntimePermission

     \*/

    public ClassLoader getContextClassLoader**()** **{**

**if** **(**contextClassLoader **==** **null)**

**return** **null;**

        SecurityManager sm **=** System**.**getSecurityManager**();**

**if** **(**sm **!=** **null)** **{**

            ClassLoader ccl **=** ClassLoader**.**getCallerClassLoader**();**

**if** **(**ccl **!=** **null** **&&** ccl **!=** contextClassLoader

**&&** **!**contextClassLoader**.**isAncestor**(**ccl**))** **{**

                sm**.**checkPermission**(**SecurityConstants**.**GET\_CLASSLOADER\_PERMISSION**);**

**}**

**}**

**return** contextClassLoader**;**

**}**

    /\*\*

     \* 设置该线程的上下文 ClassLoader。上下文 ClassLoader

     \* 可以在创建线程设置，并允许创建者在加载类和资源时向该线程中运行的代码提供适当的类加载器。 首先，如果有安全管理器，则通过

     \* RuntimePermission("setContextClassLoader") 权限调用其 checkPermission

     \* 方法，查看是否可以设置上下文 ClassLoader。

     \*

     \*

     \* 参数： cl - 该线程的上下文 ClassLoader 抛出： SecurityException - 如果当前线程无法设置上下文

     \* ClassLoader。

     \*/

    public void setContextClassLoader**(**ClassLoader cl**)** **{**

        SecurityManager sm **=** System**.**getSecurityManager**();**

**if** **(**sm **!=** **null)** **{**

            sm**.**checkPermission**(new** RuntimePermission**(**"setContextClassLoader"**));**

**}**

        contextClassLoader **=** cl**;**

**}**

    /\*\*

     \* 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 该方法旨在使程序能够断言当前线程已经保持一个指定的锁：

     \*

     \* assert Thread.holdsLock(obj);

     \*

     \* 参数： obj - 用于测试锁所属权的对象 返回： 如果当前线程在指定的对象上保持监视器锁，则返回 true。

     \*/

    public static native boolean holdsLock**(**Object obj**);**

    private static final StackTraceElement**[]** EMPTY\_STACK\_TRACE **=** **new** StackTraceElement**[**0**];**

    /\*\*

     \* 返回一个表示该线程堆栈转储的堆栈跟踪元素数组。如果该线程尚未启动或已经终止，则该方法将返回一个零长度数组。如果返回的数组不是零长度的，

     \* 则其第一个元素代表堆栈顶，它是该序列中最新的方法调用。最后一个元素代表堆栈底，是该序列中最旧的方法调用。

     \* 如果有安全管理器，并且该线程不是当前线程，则通过 RuntimePermission("getStackTrace") 权限调用安全管理器的

     \* checkPermission 方法，查看是否可以获取堆栈跟踪。

     \*

     \* 某些虚拟机在某些情况下可能会从堆栈跟踪中省略一个或多个堆栈帧。在极端情况下，没有该线程堆栈跟踪信息的虚拟机可以从该方法返回一个零长度数组。

     \*

     \*

     \* 返回： StackTraceElement 数组，每个数组代表一个堆栈帧。 抛出： SecurityException -

     \* 如果安全管理器存在，并且其 checkPermission 方法不允许获取线程的堆栈跟踪。 从以下版本开始： 1.5 另请参见：

     \* SecurityManager.checkPermission(java.security.Permission),

     \* RuntimePermission, Throwable.getStackTrace()

     \*/

    public StackTraceElement**[]** getStackTrace**()** **{**

**if** **(this** **!=** Thread**.**currentThread**())** **{**

            // check for getStackTrace permission

            SecurityManager security **=** System**.**getSecurityManager**();**

**if** **(**security **!=** **null)** **{**

                security**.**checkPermission**(**SecurityConstants**.**GET\_STACK\_TRACE\_PERMISSION**);**

**}**

            // optimization so we do not call into the vm for threads that

            // have not yet started or have terminated

**if** **(!**isAlive**())** **{**

**return** EMPTY\_STACK\_TRACE**;**

**}**

            StackTraceElement**[][]** stackTraceArray **=** dumpThreads**(new** Thread**[]** **{** **this** **});**

            StackTraceElement**[]** stackTrace **=** stackTraceArray**[**0**];**

            // a thread that was alive during the previous isAlive call may have

            // since terminated, therefore not having a stacktrace.

