线程是什么就不多说了

什么时候需要多线程：

程序需要同时执行多个任务时，增强用户的体验

当程序一个操作处于等待时(如用户输入，文件读取操作等)，程序可以做其他事，最大化利用CPU资源

单核CPU每次只能运行一个进程，多核CPU可以并行运行多个进程/线程

进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位

栈：每个线程有自己的栈，栈保存的不是对象，保存的是变量和对象的引用

堆：一个程序只有一个堆，所有对象都在对上分配

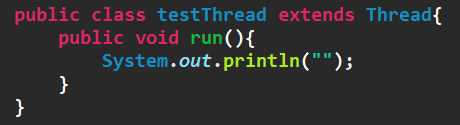
方法区：一个程序只有一个方法区，包含所有载入的类的字节码和静态变量

每创建一个线程都会开辟一个堆空间，线程就相当于是主程序的副本，在栈空间执行，需要数据时再到主存去取，然后将线程更新的数据刷新到主存

多线程的实现：

Thread类：

定义线程



run()方法的主体称为线程体，线程的所有操作都在run方法里执行

构造线程：

Thread()

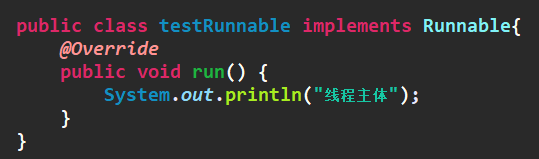
Thread(String threadName)：创建线程并为线程命名

Thread(Runnable target)

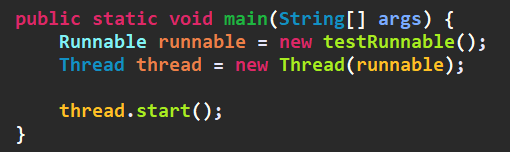
Thread(Runnable target,String threadName)

启动线程：Thread.start();

Runnable接口：



构造线程：

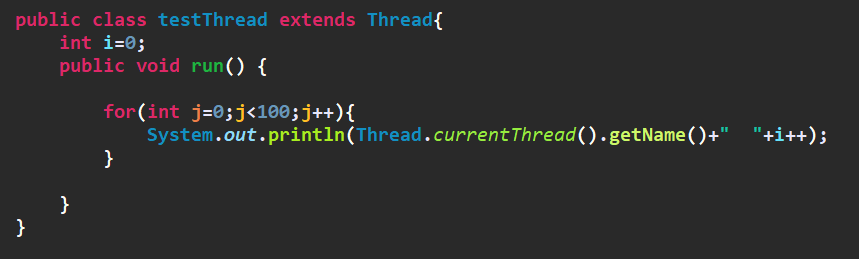


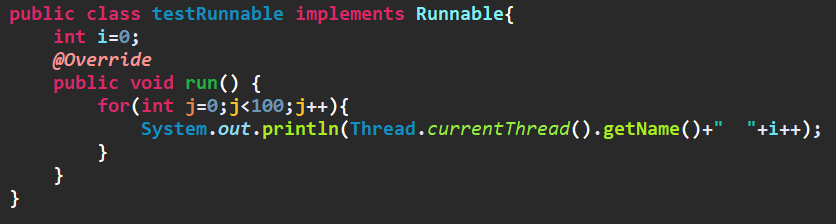
Runnable和Thread的区别：

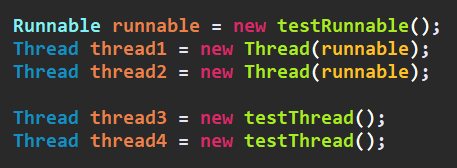
Runnable是个接口，只有一个run方法，类实现Runnable接口，说明此类是一个线程，线程启动时会去执行run方法中的方法体

Thread是个类继承了Runnable接口，并且自身还实现很多操作线程相关的方法，如果某类只是实现接口，那么只能说明此类是一个线程，而具体操作线程的方法还需要在Thread中实现，所以要将Runnable接口的实现类的对象放入到Thread中

线性的具体实现都是放在run方法里的





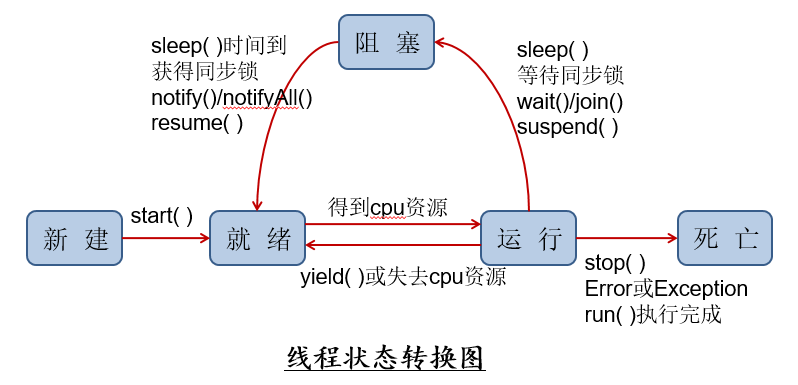


通过Runnable接口创建的线程thread1,thread2会共享Runnable接口所在的类的资源(共享testRunnable的i)，所以thread1和thread2同时对i进行自增操作(各自执行100次)，从0自增到199；而thread3和thread4会独享testThread中的资源，在创建线程的时候会复制一份资源，所以thread3对i进行100次的自增操作(0到99)，thread4对i进行100次的自增操作(0到99)，所以Runnable接口适用于处理同一份资源，尽量使用Runnable实现线程

线程分类：守护线程/用户线程

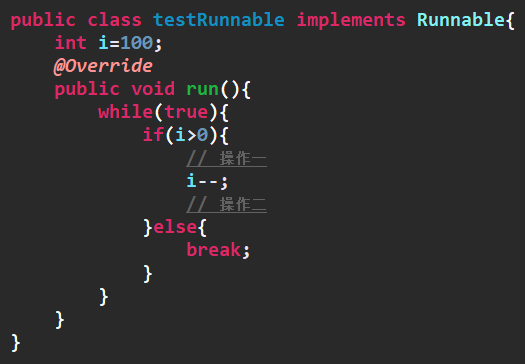
守护线程和用户线程没啥区别，唯一的区别在于守护线程是用来服务用户线程的，可以在线程start()启动前将线程设置为守护线程setDaemon(true)，当其他所有的用户线程执行完完毕后，守护线程才会结束，main方法也是一个线程，而且是守护线程，直到main方法里的所有线程执行完毕，main线程才结束，做常见对的守护线程就是JVM的gc，当所有线程执行完毕，不会产生垃圾时，gc线程才结束。

线程的五个状态：新建，就绪(start方法后，等待cpu时间片的调用)，运行(获得时间片)，阻塞(时间片用完或者手动阻塞，会到就绪状态，等待被调用)，死亡(线程执行完毕或者手动关闭)



线程同步：

当多个线程对同一资源进行操作时(Runnable)，当资源处理临界状态时候需要对注意线程之间处理的顺序，例如，线程A,B同时对Runnable中的资源i=100进行自减操作，但是i又要大于0，当资源i=1时，线程A进行操作时，i=1>0，所以执行自减操作，当线程A执行到操作一的时候，线程A的时间片用完(此时i任然等于1)，线程B获取到时间片，线程B执行，i=1>0，进行自减操作，当线程B执行完，i=0，然后再去执行线程A，再进行自减操作，i就变为-1了，这就是线程同步带来的问题。



