**Java并发编程**

**在java中实现多线程有两种方式：**

1 实现Runnable接口(java.lang.Runnable)

2 继承Thread类(Thread实现了Runnable接口)

然后编写run()方法，run方法通常被写为无限循环的。这意味着除非有某个条件终止循环，否则就会一直循环下去。启动多线程并不是直接调用run方法，而是调用start()方法。

**线程的常用方法：**

yield() :线程让步，这个方法告诉线程调度器，可以将cpu切换给其他线程。但并不一定保证这个建议被采纳。

sleep(Long millis):这个方法是Thread类的静态方法，可以使当前线程进入睡眠状态millis毫秒。

setPriority():设置线程的优先级，优先级高的线程会比优先级低的线程更有机会先执行

但并不意味着优先级低的线程没有执行机会，也就是说优先级不会引起死

锁。在绝大多数时间里，所有线程应该以默认的优先级执行。视图操纵线程的优先级通常是一种错误。设置优先级应在run方法开头设置。在构造函数设置没有意义，因为此时执行器还未执行任务。JDK有10个优先级

setDaemon():设置后台线程。当所有非后台线程结束执行时，程序也就终止了，同时会终止所有后台线程。设置后台线程需在程序启动前设置。由后台线程派生出的子线程也是后台线程。后台线程不会执行finally块中的代码

isDaemon():返回布尔值表示当前线程是否是一个后台线程。

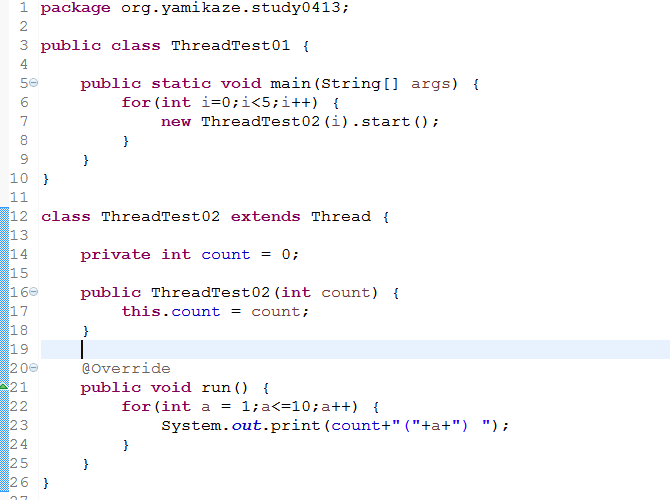
Thread.currentThread()获取当前线程。

getName():获取当前线程名字,为指定名字时返回为Thread-字符串+数字标识

join方法：有线程A，B，如果在线程B上调用A.join方法,那么线程B会等到A线程

执行结束才恢复执行。A的join方法可以被B调用interrupt方法打断。打断后B将不再等待A执行完再执行。调用interrupt方法将抛出异常。

**使用java.util.concurrent包中的执行器(Executor)来管理Thread对象。**

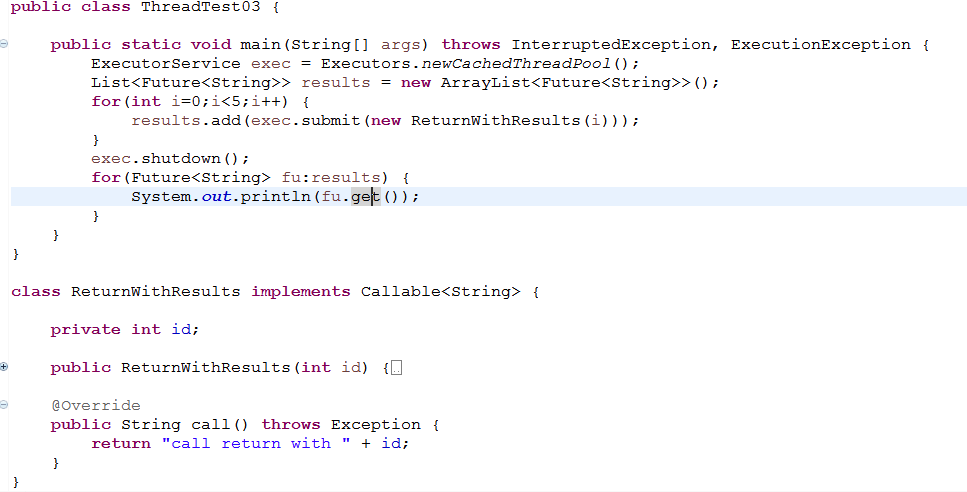
****

上图创建Thread对象启动线程为显示创建。将代码可以修改为以下



如果有大量线程，这些线程运行的任务会使用文件系统，可以使用SingleThreadExecutor来管理这些线程。因为这个执行器相当于线程数为1的FixedThreadExecutor。当向SingleThreadExecutor提交多个任务时，那么这些任务将会排队，这就保证了在同一时刻都只有唯一的任务在运行。