**if** **(**stackTrace **==** **null)** **{**

                stackTrace **=** EMPTY\_STACK\_TRACE**;**

**}**

**return** stackTrace**;**

**}** **else** **{**

            // Don't need JVM help for current thread

**return** **(new** Exception**()).**getStackTrace**();**

**}**

**}**

    /\*\*

     \* 返回所有活动线程的堆栈跟踪的一个映射。映射键是线程，而每个映射值都是一个 StackTraceElement 数组，该数组表示相应 Thread

     \* 的堆栈转储。返回的堆栈跟踪的格式都是针对 getStackTrace 方法指定的。

     \* 在调用该方法的同时，线程可能也在执行。每个线程的堆栈跟踪仅代表一个快照

     \* ，并且每个堆栈跟踪都可以在不同时间获得。如果虚拟机没有线程的堆栈跟踪信息，则映射值中将返回一个零长度数组。

     \*

     \* 如果有安全管理器，则通过 RuntimePermission("getStackTrace") 权限和

     \* RuntimePermission("modifyThreadGroup") 权限调用其 checkPermission

     \* 方法，查看是否可以获取所有线程的堆栈跟踪。

     \*

     \*

     \* 返回： 从 Thread 到 StackTraceElement 数组的一个 Map，代表相应线程的堆栈跟踪。 抛出：

     \* SecurityException - 如果安全管理器存在，并且其 checkPermission 方法不允许获取线程的堆栈跟踪。

     \* 从以下版本开始： 1.5 另请参见： getStackTrace(),

     \* SecurityManager.checkPermission(java.security.Permission),

     \* RuntimePermission, Throwable.getStackTrace()

     \*/

    public static Map**<**Thread**,** StackTraceElement**[]>** getAllStackTraces**()** **{**

        // check for getStackTrace permission

        SecurityManager security **=** System**.**getSecurityManager**();**

**if** **(**security **!=** **null)** **{**

            security**.**checkPermission**(**SecurityConstants**.**GET\_STACK\_TRACE\_PERMISSION**);**

            security**.**checkPermission**(**SecurityConstants**.**MODIFY\_THREADGROUP\_PERMISSION**);**

**}**

        // Get a snapshot of the list of all threads

        Thread**[]** threads **=** getThreads**();**

        StackTraceElement**[][]** traces **=** dumpThreads**(**threads**);**

        Map**<**Thread**,** StackTraceElement**[]>** m **=** **new** HashMap**<**Thread**,** StackTraceElement**[]>(**

                threads**.**length**);**

**for** **(**int i **=** 0**;** i **<** threads**.**length**;** i**++)** **{**

            StackTraceElement**[]** stackTrace **=** traces**[**i**];**

**if** **(**stackTrace **!=** **null)** **{**

                m**.**put**(**threads**[**i**],** stackTrace**);**

**}**

            // else terminated so we don't put it in the map

**}**

**return** m**;**

**}**

    private static final RuntimePermission SUBCLASS\_IMPLEMENTATION\_PERMISSION **=** **new** RuntimePermission**(**

            "enableContextClassLoaderOverride"**);**

    /\*\* cache of subclass security audit results \*/

    private static final SoftCache subclassAudits **=** **new** SoftCache**(**10**);**

    /\*\*

     \* Verifies that this (possibly subclass) instance can be constructed

     \* without violating security constraints: the subclass must not override

     \* security-sensitive non-final methods, or else the

     \* "enableContextClassLoaderOverride" RuntimePermission is checked.

     \*/

    private static boolean isCCLOverridden**(**Class cl**)** **{**

**if** **(**cl **==** Thread**.**class**)**

**return** **false;**

        Boolean result **=** **null;**

        synchronized **(**subclassAudits**)** **{**

            result **=** **(**Boolean**)** subclassAudits**.**get**(**cl**);**

**if** **(**result **==** **null)** **{**

                /\*

                 \* Note: only new Boolean instances (i.e., not Boolean.TRUE or

                 \* Boolean.FALSE) must be used as cache values, otherwise cache

                 \* entry will pin associated class.

