**Java多线程**

***1.Thread***

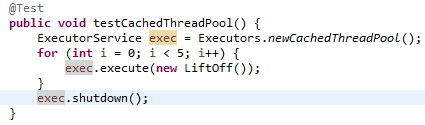
构造器只需要一个runnable对象，调用Thread对象的star()方法，里面会调用Runnable的run方法，最终执行的是Thread的Runnable方法

***2. Executor***

***2.1 CachedThreadPool***

java.util.concurrent包中的的Executor可以用来管理Thread对象。

利用这个工具我们可以不用显式的创建Thread对象。



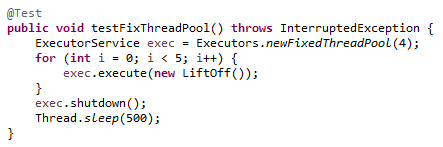
ExecutorServie利用Executors的静态方法创建线程池，线程池为每一个任务创建一个线程。

而shutdown()方法是阻止新的任务被提交到这个Executor,我试过，注释掉那行代码的话，线程还是能顺利执行的

***2.2 FixThreadPool***

FixThreadPool可以限制线程的数量，这样系统就不用为每个任务都创建线程而造成系统上的额外开销。

这样，系统在需要线程的时候直接从线程池上面获取。



从测试结果可以看见，第一个线程执行完了之后，第五个线程才可以拿到相应的线程去处理业务。

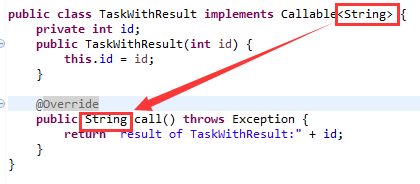
而且有一个重点，最后一个线程的线程名称跟最早结束的线程名称是一样的，所以从这里可以判断线程池确实限制了线程的大小

***2.3 SingleThreadExecutor***

SingleThreadExcecutor相当于线程池数量为1的FixThreadPool。如果有多个线程需要执行的话，线程相当于排队在等候执行。

***3. Callable***

Runnable是独立任务，并不会返回任何值，但是如果希望任务完成时能返回一个返回值，那么就应该用Callable



在TaskWithResult里面可以看出线程执行的逻辑在call方法里面，这个也是实现了Callable的接口方法。返回值跟实现接口的时候定义的泛型一致。

在TaskWirhResultTest里可以看出往ExecutorService里添加Callable的时候是用sumit的方式，而不是像Runnable那样直接用execut(new Runnable())这种方式

***4. Thread.yield()***

切换CPU执行时间片段，意思是本线程的主要逻辑已经做完了，可以将cpu从一个线程切换到另一个线程。

***5. Thread.currentThread().setPriority(int)***

线程可以设置优先级，但是并不是说，优先级低的就永远不执行，只是线程的执行频率较低。

有内置的三种级别:Thread.MIN\_PRIORITY, Thread.NORM\_PRIORITY, Thread.MAX\_PRIORITY

***6. daemon***

守护进程，也叫后台线程，

用法:

Thread thread = new Thread(new Runnable(……););

thread.setDaemon(true);//是守护进程

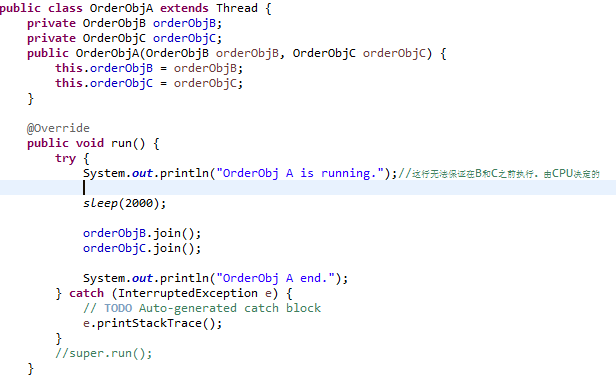
这个要在thread开启前设置

***7. join()***

如果A线程里面调用B线程实例的join()方法，那么A线程将会等待B线程执行完之后才恢复。

一个场景:A线程必须等B和C线程分别执行完之后再继续后续的执行。

这个很简单(ABC都分别继承Thread)，将B和C线程实例放到A实例中，在A的run方法里面分别对B和C的实例进行join()；B和C的线程在main方法中运行都无所谓。而且B和C是并行运行的.