Runnable是执行工作的执行任务，不返回任何值。如果希望任务完成时返回一个值，可以实现Callable<V>接口(java.uti.concurrent包)，类型参数V表示从call方法中返回的值，而不是run方法返回的值。而且加入Executor中时不在使用execute方法，而是必须使用submit方法。Submit方法会返回一个Future<V>对象，可以使用Future的get方法取出V。如下图：



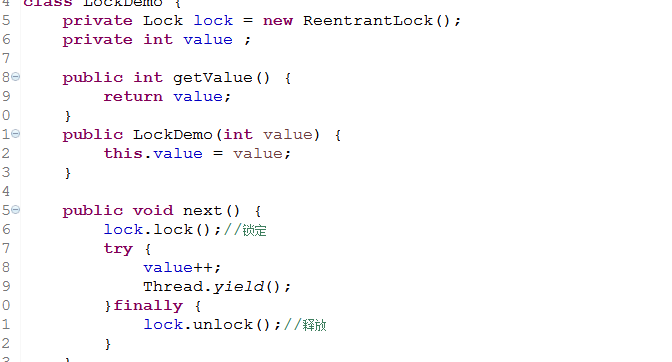
**处理线程异常：**由于线程的本质特性，从线程run方法中逃出的异常，会向外扩散到控制台。处理异常可以在run方法中使用try-catch块来处理所有异常。在JavaSE5以前，使用线程组来处理，但现在可以使用执行器（Executor）来解决这个问题。需要创建绑定了线程工厂的线程池，然后再工厂内部为线程设置捕获逃出run方法的异常处理器。Thread.UncaughtExceptionHandler会为每个Thread对象附加一个异常处理器，异常处理器的uncaughtException方法会在线程因未捕获异常而临近死亡时调用。

****

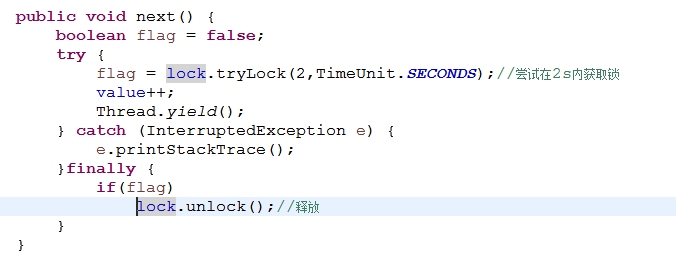
如果所有线程使用的都是同一个异常处理器，可以使用Thread.setDefalutUncaughtException 来设置一个静态域，如果线程发生错误时，如果线程没有私有的异常处理器，将会使用Thread设置的处理器处理。

**线程同步：**

1. 加锁：java是使用synchronized关键字在可能会发生冲突的代码块来进行加锁。方法或代码块被加上synchronized关键字时（代码块加上synchronized被称为同步控制块），同一时刻只能由一个任务来调用。Synchronized锁住的代码块或方法越多，那么程序执行的效率就越低。
2. 使用显示的Lock对象。Lock类存在于java.util.concurrent.locks包中。Lock对象必须显示的创建，锁定和释放。Lock方式的锁比较synchronized的优点是可以在发生异常时将系统维护在正确的状态。



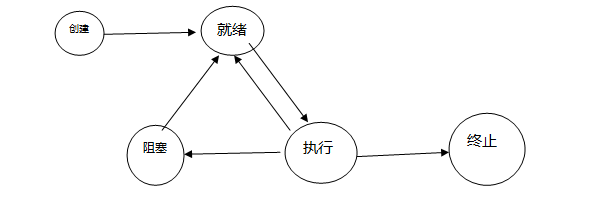
或者尝试一段时间获取锁，然后放弃或者尝试获取所且最终结果失败，就必须使用java.util.concurrent.locks.Lock，ReentranLock允许你尝试着获取且最终未获取锁



防止线程在共享资源上产生冲突：

1. 给共享资源加上volidate关键字修饰，解决了内存可见性问题。 但是会影响cpu的优化。
2. 根除对变量的共享。线程本地存储是一种自动化机制，可以为使用相同变量的不同线程都创建不同的存储，假设有5个线程对一个变量x，那么线程本地存储就会生成5个用于x的不同的存储块，创建和使用线程本地存储使用ThreadLocal来实现。

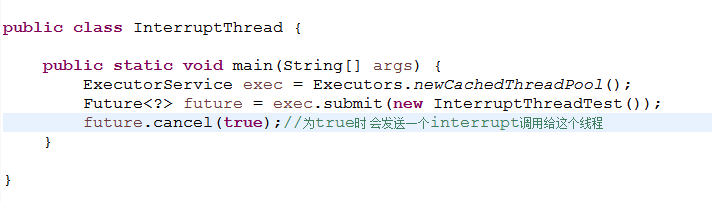
**线程的状态：**

1. 新建：线程被创建时会短暂的处于这个状态，此时已经分配了必须的资源，并执行了系统的初始化，之后调度器将这个线程转变为可运行或阻塞状态
2. 就绪：这种状态下，只要调度器将时间片分配给线程，线程就可以运行
3. 阻塞：线程能够运行，但是有某个条件阻止线程运行。当线程处于阻塞状态是，调度器不会分配线程任何cpu时间，直到线程进入就绪状态
4. 死亡：处于这个状态的线程是不可调度的，并且再也不会得到cpu时间分配。他是任务结束或者不再是运行的