                 \*/

                result **=** **new** Boolean**(**auditSubclass**(**cl**));**

                subclassAudits**.**put**(**cl**,** result**);**

**}**

**}**

**return** result**.**booleanValue**();**

**}**

    /\*\*

     \* Performs reflective checks on given subclass to verify that it doesn't

     \* override security-sensitive non-final methods. Returns true if the

     \* subclass overrides any of the methods, false otherwise.

     \*/

    private static boolean auditSubclass**(**final Class subcl**)** **{**

        Boolean result **=** **(**Boolean**)** AccessController

**.**doPrivileged**(new** PrivilegedAction**()** **{**

                    public Object run**()** **{**

**for** **(**Class cl **=** subcl**;** cl **!=** Thread**.**class**;** cl **=** cl

**.**getSuperclass**())** **{**

**try** **{**

                                cl**.**getDeclaredMethod**(**"getContextClassLoader"**,**

**new** Class**[**0**]);**

**return** Boolean**.**TRUE**;**

**}** **catch** **(**NoSuchMethodException ex**)** **{**

**}**

**try** **{**

                                Class**[]** params **=** **{** ClassLoader**.**class **};**

                                cl**.**getDeclaredMethod**(**"setContextClassLoader"**,**

                                        params**);**

**return** Boolean**.**TRUE**;**

**}** **catch** **(**NoSuchMethodException ex**)** **{**

**}**

**}**

**return** Boolean**.**FALSE**;**

**}**

**});**

**return** result**.**booleanValue**();**

**}**

    private native static StackTraceElement**[][]** dumpThreads**(**Thread**[]** threads**);**

    private native static Thread**[]** getThreads**();**

    /\*\*

     \* 返回该线程的标识符。线程 ID 是一个正的 long 数，在创建该线程时生成。线程 ID 是唯一的，并终生不变。线程终止时，该线程 ID

     \* 可以被重新使用。

     \*/

    public long getId**()** **{**

**return** tid**;**

**}**

    /\*\*

     \* 线程状态。线程可以处于下列状态之一：

     \*

     \* NEW 至今尚未启动的线程处于这种状态。 RUNNABLE 正在 Java 虚拟机中执行的线程处于这种状态。 BLOCKED

     \* 受阻塞并等待某个监视器锁的线程处于这种状态。 WAITING 无限期地等待另一个线程来执行某一特定操作的线程处于这种状态。

     \* TIMED\_WAITING 等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态。 TERMINATED 已退出的线程处于这种状态。