解决办法：当多个线程操作共享数据的语句时，只能让一个线程执行完后，其他线程才能操作

Synchronized：

synchronized用于解决线程同步问题，使用方式：

synchronized(对象){

//需要同步的代码块

}

声明在方法前，表示此方法为同步方法

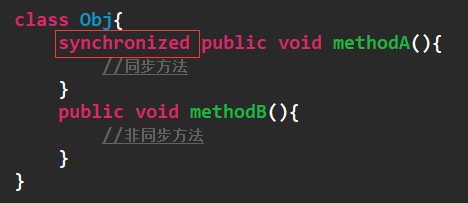
Synchronized public void test(){}

使用的几种情况：

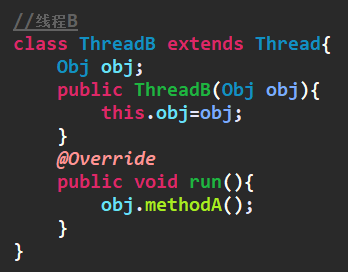
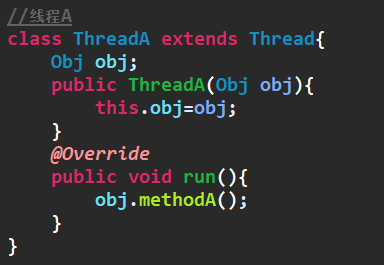
对象锁：synchronized修饰非静态方法，synchronized(this)，synchronized(非this对象)

synchronized修饰非静态方法：

Obj类：



线程A/B



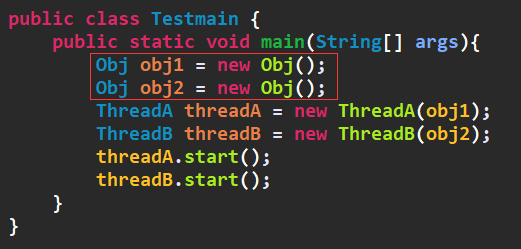
测试：



说明：Obj中有一个同步方法methodA和非同步方法methodB，线程A/B的run方法中执行Obj的同步方法methodA，在测试类实例化了一个Obj对象，并将此obj传入给线程A/B，此时的Obj其实就是一个锁，线程启动后，如果线程A获取先执行methodA，那么线程A就拥有了obj锁，当线程A时间片用完(methodA还未执行完)，线程B执行，但是obj锁已经被线程A占用，所以线程B只有等待，等待线程A执行完methodA方法后，是否obj锁，线程B才能执行methodA方法

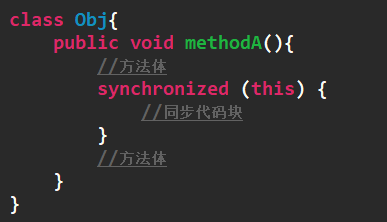
如果线程B执行的是非同步方法methodB，线程A获取锁后，执行methodA，线程B执行时，由于methodB是非同步方法，所以不需要锁都可以执行

如果Obj的methodB是同步方法，name线程B在执行methodB的时候，要判断线程B是否拥有锁，即时线程A执行的是methodA，只要线程执行的是同步方法，就要判断当前线程是否拥有锁，而此例中，线程A/B中都对锁Obj有使用权，只有去竞争



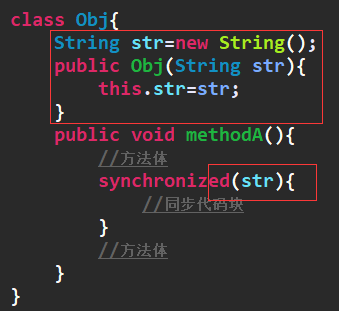
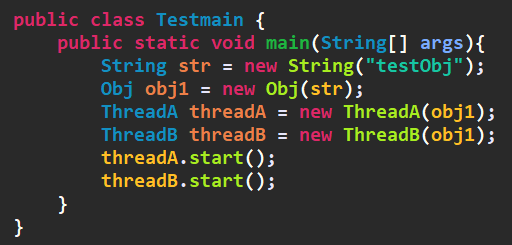
线程A/B拥有不同的锁，所以不存在竞争锁的情况，所以synchronized相对于没用(因为这里一个线程拥有一个锁，就不存在竞争的关系)

synchronized(this)：



当前修饰的是方法内的同步代码块，当执行到同步代码的时候会去判断当前线程是否拥有this锁，此时的this就是Obj的实例，使用方法和synchronized修饰方法一样，只是此在方法内部，修饰的代码块更准确

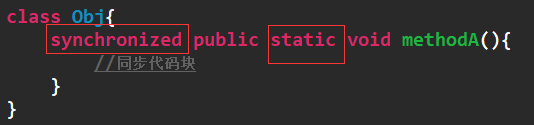
synchronized(非this对象)：

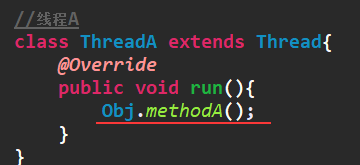
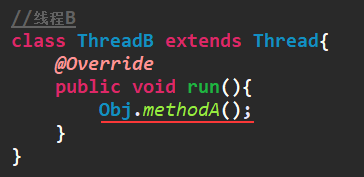
 

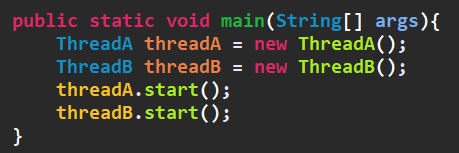
此时争夺的锁是str字符串

类锁：synchronized修饰静态方法，synchronized (类.class)

synchronized修饰静态方法：

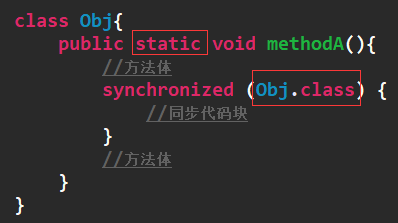




说明：此时线程争夺的是Obj类

synchronized (类.class)：



内置锁

在java中每个对象都拥有一个锁，而每个类也拥有一个锁

对象锁和类锁

A a=new A();对象a拥有一个锁(对象锁)，而A也拥有唯一的类锁，一个类只有一个锁(类锁)，而一个类可以实例化出多个对象，每个对象都拥有自己的锁，注意：对象锁不具有继承性，即A继承B，a实例化对象的锁，不拥有B的锁； 类锁和对象锁相互不干预

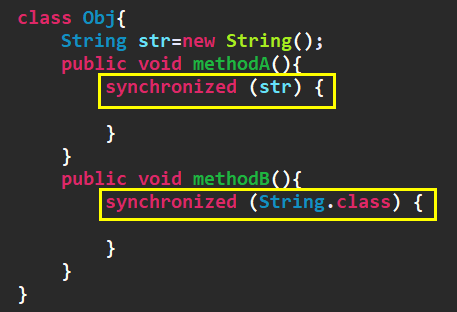
一般不使用String作为锁，因为字符串常量池的原因，会被放入到缓存，不能确保锁的唯一

synchronized修饰非静态方法，synchronized(this)，synchronized(非this对象)

synchronized修饰非静态方法修饰的是方法，锁住的范围比较，颗粒度划分不够，锁就是当前所在类的对象；synchronized(this) 对同步代码块进行上锁，缩小了上锁的范围，细化了颗粒度，this表示的是当前所在类的对象，它和synchronized修饰非静态方法的区别在于锁的范围；synchronized(非this对象)和synchronized(this)的区别在于拥有的锁不同

synchronized修饰静态方法，synchronized (类.class)

synchronized修饰静态方法 锁是当前类，修饰的方法，范围大，颗粒度不细；synchronized (类.class) 修饰的同步代码块，颗粒度更细，锁是类，但一般都是当前所在的类



运行过程：当线程A运行到synchronized时，会去获取当前的锁str(类锁/对象锁)，当线程B执行methodA的时候，先回去获取synchronized的锁，发现已经被获取了，而自己又没有对应的锁，所以线程B等待，但是如果线程B去执行methodB，会获取到String的类锁(此时线程A的methodA方法还未执行完，还没有去执行methodB)，线程B就可以执行methodB方法(注意，这里线程A/B操作的是同一个资源Obj，如果线程A/B操作的是两个不同Obj，methodA就不存在同步问题了，但是，methodB会存在，因为methodB的锁是类锁，唯一的，而对象锁str因为是在不同的Obj中，所以有两个锁)

synchronized可以加任何锁(str,this,Obj.class，这里只是加锁，对加什么锁，没有任何要求)，当第一个线程执行到synchronized时，获取到当前synchronized的锁，当其他线程执行synchronized时，会判断当前线程是否拥有对应的锁；类锁和对象锁是互不干预的，当线程A执行methodA时，只会获取到str锁，而不会获取到str对应的String类锁，所以线程B在执行methodB的时候可以获取到String的类锁。允许一个线程在拥有对象锁和其对应的类锁，但是在获取对象锁的时候不会获取对应的类锁。

线程常用方法：

Thread thread =new Thread();

thread.start()：启动线程

thread.getName()：获取线程名称

thread.getId()：获取线程的id

thread.setName(String threadName)：设置线程名称

Thread.currentThread()：获取当前线程

thread.setPriority(int newPriority)：设置线程的优先级(被cpu调用的概率)1,5,10

thread.getPriority()：获取优先级

thread.isAlive()：判断当前线程是否还存活

thread.setDaemon(true)：设置当前线程为守护线程

thread.yield()：线程让步，把执行机会让给优先级相同或更高的线程，释放cpu，变为就绪状态，和其他线程同时竞争cpu

thread.interrupt()：interrupt不会中断一个正在运行的线程，即时正在运行的线程调用了interrupt方法，线程任然继续执行，只是当线程处于阻塞状态(wait，join，sleep)时，调用interrupt，会抛出一个InterruptedException，使得提前结束阻塞状态。