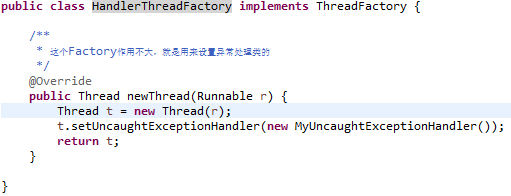
***8. 异常的捕获***

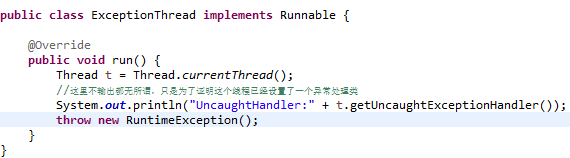
一般来说，线程的异常是不能被捕捉到的。

好比说你在Thread的run方法里面抛出一个异常throw new RuntimeException();，在线程start的时候try-catch是无法捕捉到的,就是说不会到catch里面去



看下一个程序:







我们可以看到Executors.newCachedThreadPool的时候，可以设置线程工厂ThreadFactory，ThreadFactory是一个接口。实现它，并且重写他的线程生成方法即可

在线程中设置异常处理类UncaughtExceptionHandler，同样，UncaughtExceptionHandler也是一个接口。做一个类实现它，拿我们就拥有一个异常处理类

另一种方式，可以全局设置:Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandler());

这样系统中所有的线程都交由这个异常类去处理异常。

这种方式也是try-catch不住的，只不过我们可以在自定义的异常处理类那里去处理这些个异常。

***9. sychronized***

如果一个任务进入了一个标记为synchronized方法里面，那么其他任务必须等这个任务完成之后才能进入

每一个特定的对象上都含有一把单一的锁(也称为监视器),如果在对象上调用其任意的synchronized方法，那么整个对象都会被加锁，这个对象上的其他synchronized方法只能等这个方法执行完之后再进行。

1. synchronized(this)那么对象中的其他synchronized方法不能被其他线程使用。
2. synchronized methodA 和 synchronized methodB 会互斥，也就是说同一个对象同一时间只能有一个method在运行
3. synchronized methodA 和其他非synchronized方法并不互斥。也就是说其他非synchronized方法能正常使用
4. 类中定义一个Object，synchronized这个Object，那么将缩小锁的范围，并不会对其他方法锁或者synchronized(this)互斥。反过来依然，方法和对象锁，并不能影响内部域Object的锁
5. 一个线程可以多次获得某个锁。

***10. Lock接口***

java.util.concurrent.Lock

它其中有一种实现方式Lock lock = ne ReentrantLock();

Lock需要显式的lock,还有unlock，一般unlock()都会放在finnally里面执行。

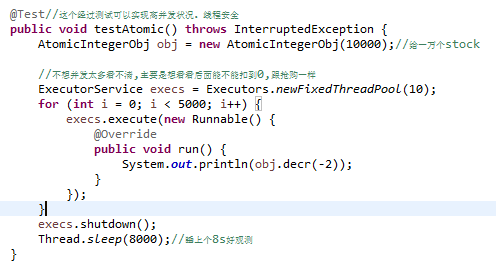
synchronized需要写的代码更少，但是lock有时候会挺灵活。因为synchronized一旦获取不到锁就会一直等待。但是Lock可以用lock.tryLock(2, TimeUnit.SECONDS)去尝试连接一段时间如果确实获取不到锁的话，那么就离开。关键点是在尝试获取锁等待的时间特性。

***11. volatile***

它能保证所有线程读取被volatile修饰的变量是最新的值，但是并不能解决并发问题，因为这个不是原子操作。有可能两个线程同时将值从工作内存固化到物理内存的时候发生冲突。