     \* 在给定时间点上，一个线程只能处于一种状态。这些状态是虚拟机状态，它们并没有反映所有操作系统线程状态。

     \*/

    public enum State **{**

        /\*\*

         \* 至今尚未启动的线程的状态。

         \*/

        NEW**,**

        /\*\*

         \* 可运行线程的线程状态。处于可运行状态的某一线程正在 Java 虚拟机中运行，但它可能正在等待操作系统中的其他资源，比如处理器。

         \*/

        RUNNABLE**,**

        /\*\*

         \* 受阻塞并且正在等待监视器锁的某一线程的线程状态。处于受阻塞状态的某一线程正在等待监视器锁，以便进入一个同步的块/方法， 或者在调用

         \* Object.wait 之后再次进入同步的块/方法。

         \*/

        BLOCKED**,**

        /\*\*

         \* 某一等待线程的线程状态。某一线程因为调用下列方法之一而处于等待状态： 不带超时值的 Object.wait 不带超时值的

         \* Thread.join LockSupport.park 处于等待状态的线程正等待另一个线程，以执行特定操作。

         \* 例如，已经在某一对象上调用了 Object.wait() 的线程正等待另一个线程， 以便在该对象上调用 Object.notify() 或

         \* Object.notifyAll()。已经调用了 Thread.join() 的线程正在等待指定线程终止。

         \*/

        WAITING**,**

        /\*\*

         \* 具有指定等待时间的某一等待线程的线程状态。某一线程因为调用以下带有指定正等待时间的方法之一而处于定时等待状态： Thread.sleep

         \* 带有超时值的 Object.wait 带有超时值的 Thread.join LockSupport.parkNanos

         \* LockSupport.parkUntil

         \*/

        TIMED\_WAITING**,**

        /\*\*

         \* 已终止线程的线程状态。线程已经结束执行。

         \*/

        TERMINATED**;**

**}**

    /\*\*

     \* 返回该线程的状态。 该方法用于监视系统状态，不用于同步控制。

     \*

     \* 返回： 该线程的状态。 从以下版本开始： 1.5

     \*/

    public State getState**()** **{**

        // get current thread state

**return** sun**.**misc**.**VM**.**toThreadState**(**threadStatus**);**

**}**

    // Added in JSR-166

    /\*\*

     \* 当 Thread 因未捕获的异常而突然终止时，调用处理程序的接口。

     \*

     \* 当某一线程因未捕获的异常而即将终止时，Java 虚拟机将使用 Thread.getUncaughtExceptionHandler() 查询

     \* 该线程以获得其 UncaughtExceptionHandler 的线程，并调用处理程序的 uncaughtException

     \* 方法，将线程和异常作为参数传递。 如果某一线程没有明确设置其 UncaughtExceptionHandler，则将它的 ThreadGroup

     \* 对象作为其 UncaughtExceptionHandler。 如果 ThreadGroup

     \* 对象对处理异常没有什么特殊要求，那么它可以将调用转发给默认的未捕获异常处理程序。

     \*/

    public interface UncaughtExceptionHandler **{**

        /\*\*

         \* Method invoked when the given thread terminates due to the given

         \* uncaught exception.

         \* <p>

         \* Any exception thrown by this method will be ignored by the Java

         \* Virtual Machine.

         \*

         \* @param t

         \*            the thread

         \* @param e

         \*            the exception

         \*/

        void uncaughtException**(**Thread t**,** Throwable e**);**

**}**

    // null unless explicitly set

    private volatile UncaughtExceptionHandler uncaughtExceptionHandler**;**

    // null unless explicitly set

    private static volatile UncaughtExceptionHandler defaultUncaughtExceptionHandler**;**

    /\*\*

     \* 设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序。 未捕获到的异常处理首先由线程控制，然后由线程的ThreadGroup对象控制，最后由未捕获到的默认异常处理程序控制。如果线程不设置明确的未捕获到的异常处理程序，并且该线程的线程组（包括父线程组）未特别指定其

     \* uncaughtException 方法，则将调用默认处理程序的 uncaughtException 方法。

     \*

     \* 通过设置未捕获到的默认异常处理程序，应用程序可以为那些已经接受系统提供的任何“默认”行为的线程改变未捕获到的异常处理方式（

     \* 如记录到某一特定设备或文件）。

     \*

     \* 请注意，未捕获到的默认异常处理程序通常不应顺从该线程的 ThreadGroup 对象，因为这可能导致无限递归。

     \*

     \*

     \* 参数： eh - 用作未捕获到的默认异常处理程序的对象。 如果为 null，则没有默认处理程序。 抛出： SecurityException -

     \* 如果安全管理器存在并拒绝 RuntimePermission ("setDefaultUncaughtExceptionHandler")

     \* 从以下版本开始：

     \*/

    public static void setDefaultUncaughtExceptionHandler**(**

            UncaughtExceptionHandler eh**)** **{**

        SecurityManager sm **=** System**.**getSecurityManager**();**

**if** **(**sm **!=** **null)** **{**

            sm**.**checkPermission**(new** RuntimePermission**(**

                    "setDefaultUncaughtExceptionHandler"**));**

**}**

        defaultUncaughtExceptionHandler **=** eh**;**

**}**

    /\*\*

     \* 返回线程由于未捕获到异常而突然终止时调用的默认处理程序。如果返回值为 null，则没有默认处理程序。

     \*/

    public static UncaughtExceptionHandler getDefaultUncaughtExceptionHandler**()** **{**