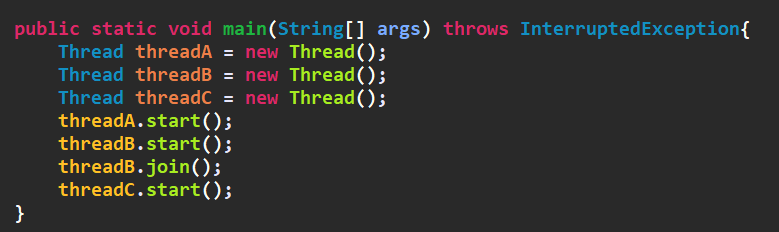
阻塞线程的3种方式：sleep,yield,join

thread.getState()：获取线程的状态

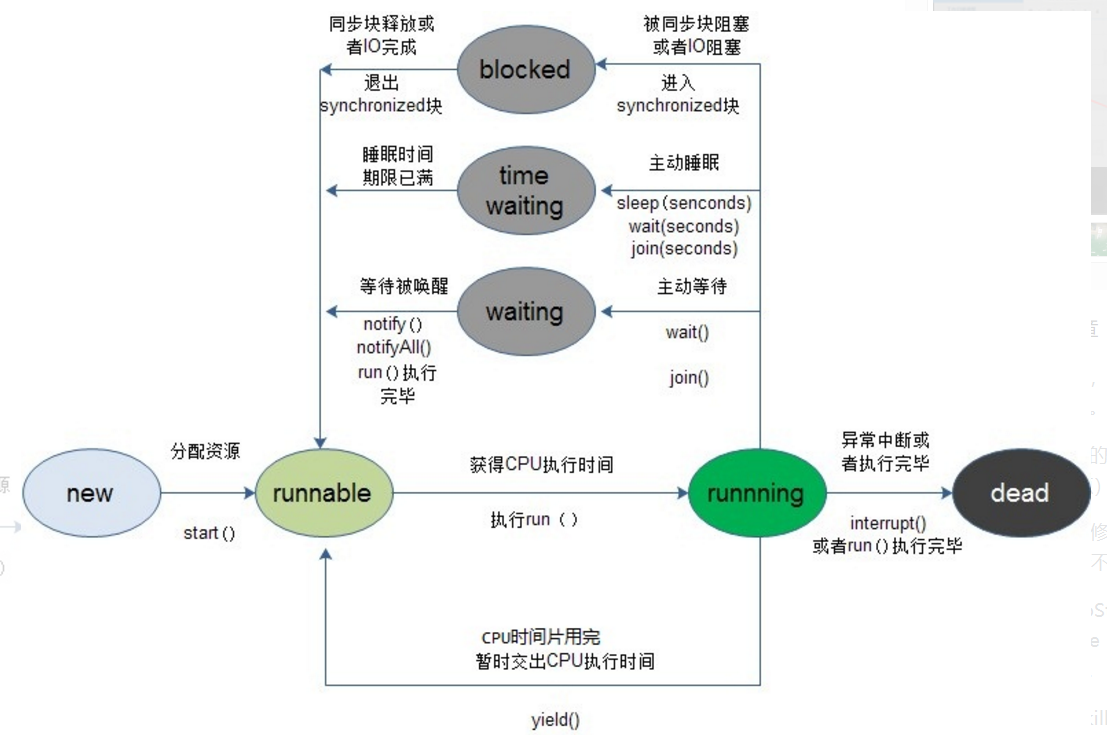
thread.sleep(long millis)：令当前线程在指定时间段内放弃cpu，使其他线程有机会执行，时间到后重排队，但是不会释放锁

thread.stop()：强制结束当前线程生命周期

thread.join()：



当执行threadB.join()时，程序中有线程threadA，threadB，main(守护线程)在执行，执行threadB.join()方法后，main线程将停止执行，线程threadA，threadB继续执行，直到线程threadB执行完(不管线程A执行完没)，main线程继续执行。



线程通信：

线程通信常用的方法：wait(),notify,notifyAll();

典型问题：生产者消费者问题



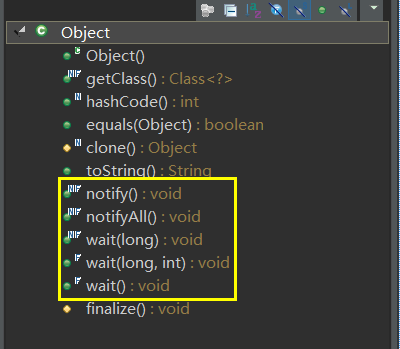
既然是线程通信，线程之间肯定要有共享部分，才能通信

Kitchen是生产者和消费者的共享部分，make是生产者的方法，eat是消费者的方法，当消费者和生产者线程同时start时，假设先执行消费者线程，在执行eat方法时，因为num为0，所以消费者进入wait，等待生产者的唤醒，处于wait状态的线程不会被cpu调度，消费者线程释放锁和cpu，生产者线程执行，在对num进行自增后，调用notifyAll方法唤醒进入等待状态的消费者，注意，这里只是将消费者唤醒，进入就绪状态，而不是运行状态，此时生产者和消费者都处于就绪状态等待CPU的调度

wait, notify和锁的关系：线程的同步、等待、唤醒都与对象锁是密不可分的

线程通信为什么需要锁：就是为了确保通信的线程之间能同步运行，但是在线程通信中的锁必须使用对象锁，不能使用类锁，为啥，下面会讲

wait,notify的方法都是Object的，而不是线程的，当调用wait方法时obj.wait()，当前线程进入等待状态，等待其它线程调用obj的notify方法，当线程调用obj.notifyAll()会唤醒所有等待obj的线程，注意这里的obj必须是synchronized的对象锁，对象锁可以是任意对象，但是调用wait,notify的必须是对象锁，否则会抛IllegalMonitorStateException，因为当前线要调用wait方法，必须是当前对象的持有者，而线程持有的对象就是synchronized中的对象，这里的wait方法实际上是当前线程调用的



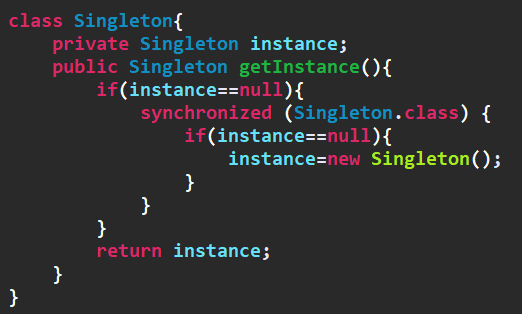
线程同步和线程通信的区别：

线程通信是指定多个线程之间相互指向，相互唤醒其他线程，而线程同步不能唤醒其他线程，只能等cpu来调度，而且处于wait状态的线程不会被cpu调度

wait()：令当前线程挂起并放弃cpu，释放同步资源

notify()：唤醒正在排队等候同步资源优先级最高的线程

notifyAll()：唤醒正在排队等候资源的所有线程



释放锁的操作：

当前线程的同步方法、同步代码块执行结束

当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。

当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束

当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

不会释放锁的操作：

线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行

线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。

应尽量避免使用suspend()和resume()来控制线程

notify不会释放锁，只会唤醒当前等待的线程，知道执行完synchronized所处的代码块(但是一般notify后，就执行完synchronized了)，wait会释放锁

Lock：

Lock和synchronized一样，都是用来实现线程同步，Lock是一个类，而synchronized是关键字，基于JVM

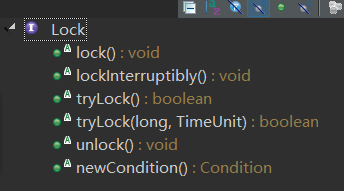
synchronized的缺陷：

当一个线程获取到对应的锁并执行时，其他线程便只能一直等待(当前时间片内不能做其他事，浪费资源)，直到释放锁

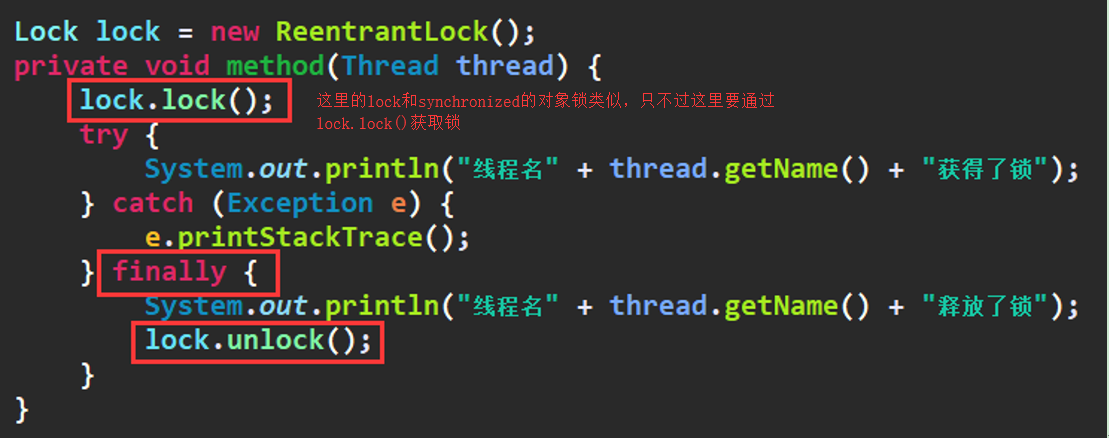
当读-读操作时，synchronized不能实现，当一个线程在执行读操作时，其他线程只能等待