***12. util包的原子类工具***

AtomicInteger,AtomicLong,AtomicReference



***13. ThreadLocal***

ThreadLocal可以为竞争资源创建副本，比如同一个对象在使用ThreadLocal时都会存一份副本，保证线程之间资源不会竞争。每个线程操作的都是自己本地的对象。

***14. 线程的四种状态***

a) 新建(new):当线程被创建时，就会短暂出于这种状态，它有资格获取CPU时间.

b)就绪(Runnable):这种状态下，只要调度器将时间分配给线程，那么线程就可以运行

c)阻塞(Blocked):线程可以运行，但是某个条件阻止它运行了。此时不会分配任何CPU时间

d)死亡(Dead):是线程的最终状态，不会能再被调度

***15. interrupt(中断)***

Thread.interrupt()可以中断被阻塞的任务,但是不等于break,如果是死循环就没办法跳出。他只是干扰目前线程的阻塞

如果线程里面需要Thread.sleep(xxx)去等待某些资源时可以使用interupt方法,但是会抛出InterrruptedException,所以java虚拟机就是告诉你，可以捕获这个异常，并且释放相关资源。

一般情况下直接使用ExecutorService去中断，比如如下代码:

ExecutorService exec = Executors.newCachedThreadPool();

Future<?> future = exec.submit(new SleepBlocked());

future.cancel(true);

//exec.shutdownNow();

特别说明:睡眠就想上面那样是可以被中断的，但是IO和synchronized是不能被中断的，这一点记住就行，书上的例子已经验证。p696

***16. 代码解锁***

正因为synchronized方法是不可以被中断的，所以concurrent包提供的lock可以解决这个问题

Lock lock = new ReentrantLock();

用lock锁住需要的代码块(lock.lock())，其他代码块如果希望有可能获取这个锁就设置lock.lockInterruptibly.说明这个lock是可以被中断的

在上层直接使用thread.interrupt即可

***17. wait,notify,notifyAll***

这几个方法没有在线程里面，是直接在对象里面的。因为锁发生一般是在对象当中。

wait是让当前线程出于等待状态，它也可以像Thread.sleep(xxx)一样，睡眠一段时间。跟sleep不同的是，wait会释放当前锁，意思我当前的事情做完了，锁可以让出去了。

所以在本对象中，其他代码块里面之星notifyAll的时候就可以唤醒wait线程。

***18. Condition***

Java的concurrent包里面有一个显式的工具可以代替notify

Condition的产生:

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();

线程等待:condition.await();

唤醒wait中的线程: condition.signalAll();

***19. 线程队列***

可以使用队列实现有序的生产消费逻辑

接口是:BlockingQueue<MyRunnable> queue;

实现类有:LinkedBlockingQueue<MyRunnable>()【无边界队列】,ArrayBlockingQueue<MyRunnable>(8)【有边界队列，就是限制大小】,SynchronousQueue<MyRunnable>()【大小只有1】

具体用的时候直接new就可以了

放置runnable元素:queue.put(MyRunnable);//生产者

消费runnable元素:queue.take();//消费者

***20. 死锁***

死锁的概念是:一个任务在等待另一个任务，一直传递下去，最后的任务又依赖第一个任务，造成程序无法继续往下执行。

四个条件同时满足时，就会引发死锁。

1. 互斥条件。任务使用的资源里面有一个是不能共享的。
2. 任务持有资源，并且等待其他任务持有的资源。
3. 资源不能被任务抢占
4. 必须有循环等待

***21. java.util.concurrent.CountDownLatch***

大概了解下就可以了，

countDownLatch里面包含一个计数器，一开始的时候初始化一个具体的数字。可以将这个对象放入一堆线程里面去进行latch.countDown();

那么之前的线程(可以是很多个),在具体业务逻辑代码上实行latch.await();

等这个计数器完全countDown完之后，latch.await()下面的逻辑才会继续进行。