**return** defaultUncaughtExceptionHandler**;**

**}**

    /\*\*

     \* 返回该线程由于未捕获到异常而突然终止时调用的处理程序。 如果该线程尚未明确设置未捕获到的异常处理程序，则返回该线程的 ThreadGroup

     \* 对象，除非该线程已经终止，在这种情况下，将返回 null。

     \*

     \* 从以下版本开始： 1.5

     \*/

    public UncaughtExceptionHandler getUncaughtExceptionHandler**()** **{**

**return** uncaughtExceptionHandler **!=** **null** **?** uncaughtExceptionHandler

**:** group**;**

**}**

    /\*\*

     \* 设置该线程由于未捕获到异常而突然终止时调用的处理程序。

     \* 通过明确设置未捕获到的异常处理程序，线程可以完全控制它对未捕获到的异常作出响应的方式。如果没有设置这样的处理程序，则该线程的

     \* ThreadGroup 对象将充当其处理程序。

     \*

     \*

     \* 参数： eh - 用作该线程未捕获到的异常处理程序的对象。如果为 null，则该线程没有明确的处理程序。 抛出：

     \* SecurityException - 如果当前线程无权修改该线程。 从以下版本开始： 1.5 另请参见：

     \* setDefaultUncaughtExceptionHandler

     \* (java.lang.Thread.UncaughtExceptionHandler),

     \* ThreadGroup.uncaughtException(java.lang.Thread, java.lang.Throwable)

     \*/

    public void setUncaughtExceptionHandler**(**UncaughtExceptionHandler eh**)** **{**

        checkAccess**();**

        uncaughtExceptionHandler **=** eh**;**

**}**

    /\*\*

     \* Dispatch an uncaught exception to the handler. This method is intended to

     \* be called only by the JVM.

     \*/

    private void dispatchUncaughtException**(**Throwable e**)** **{**

        getUncaughtExceptionHandler**().**uncaughtException**(this,** e**);**

**}**

    /\* Some private helper methods \*/

    private native void setPriority0**(**int newPriority**);**

    private native void stop0**(**Object o**);**

    private native void suspend0**();**

    private native void resume0**();**

    private native void interrupt0**();**

**}**

# 使用举例

## Daemon

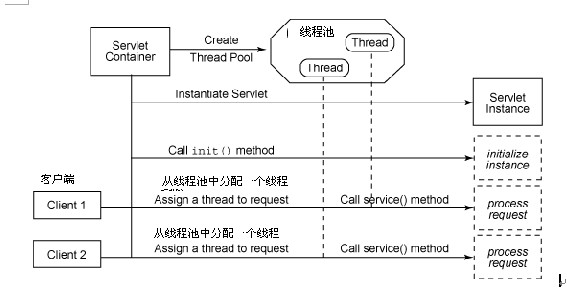
Daemon的作用是为其他线程的运行提供服务，比如说GC线程。通过成员变量daemon 来标志

其实User Thread线程和Daemon Thread守护线程本质上来说去没啥区别的，唯一的区别之处就在虚拟机的离开：如果User Thread全部撤离，那么Daemon Thread也就没啥线程好服务的了，所以虚拟机也就退出了。

**JRE判断程序是否执行结束的标准是所有的前台执线程行完毕了**，而不管后台线程的状态，因此，在使用后台线程候一定要注意这个问题。

daemon Thread实际应用在那里呢？举个例子，web服务器中的Servlet，容器启动时后台初始化一个服务线程，即调度线程，负责处理http请求，然后每个请求过来调度线程从线程池中取出一个工作者线程来处理该请求，从而实现并发控制的目的。

网上摘的一个图，方便大家理解：



### 代码示例

/\*\*

 a.setDaemon(false); //可以被写入daemon字符串,默认值

 a.setDaemon(true);//无法被写入daemon字符串,默认值

 \*/

public class ThreadDaemonTest **{**

    public static void main**(**String**[]** args**)** **{**

        Thread a **=** **new** Thread**(new** TestRunnable**());**

        //a.setDaemon(true);