当资源竞争不激烈时，synchronized的性能好，当竞争激烈时Lock好

Lock

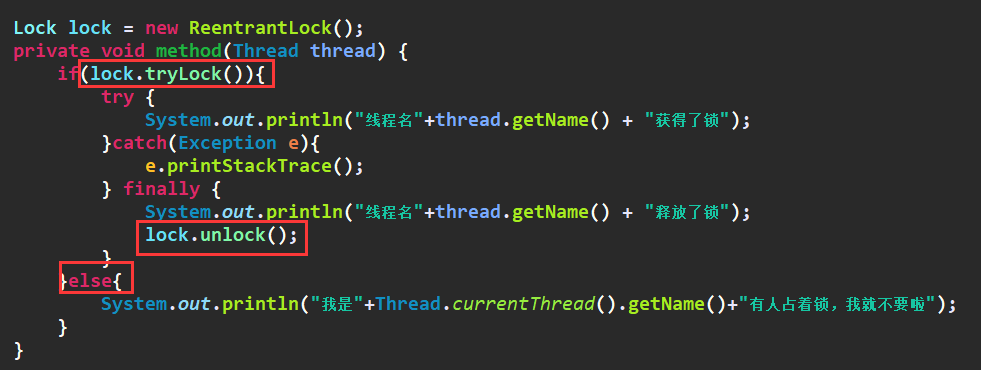


lock，tryLock，tryLock(long,TimeUnit)，lockInterruptibly都是用来获取锁的，unLock用户释放锁，在Lock中，锁必须手动释放(synchronized不用手动释放，执行完synchronized代码块自动释放)



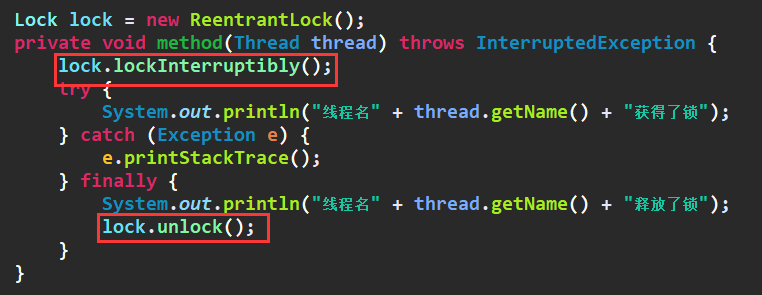
在使用Lock获取锁的时候，必须在finally中释放锁

通过lock获取锁，其他线程只能等待当前线程释放锁(这里和synchronized差不多)并且Lock也会判断处理的资源是否是同一份



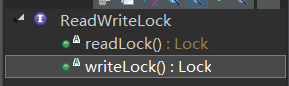
tryLock会去试图获取锁，如果能获取返回true，不能获取就执行其他的，不要等待

tryLock(long time, TimeUnit unit)和tryLock类似，在一定时间内如果不能获取锁返回false，可用于解决死锁问题(线程A/B在执行的时候同时需要资源C/D，线程A获取C，线程B获取D，那么线程A/B将会死锁等待对方释放锁，通过tryLock(time)，在指定时间内没获取到锁，放弃获取锁，并且可以释放出当前的资源，解决死锁，典型问题哲学家进餐)

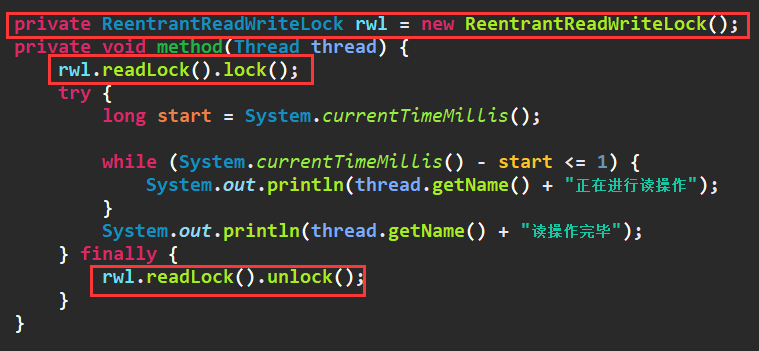


当线程获取这个方法时，如果线程正在等待获取锁，那么当前线程直接中断(不等时间片用完)，直接释放cpu，然后cpu重新调度(这解决了synchronized的缺陷①)，如果获得则继续执行啊，注意，这里lock.lockInterruptibly不能要放在try里面，因为如果线程未获取到锁，lockInterruptibly会执行一个interrupt，执行此方法会抛出一个异常，如果放在try里面，抛出异常后会执行finally方法，会释放锁，会出错，所以要将lock.lockInterruptibly放在try外面

ReadWriteLock：一个用来获取读锁，一个用来获取写锁，分成两个锁分配给线程



ReadWriteLock 的实现类ReentrantReadWriteLock

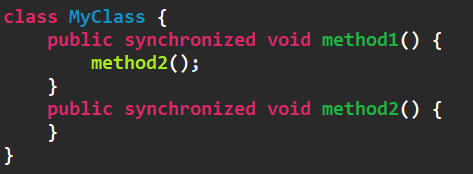


这样多个线程就可以同时进行当前代码块，当然在设计的时候，这段代码块必须设计为读操作，不能设计为写操作 (解决了synchronized的缺陷②)

当一个线程获取到ReentrantReadWriteLock的读锁后，其他线程在其他方法中获取写锁时，必须等读锁的线程释放后才能获取到写锁，同样写锁被占用，要等写锁释放后才能获取读锁

锁的几个概念：

可重入锁



当线程执行method1的时候获取到this对象锁，在method1中执行method2的时候，method2又是一个synchronized，此时就不用在获取锁了，因为当前锁就是this(但是如果是执行的是其他对象的synchronized方法，仍然要获取锁)，Lock也具有可重入锁，只要是同一个锁

可中断锁

线程A在执行锁中代码，线程B在等待获取锁，B就不想获取了，B可以中断自己，会到就绪状态，就是lockInterruptibly()

而synchronized不具有

公平锁

即尽量以请求锁的顺序获取锁，当多个线程在等待锁的时，当锁释放时，让等待时间最久的线程获取锁，非公平锁即获取顺序无序，可能会导致某个线程永远获取不到锁

synchronized就是非公平锁，而ReentrantLock和ReentrantReadWriteLock默认情况下也是非公平锁，但是可以设置为公平锁，ReentrantLock lock=new ReentrantLock(true);公平锁

读写锁

Synchronized称为内置锁， Lock称为显示锁

Lock线程通信

wait和notify只能被同步监听锁调用，否则会抛IllegalMonitorStateException，但是Lock没有同步锁，所以不能使用wait和notify，Lock的线程通信，采用Condition接口中的await,signal,signalAll方法取代Object中的wait，notify，notifyAll方法

ReentrantLock常用方法

lock.getQueueLength()：获取等待锁释放的线程数

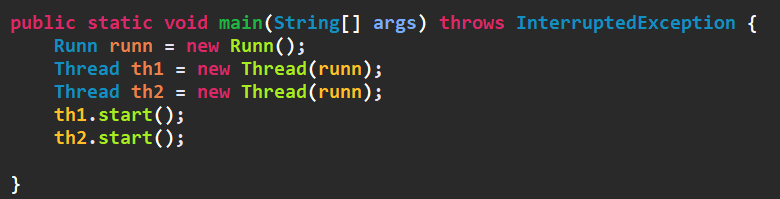
lock.hasQueuedThread(Thread thread)：判断指定线程是否等待该锁释放