线程在以下情况会进入阻塞状态：

1. 执行了sleep方法
2. 使用wait方法将线程挂起
3. 等待IO输入输出
4. 任务试图在某个对象上执行同步代码，但锁被另一个对象取得

打断阻塞状态可以调用interrupt方法，这个方法由必须要有Thread类的对象改为了Thread的静态方法。但推荐使用Executor来执行打断操作。Executor的shutdown方法执行时会发送一个interrupt调用给它启动的所有线程。如果想要对单个线程进行interrupt操作，Executor的submit方法会返回一个Futrue<?>泛型对象，Future对象的cancel方法参数为true时，就会在该线程上调用interrupt方法。



但是这个方式不能中断正在试图获取synchronized锁和试图执行IO的线程。对于这种问题可以采取关闭任务在其上发生阻塞的底层资源。可以通过Thread.interrupted方法或者isInterrupted方法查看线程是否被中断过，还可以清除中断状态，清除状态确保并发结构不会因为某个任务中断这个问题通知你两次。

**线程之间的协作：**

Object的wait方法可以使你等待某个条件发生变化，而改变这个条件将由另一个任务来改变。调用这个方法会将任务挂起，并且只有在notify或者notifyAll方法调用或者等待时间到期时，这个任务才会被唤醒。调用wait方法时会释放当前任务获得的锁，而sleep和yield方法不会释放锁。而且只能在同步控制方法或者同步控制块里调用wait等方法，否则在之外调用编译不会报错，但运行时会抛出IllegalMonitorStateException。当有多个任务在单个对象上处于wait状态时，调用notifyAll比notify更安全，但如果只有单个对象处于wait状态，那么调用notify相比调用notifyAll就是一种优化。此外，使用notify，所有wait状态中的任务必须等待相同的条件。notify只会唤醒众多等待同一个锁的任务，因为如果有多个任务等待不同的条件，使用notify就不会知道是否唤醒了恰当的任务。

NotifyAll方法时唤醒所有正在等待的任务，但并不意味着可以唤醒所有正在等待的任务。只能唤醒所有正在等待相同锁的任务。且只有当前线程对象拥有者或者锁对象才能调用notify和notifyAll方法。意味着一个线程对象唤醒另一个线程对象时，只有将该线程对象作为另一个线程的锁放在同步控制块里。因为同步方法的锁为当前线程对象本身。

还可以使用显示锁对象与java.util.concurrent包中的Condition类来完成挂起或唤醒操作。Condition对象可由Lock对象的newCondition方法得到。

await方法与wait方法一致，signal方法与notify方法一致，signalALl方法与notifyAll方法一致。但与使用notifyAll方法相比，signalAll方法更加安全

wait与notifyAll方法以一种比较低级的方式解决了任务互操作问题。但在许多情况下，可以使用同步队列来解决任务协作问题。同步队列在任何时刻都只允许一个任务插入或移除元素。Java.util.concurrent包中的LinkedBlockingQueue(无届队列)或者ArrayBlockingQueue(具有固定的尺寸)。在同步队列之前，可以使用管道进行输入输出。

管道类PipedWriter提供了实现。当一个线程向管道写入数据后，休眠一个随机数时间，另外一个线程通过Writer读取数据，如果没有数据，管道将自动阻塞。

**死锁：**某个任务在等待另一个任务，而后者又在等待别的任务，这样一直下去，直到这条链条上的任务又在等待第一个任务释放锁，这样没有哪个线程能够继续运行。即一个进程集合中的每个进程都在等待只能由此集合中的其他进程才能引发的事件，而无期限陷入僵持的局面被称为死锁。死锁有可能发生在你的程序里，而事先没有任何征兆。

当下列四个问题全部满足时就会发生死锁：

1. 互斥条件：任务中使用的资源至少有一个是不能够共享的。例如哲学家就餐中的筷子
2. 至少有一个任务它必须持有一个资源且正在等待另一个资源
3. 资源不能被任务抢占
4. 必须有循环等待

而要破坏死锁，只需破坏上述条件中的任意一个即可。

Lock锁与synchronized区别： Lock位于java.util.concurrent.lock包

1 Lock锁与synchronized都可以重入

2 Lock锁可以中断

3 Lock锁可限时，当在一定时间内未获取锁时，自动返回false

4 Lock锁时公平的。先来的线程能先得到锁，其他所不能保证这个条件。