        a**.**start**();**

**}**

    static class TestRunnable **implements** Runnable **{**

        public void run**()** **{**

**try** **{**

                Thread**.**sleep**(**1000**);**// 守护线程阻塞1秒后运行

                File f **=** **new** File**(**"daemon.txt"**);**

                FileOutputStream os **=** **new** FileOutputStream**(**f**,** **true);**

                os**.**write**(**"daemon"**.**getBytes**());**

**}**

**catch** **(**IOException e1**)** **{**

                e1**.**printStackTrace**();**

**}**

**catch** **(**InterruptedException e2**)** **{**

                e2**.**printStackTrace**();**

**}**

**}**

**}**

**}**

}

## sleep方法不会释放锁

/\*\*

 \* sleep相当于让线程睡眠（是抱着睡觉的），交出CPU，让CPU去执行其他的任务。

　　但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，

         则即使调用sleep方法，其他线程也无法访问这个对象。看下面这个例子就清楚了

 \*/

public class ThreadSleepTest **{**

    private int i **=** 0**;**

    private Object object **=** **new** Object**();**

    public static void main**(**String**[]** args**)** **throws** IOException  **{**

    ThreadSleepTest test **=** **new** ThreadSleepTest**();**

        MyThread thread1 **=** test**.new** MyThread**();**

        MyThread thread2 **=** test**.new** MyThread**();**

        thread1**.**start**();**

        thread2**.**start**();**

/\*        i:1

          线程Thread-0进入睡眠状态

          线程Thread-0睡眠结束

        i:2

        i:3

          线程Thread-1进入睡眠状态

          线程Thread-1睡眠结束

        i:4\*/

**}**

    class MyThread **extends** Thread**{**

        @Override

        public void run**()** **{**

            synchronized **(**object**)** **{**

                i**++;**

                System**.**out**.**println**(**"i:"**+**i**);**

**try** **{**

                    System**.**out**.**println**(**"线程"**+**Thread**.**currentThread**().**getName**()+**"进入睡眠状态"**);**

                    Thread**.**currentThread**().**sleep**(**10000**);**

**}** **catch** **(**InterruptedException e**)** **{**

                    // TODO: handle exception

**}**

                System**.**out**.**println**(**"线程"**+**Thread**.**currentThread**().**getName**()+**"睡眠结束"**);**

                i**++;**

                System**.**out**.**println**(**"i:"**+**i**);**

**}**

**}**

**}**

**}**

## 中断线程

/\*\*我该什么方法来中断你，我的线程！！！！

 \*/

public class ThreadInterruptTest **{**

    public static void main**(**String**[]** args**)** **throws** IOException  **{**

        ThreadInterruptTest test **=** **new** ThreadInterruptTest**();**

        MyThread thread **=** test**.new** MyThread**();**

        thread**.**start**();**

**try** **{**

            Thread**.**currentThread**().**sleep**(**2000**);**

**}** **catch** **(**InterruptedException e**)** **{**

**}**

        thread**.**interrupt**();**

**}**

    class MyThread **extends** Thread**{**

    private volatile boolean isStop **=** **false;**

    public void setStop**(**boolean stop**){**

**this.**isStop **=** stop**;**

**}**

        @Override

        public void run**()** **{**

            int i **=** 0**;**

**while(** i**<**Integer**.**MAX\_VALUE**){**//会一直运行直到变量i的值超出Integer.MAX\_VALUE。直接调用interrupt方法不能中断正在运行中的线程

           // while(!isInterrupted() && i<Integer.MAX\_VALUE){//若干个值之后，while循环就停止打印了。一般情况下不建议通过这种方式来中断线程

           //while(!isStop){//这个是最佳方法

                System**.**out**.**println**(**i**+**" while循环"**);**

                i**++;**

**}**

**}**

**}**

**}**

# REF

1.  源码解析

<http://m.myexception.cn/program/1033904.html>

2.[Java中join和yield的作用](http://blog.csdn.net/reboot123/article/details/22589621)图解

<http://blog.csdn.net/reboot123/article/details/22589621>

3.[java的守护线程与非守护线程](http://www.cnblogs.com/super-d2/p/3348183.html)

<http://www.cnblogs.com/super-d2/p/3348183.html>

4.关于Thread的一些实例

<http://www.cnblogs.com/dolphin0520/p/3920357.html>