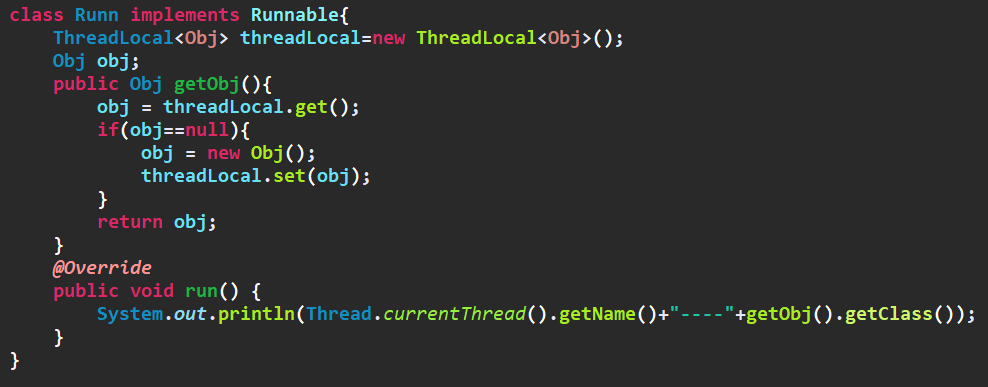
lock.hasQueuedThreads()：判断判断是否有线程在等待释放锁

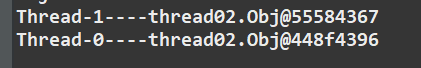
lock.isHeldByCurrentThread()：判断当前线程是否拥有当前锁

lock.isLocked()：判断一个锁释放被一个线程持有

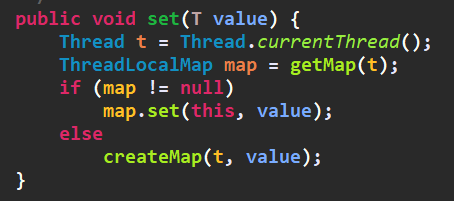
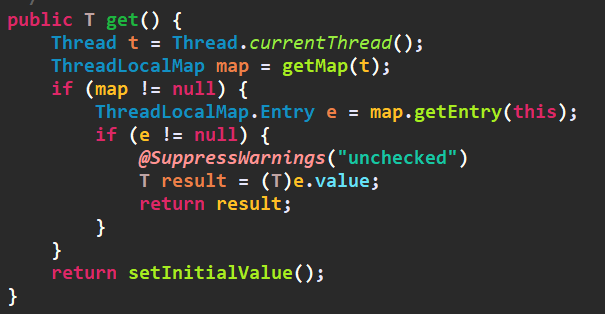
ThreadLocal：







ThreadLocal的get方法表示从当前线程获取对应的Obj，一个ThreadLocal中只存放了一个Obj，set方法表示将创建的Obj存放到ThreadLocal中，这样实现了每个线程都拥有自己的Obj，并且在线程任何地方都可以通过get方法获取到，这里的getObj方法不需要同步，因为获取get和set都是在自己线程中进行的，ThreadLocal的作用就是确保每个线程拥有自己独享的变量，并且在线程执行时能够获取到，ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

当执行set方法时，会去获取当前线程，然后从当前线程中获取ThreadLocalMap(在Thread类中有一个ThreadLocalMap)，然后将值放入到ThreadLocalMap，实际还是将值放入到当前线程的，保存在ThreadLocalMap中

同样get方法也是，先获取当前线程，然后获取当前线程的ThreadLocalMap，从中获取到，Obj

注意这里是以this(ThreadLocal)为键，保存的Obj为value，这是因为一个线程中可以有多个ThreadLocal，而实际保存value的是线程中的ThreadLocalMap，因此只能通过ThreadLocal作为键，并且一个ThreadLocal中只能保存一个副本，但是，因为副本会一直跟随线程直到死亡，所以对性能有一定的影响，合理使用

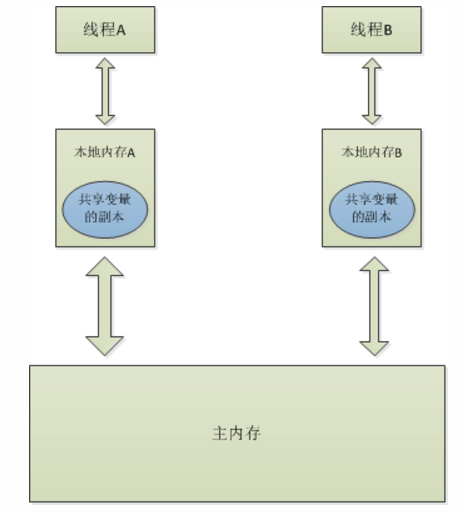
ThreadLocal<Obj> threadLocal=new ThreadLocal<Obj>();

ThreadLocal<Obj1> threadLocal1=new ThreadLocal<Obj1>();

Volatile：并发程序正确地执行，必须要保证原子性、可见性以及有序性

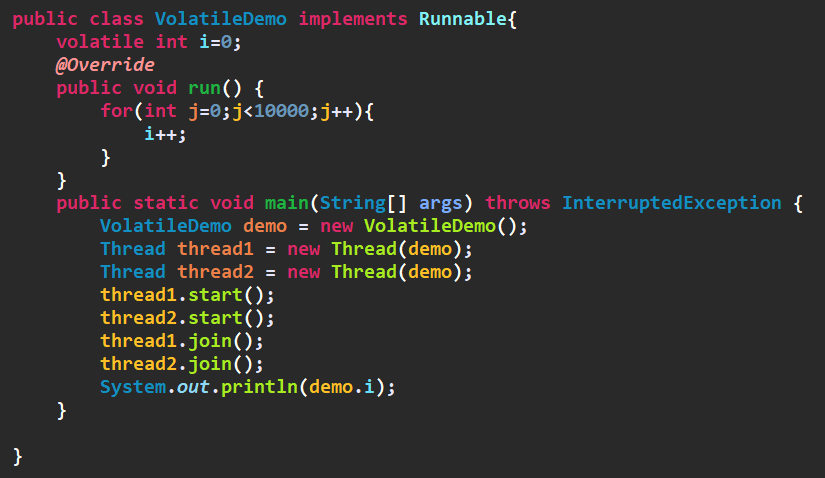
可见性：计算机会为每个正在执行的程序分配内存空间，而程序的线程在执行时，程序会为每个线程分配空间(分配的是程序的内存空间)，每个线程的内存空间分为工作内存(即单独分配给某个线程的，存储局部变量等)和共享变量(即从程序的主存中复制的线程之间共享的变量，这样做的目的是减少线程和主存的通信频率，提高效率)，可见性是指线程访问的变量是否是最新值，局部变量不存在可见性，共享内存才会出现可见性(当线程A修改共享变量后，但是此时的共享变量是从主存中的副本，线程B再去读取共享变量，注意此时读取的是主存的副本，A修改后，还未放入到主存，B读取，读取到的还是原来副本中的值，不是最新的)，现在就需要将A修改的值直接刷新到主存，而B直接从主存中获取，不到副本中获取，可以在共享变量使用volatile关键字(volatile int num;)，对于使用volatile关键字的属性，线程获取时直接到主存中获取，更新后直接刷新到主存中

volatile和synchronized实现可见性的区别：volatile是使得每次读取变量直接从主存中读取，写入直接写入到主存中，而synchronized是通过上锁的方式实现，当一个线程A在执行时，其他线程无法执行，线程A执行完后，将结果从缓存中刷新到主存



在 Java 中 volatile、synchronized 和 final 实现可见性。

原子性：即一个操作或多个操作要么全部执行，要么全部不执行，在java中x=10是原子操作，y=x不是原子操作，因为先要获取x的执行再将值赋值给y，这是两个步骤，单独的读取和赋值时原子的，但是加在一起就不一定了



最终的输出结果<=20000(20000的概率小)，按理来说使用看volatile具有可见性，为啥不等于20000，因为i++不是原子操作，读取，赋值两个步骤，即当线程A读取到i=0时，还未进行赋值操作(即此时i=0，不会将i=0放入主存，因为i的结果都没有改变)，线程B执行，从主存获取到i=0，进行i++操作，执行完一次循环后，将结果放回到主存(i=1)，然后线程A继续执行，对i=0进行i++，执行完循环后，将结果i=1放入到主存，这样循环两次，i只加了1，因为线程A和B可以同时操作i++操作，如果对i++进行上锁，那么就为20000(还有种情况：当线程A获取到i=1后，其他线程执行了很多次将i变为10，而线程A将结果更新到主存，结果又变为2了，所以最终结果会小于20000)，如果循环里的代码是原子操作就不会出现这种情况，每执行一次，都会刷新到主存

在java中只有简单的读取和赋值是原子操作，当范围扩大时，为了确保原子操作，可以通过使用synchronized和Lock

volatile不具有原子性，但是在写操作时，可以转为为原子操作(比如实例化对象的时候，对象创建好后才刷新到主存中)

java内存模型只保证基本的读取和赋值操作是原子性

有序性：即程序的执行顺序是按照代码的先后顺序执行

int a=10; //语句1

int r=2; //语句2

a=a+3; //语句3

r=a\*a; //语句4

程序在执行时这段代码时，可能会发生指令重排序(即执行顺可能是语句2，语句1)，但是执行结果和代码的顺序是一样的(即语句3坑在语句1之前，因为他们之间存在依赖性，也不能语句4在语句3之前)，2,1,3,4的顺序对执行结果不会有任何影响

指令重排序对单线程下没任何问题，但是在多线程下就会有影响

可以通过volatile保证一定的有序性，通过synchronized，lock执行同步代码块保证有序性，java模型具有一些先天的有序性，叫happen-before原则(java内存模型具有一些先天的有序性，即不需要任何手段就可以保证有序)

①程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作(但是还是会发生指令重排序，在多线程中不能正确执行)

②锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作

③volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作(两个线程，一个写，一个读，写先；将写操作视为原子操作)

④传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

⑤线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

⑥线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

⑦线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

⑧对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

前4条重要，后4条显而易见

volatile实现有序性

x = 2;        //语句1

y = 0;        //语句2

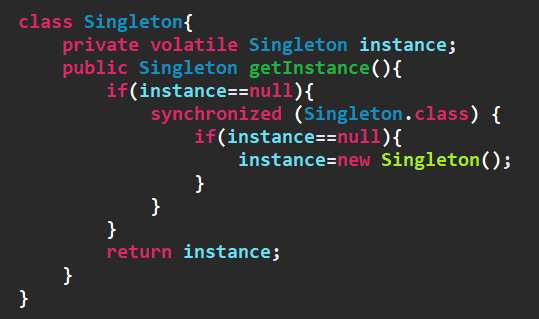
volatile flag = true;  //语句3

x = 4;         //语句4

y = -1;       //语句5

在进行指令重排序的时候，不会将语句3放到语句1,2前，不会将语句3放到语句4,5后，即语句1,2必须在语句3执行前执行，在语句4,5必须在语句3执行之后执行，1,2的顺序不管，4,5的顺序不管

语句1,2的执行结果对语句3,4是可见的



单例模式中，instance=new Singleton()；语句并不是原子性的，jvm在创建对象时做3个事情1，给instance分配内存，2，调用Singleton的构造函数实例化成员，3，将instance对象指向分配的内存空间(执行完这步 instance 就为非 null 了,指向了内存空间)；但是因为存在指令重排序，执行顺序可能不是1,2,3而是1,3,2，如果线程A执行到3后，被线程B抢占(线程A将结果刷新到内存)，线程B获取到的instance还是一个未完全实例化的instance，执行时就会报错，所以需要让new Singleton这个操作具有原子性，可以使用volatile修饰instance(instance的写入操作将会变成一个原子操作，没有初始化完，就不会被刷新到主存中)，这里应该是个特例记住即可(这里应该体现的是volatile的有序性)

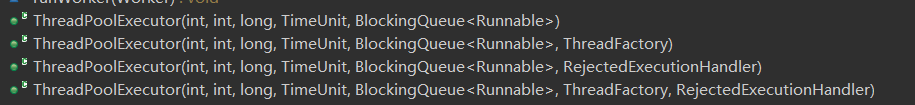
Volatile实现了有序性和可见性

当线程切换后，如果当前属性发生了改变则将属性刷新到主存或缓存，没有改变的下次时间片回来后继续执行

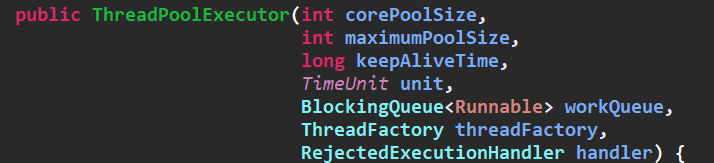
volatile在读性能消耗和普通变量几乎相同，但是写操作慢，因为为了确保有序性

线程池：

线程的创建和销毁都会占用大量资源，所以采用线程池来获取线程



ThreadPoolExecutor是线程池的一个核心类，提供了4个构造器，前三个构造器实际最终都是调用第四个构造器



corePoolSize：核心池大小，创建线程池后，默认线程池中没有任何线程，当有任务来了的时候就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会将到达的线程存放到缓存队列中(指定线程池最大处理量)

maximumPoolSize：线程池最大线程数，即线程池中最多能创建多少个线程

keepAliveTime：线程没有任务执行时在保持多久的时间会终止，默认情况下只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，但是如果调用了allowCoreThreadTimeOut(true)时，在线程池中的线程数不大于corePoolSize时，keepAliveTime也会起作用

unit：keepAliveTime的时间单位

workQueue：阻塞队列，储存等待执行的线程(即当线程池中的线程数大于corePoolSize时)

ArrayBlockingQueue/PriorityBlockingQueue较少，LinkedBlockingQueue较多

threadFactory：线程工厂，主要用来创建线程(可以指定，也可以采用默认的)

handler：表示当拒绝处理任务时的策略(线程数达到maximumPoolSize时，有新线程请求将拒绝)

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

线程池在可以放很多线程(maximumPoolSize)但是线程同时处理的线程数是有限的(corePoolSize)，不能处理的线程存放到缓存队列中(workQueue)

可以通过setCorePoolSize()和setMaximumPoolSize()对线程池进行动态调整

ThreadPoolExecutor的继承关系



ThreadPoolExecutor比较重要的几个方法

execute()：核心方法，通过此方法可以向线程池提交一个任务，交由线程池执行(执行一个Runnable任务)

submit()：也是用来向线程池提交任务额，但是和execute不同，它能够返回任务执行的结果

shutdown()和shutdownNow()是用来关闭线程池的。

表示线程池状态

volatile int runState; //当前线程池的状态，volatile变量用来保证线程之间的可见性

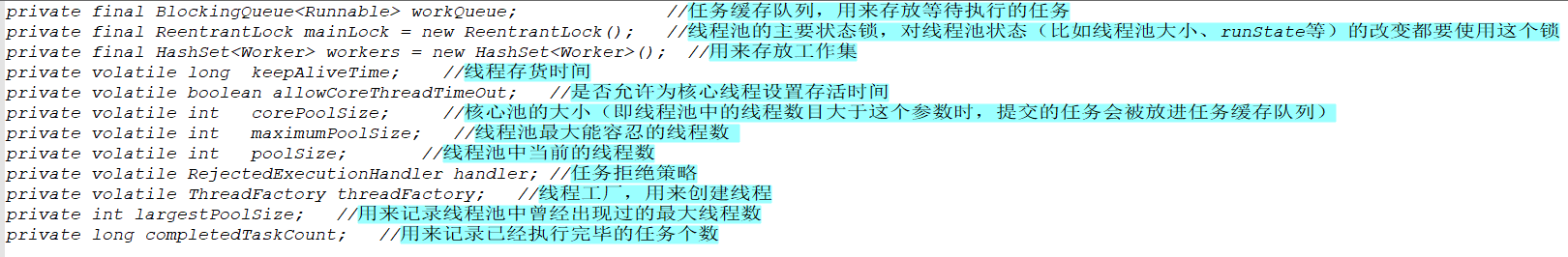
static final int RUNNING    = 0; //创建线程池后，初始时

static final int SHUTDOWN   = 1; //调用shutdown方法后，线程池不接受新的任务，但是让所有等待的线程执行完任务

static final int STOP       = 2; //调用shutdownNow方法后，线程池不接受新的任务，并且中断所有正在执行的任务

static final int TERMINATED = 3; //调用shutdown/shutdownNow方法后，等待所有线程结束(包含缓存中)后，线程池的状态

ThreadPoolExecutor的一些成员变量



线程池初始化：

默认情况下，创建线程池后，池中没有线程，需要任务提交后才会创建线程，实际可以在创建好线程池后就立即创建线程

prestartCoreThread()：初始化一个核心线程；

prestartAllCoreThreads()：初始化所有核心线程

任务缓存队列：workQueue当线程池中请求的线程大于corePoolSize时，新请求的线程会被放入到缓存队列中

ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

任务拒绝策略：当线程池中的数目达到maximumPoolSize时，新请求的线程将会被拒绝

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

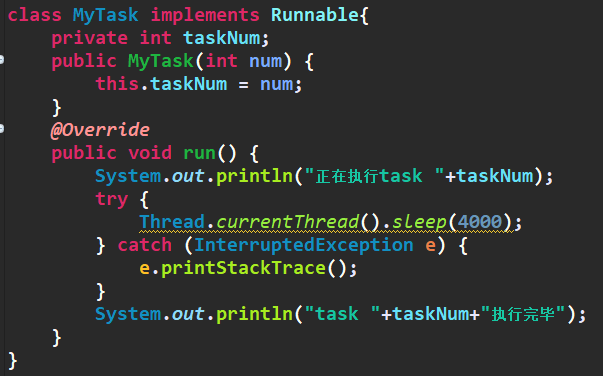
线程池的关闭：当线程池不使用时，如果不关闭，GC是不会回收的，这样就会导致内存泄漏

shutdown：不会立即终止，等待所有线程执行完毕(包含任务缓存队列里的)才关闭

shutdownNow：立即终止，包含正在执行的任务

设置allowCoreThreadTimeOut为true，在过期时间后，核心线程会被释放，则线程池被GC回收

示例





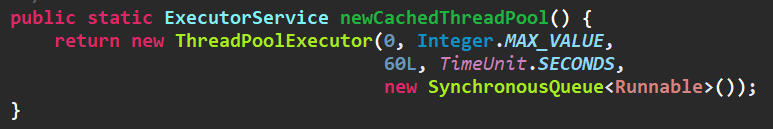
这里只创建了Runnable，并未创建Thread，Thread的创建和运行都在线程池内部进行

但是一般不使用ThreadPoolExecutor创建线程池，常用线程池

ExecutorService pool = Executors.newCachedThreadPool ();

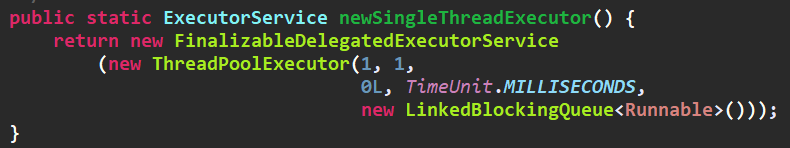
Executors.newCachedThreadPool();        //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

线程数无限制/有空闲线程则复用空闲线程，若无空闲线程则新建线程/减少频繁创建/销毁线程，减少系统开销



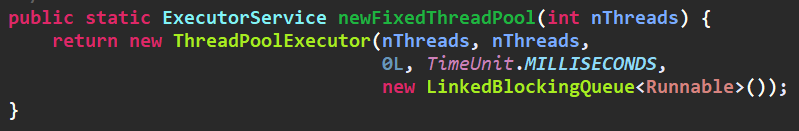
Executors.newSingleThreadExecutor();   //创建容量为1的缓冲池

有且仅有一个工作线程执行任务/所有任务按照指定顺序执行，即遵循队列的入队出队规则



Executors.newFixedThreadPool(int);    //创建固定容量大小的缓冲池

可控制线程最大并发数（同时执行的线程数）/超出的线程会在队列中等待

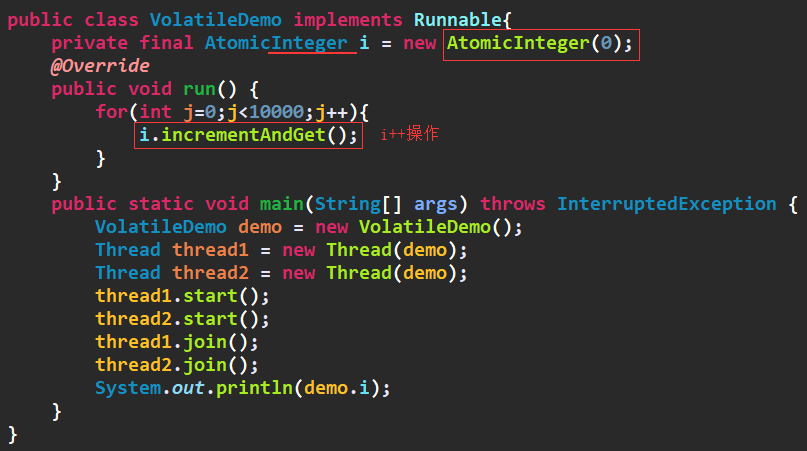


为什么要创建线程池

线程的创建和销毁都很销毁内存/如果一个程序创建过多的线程会导致内存占用大，影响程序性能，可通过线程池来限制程序拥有的线程数

Atomic

前面在多个线程对i++进行操作时，输出结果不一致，是因为没有i++不是原子操作，可以通过java.util.concurrent.atomic.\*下的类进行原子操作(也可以通过synchronized加锁)



最终输出结果为20000，因为i.incrementAndGet();进行的是原子操作

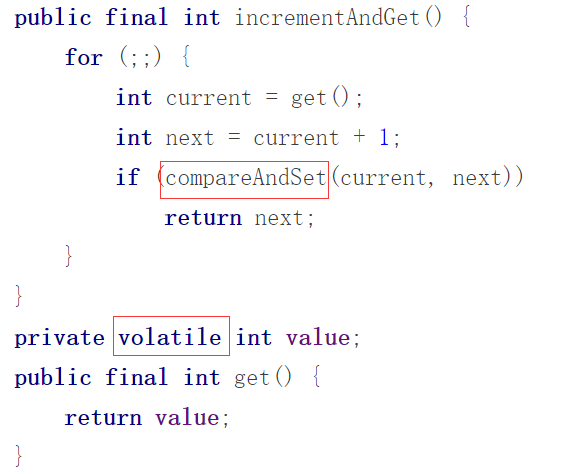
实现原理：

先了解一下CAS机制

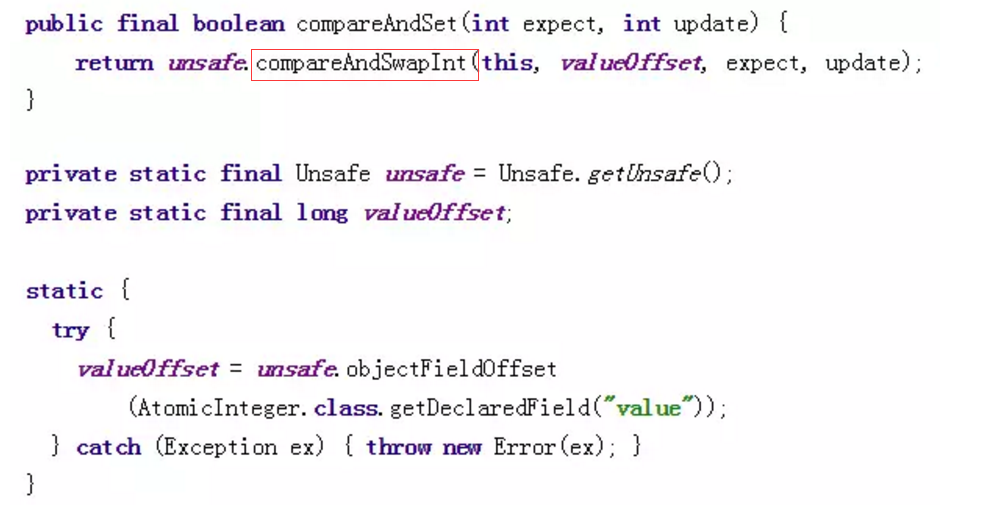
CAS(compareAndSet)是CPU的一个指令，CAS指令当中使用了3个基本的操作数，内存地址V，旧的预期值A，要修改的新值B，当更新一个变量的时候，只有当前变量的预期值A和内存地址V中的实际值相同，才会将内存地址V对应的值更新为B

在内存地址V中对应的值为10，当线程A想要将变量的值变为11，对于线程A来说，旧的预期值A为10，要修改的新值B=11

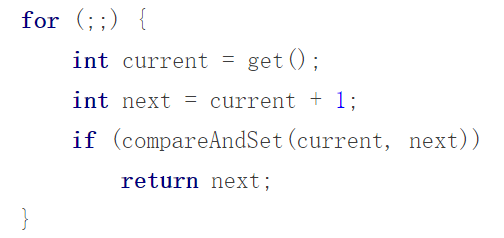
当线程A要执行CAS指令时，线程B也要对V的值进行操作，并且线程B先于线程A值，将V对应的值修改为11，然后线程A继续执行，当前线程A的旧的预期值为10，旧的预期值和实际V对应的值进行比较，如果相等，则把线程A把V对应的值修改为线程A对应的新值B，但是此时V对应的值为11而线程A的旧的预期值为10，不相等，所以CAS指令执行失败(CAS是一条硬件的指令，是一次就必须执行完毕的)，线程A对V的值修改失败后，线程A重写获取当前V对应的值作为旧的预期值(这里预期的意思就是获取到V的值后，有可能其他线程在线性A执行CAS指令时对V对应的值进行修改，所以叫预期)，然后再重新执行CAS指令，直到线程A执行成功，这个重新尝试的过程叫自旋(就是循环)



而在AtomicInteger的incrementAndGet中，先通过get从主存中获取当前值，进行自增后，循环执行compareAndSet方法，而这里执行的compareAndSet实际上就是CAS指令，这里的循环操作就只自旋



Java不像C语言那样可以直接访问操作系统底层，但是java可以通过Unsafe执行硬件的操作，compareAndSwapInt实际上执行的就是CAS指令，valueOffset就是CAS的V，expect就是旧的预期值A(volatile修饰，直接从内存中获取)，update就是更新值B，执行完后，返回一个boolean值，



True返回更新的结果，false继续循环，从主存在获取当前值(volatile修饰的)，然后对新值进行+1，然后继续执行CAS指令(自旋)

CAS和乐观的理念差不多，synchronized就是悲观锁

CAS循环时间长开销大

ABA问题

CAS只能保证变量是原子操作，要保证整个代码块是原子操作还得用锁(synchronized)

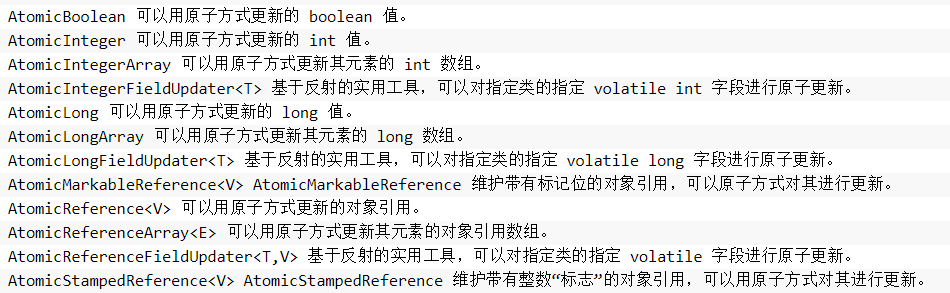
CAS存在ABA问题：内存V中存在一个变量A，线程1,2以获取到变量A，线程3还为获取，线程1,2都是希望将期望值更新为B，线程3希望值为A，线程1执行后，V的值变为B，然后线程3开始执行，获取到旧的预期值B，然后执行CAS操作，整个过程无其他线程，自然线程3将V的值变为A，然后线程2执行，将V的值变为了B，V的值从A--->B--->A--->B

例如，A有存款100，现要取50，但是取款机出问题了，开启了两个取款线程1,2，两个线程获取的旧的期望值为100，更新值为50，线程1执行完后，存款余额为50，此时线程2阻塞，有条新的线程3，给A汇款50，此时A的存款变为100，而此时再执行线程2，存款就变为50了

解决方式，加版本号，进行CAS指令时，不仅要比较旧的期望值A是否和地址V中的实际值是否相等，还要比较当前值A和地址中的值的版本号是否相等(线程获取到地址中的值A后，在执行CAS指令这段时间，是否对地址V的值修改过)

AtomicStampedReference和AtomicMarkableReference可以解决ABA问题

其他原子操作



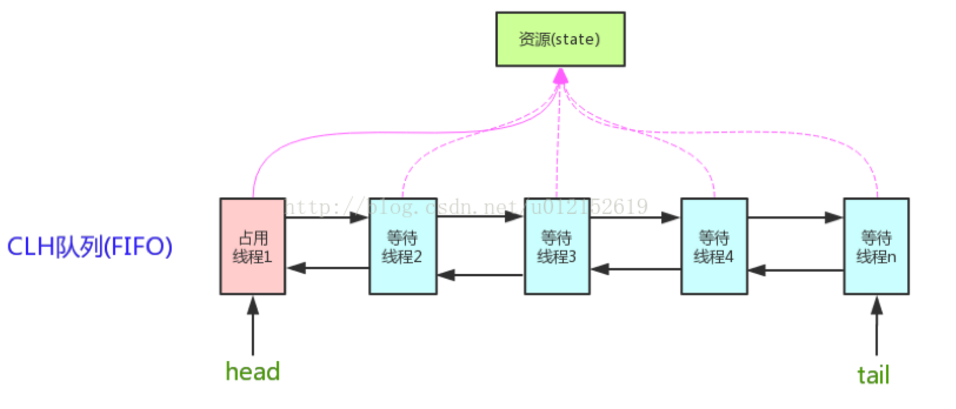
volatile实现了可见性，CAS实现原子操作

Lock锁的底层实现是通过AQS(AbstractQueuedSynchronizer)

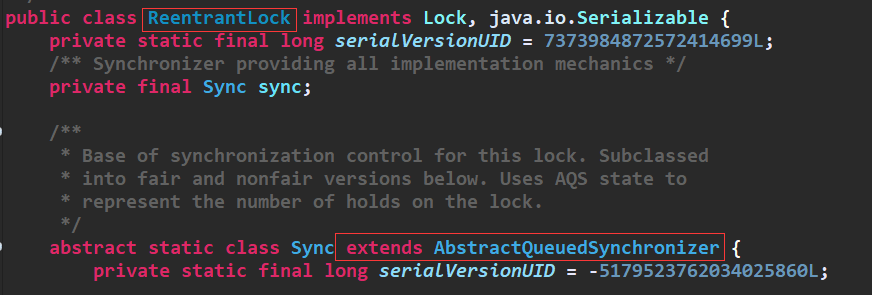
AQS采用的是CLH锁，CLH是一种自旋锁(自旋就是循环的意思，自旋锁和互斥锁类似，当线程访问互斥锁时，如果锁被占用，申请者只能等待锁释放，而如果自旋锁被占用，申请者就一直循环看这个锁是否释放了)

CLH锁是一种基于链表的自旋锁，在CLH中的节点QNode中包含一个locked字段，若该字段为true表示当前线程需要获取锁，且不释放锁，为false表示释放了锁(每个节点表示的就是一个线程)，当一个线程需要获取锁时，就会创建一个新的QNode，将其中的locked设置为true表示需要获取锁，然后加入到链表的尾部，链表中所有QNode的locked设置为ture，表示所有的线程都要获取锁，那么就是链表头节点(线程)获取锁，当释放锁时，locked变为false，则下一个节点获取到锁，如果一个节点想获取锁，此节点将会不断轮询前一个节点的locked，当前一个节点的lock变为false时，当前节点就可以获取到锁，CLH是基于FIFO(先来先服务)，CLH通过判断locked的值来决定当前线程是否拥有锁，资源消耗低

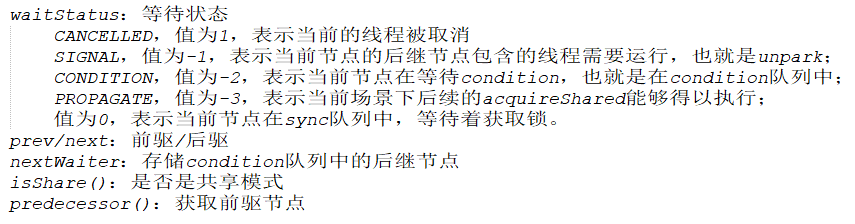
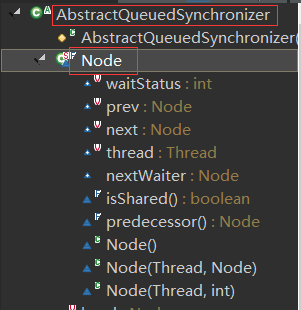
AQS内部维护了一个volatile的state和FIFO队列



state锁的状态，当state为0时，表示当前无线程持有锁，state>0时，表示当前锁被线程持有



Lock的锁是Sync，而sync又继承自AbstractQueuedSynchronizer(AQS)



在AbstractQueuedSynchronizer声明了一个Node(线程)，SHARED表示是共享模式/EXCLUSIVE表示独占模式(就是读-读/读-写)，share表示读锁，exclusive表示写锁，ReadWriteLock的读写锁就是在share/exclusive上声明的

<https://blog.csdn.net/u012152619/article/details/74977570>

Runnable的中文意思是任务

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

sleep：当线程睡眠时间满后，不一定会立即得到执行，因为此时可能CPU正在执行其他的任务。

yield：交出CPU，不释放锁，和sleep比较，yield不能控制具体的交出cpu的时间，并且yield只能让优先级相同或更高的获取CPU，yield并不会进入阻塞状态，而是重会就绪状态

JMM：Java Memory Model(java内存模型)

每个线程不能访问其他线性中的数据