## 多线程实践

一、Java多线程技能

1.1 进程和线程

进程：操作系统的基本执行单元

线程：程序的执行线索

1.2 创建线程的常用方式

1.继承Thread类

2.实现Runnable接口

3.实现Callable接口

1.3 线程安全问题

多线程并发访问临界资源，破坏原子操作

1.4 停止线程

1. interrupt方法

interrupted:判断当前线程是否中断，并清除中断标志位

isInterrupted:判断线程是否中断，不清除中断标志位

2. 异常抛出法

判断线程处于中断状态后，抛出interruptedException来防止线程继续执行（也可以使用return）

3. stop 方式（弃用）

1.5 暂停线程(弃用)

1.线程挂起suspend（如果一个线程使用打印功能时被挂起，那么其它线程无法使用该打印方法，因为使用了同步机制System.out.print(“hello”)

2.线程恢复 resume

1.6 线程放弃CPU资源

yield()方法

1.7 线程优先级

操作系统中，线程可以划分优先级，优先级高的线程获取cpu时间片的概率也高。线程具有继承性，即线程的优先级与创建它的线程的优先级保持一致。线程又具有随机性，不一定优先级高的线程总是最快的完成任务，但是总体来说优先级高的线程获取cpu资源的概率要大。可以通过setPriority()方法来设置线程的优先级。

1.8 守护线程

Java中存在两种线程，一种是用户线程，一种是守护线程。

守护线程最典型的应用是GC(垃圾回收)thread.setDaemon(true),当所有的用户线程结束了，守护线程会随着JVM一起结束。

二、对象及变量的并发访问

栈内存独立，堆内存共享。方法中的变量永远是安全的，多线程访问临界资源，破坏了原子操作，就会引发线程安全问题。同步的原则是对共享资源做同步，锁对象必须是同一个对象。

1.synchronized对象监视器为Object

1.当对一个方法加synchronized时，锁对象就是this。当出现异常时，线程会释放掉它拥有的锁对象。同步不具有继承性，即子类无法直接继承父类的同步方法，子类想要同步，必须在子类方法中添加synchronized关键字。

2.Synchronized锁重入：

指的是一个线程在synchronized方法中可以调用该类的另一个synchronized方法，该线程一定可以获取到该锁对象。当存在继承关系时，子类也可以通过可重入锁调用父类上锁（加synchronized）的方法。

3.数据类型String的常量池特性：

同步代码块不使用String作为锁对象。会造成同步，其他线程不执行。

4.多线程死锁问题：

多个线程等待对方的锁释放，造成线程无法继续执行，锁也无法释放，形成死锁。

2.synchronized对象监视器为Class

对象监视器为字节码对象，一定是同一个对象，可以完成同步功能，但是可能会影响效率。Synchronized修饰的方法锁对象是this。Synchronized修饰的static方法锁对象是class字节码对象，两者不同。

3.非线程安全是如何出现的

多线程访问临界资源，破坏了原子操作，造成了数据的不安全性。

4.volatile的主要作用

使变量在多个线程中可见，当线程访问volatile修饰的变量时，强制从公有堆栈中取值。

5.volatile与synchronized的区别

Volatile关键字：只能修饰变量，可以保持变量的可见性，但是不能保证原子性，非阻塞

Synchronized关键字：修饰方法或者代码块，可以保证可见性和原子性，阻塞，synchronized具有volatile的同步功能，它能够将共享内存数据同步到线程的工作内存中。

6.线程安全问题主要研究的两点：

原子性、可见性 （公有堆栈和私有堆栈及其同步问题）

7.使用wait()和notify()方法完成通信

1.wait()使线程等待，notify使等待的线程继续运行。wait()和notify()基于同一个锁对象进行通信。

2.调用wait(）或notify()方法前需要先获取到锁对象，即必须在同步方法或者同步代码块中。获取的锁对象和释放的锁对象必须是同一个即由获取到的锁对象调用wait()或notify()。不然会抛异常。

3.调用wait()方法后，线程释放掉了锁对象进入队列等待状态，等待另一个线程的notify方法。

4.调用notify()方法后，当前线程不会马上释放锁，呈wait状态的线程也不会立马获得锁对象，需要等到当前线程执行完程序即退出synchorined代码块。notify()只随机通知一个线程进行唤醒，notifyAll()唤起所有线程。

5.wait(long) 表示超过这段时间线程没有被唤醒则自动唤醒。

6.过早通知的问题 即当线程先执行notify()方法时可能会引起wait()方法调用后线程一直等待问题。

8.生产者和消费者模式的实现

存在虚假唤醒，比如生产者唤醒生产者，消费者唤醒消费者，使用while（条件）做检查，重复检查条件。

存在所有线程处于wait()状态情况，使用notifyAll()。

1.一生产一消费

2.一生产多消费

3.多生产一消费

4.多生产多消费

9.join()方法的实现

Join()方法表示当调用join()的线程结束后当前线程才继续执行。

Join(long millis)表示等待指定时间后继续执行。

如：在t2线程中调用t1.join()表示暂停t2线程，t1线程执行完后t2线程继续执行。

Join的内部是由wait()实现的，通过判断调用join的线程是否alive或者是否已经过了指定等待的时间。

Join(long) 和 sleep(long)的区别：

Join()内部是wait()实现，会释放锁对象，sleep()不会释放锁对象。注意，这里锁对象指的是线程对象。

10.ThreadLocal类的使用（需要阅读源码）

用于解决每个线程拥有自己的数据。ThreadLocal好比一个Map，key对应着每个线程，value对应着这个线程的数据。

Get()方法：在哪个线程中调用threadLocal.get()方法就是取这个线程对应的值。

Set()方法：设置当前线程的副本变量值。

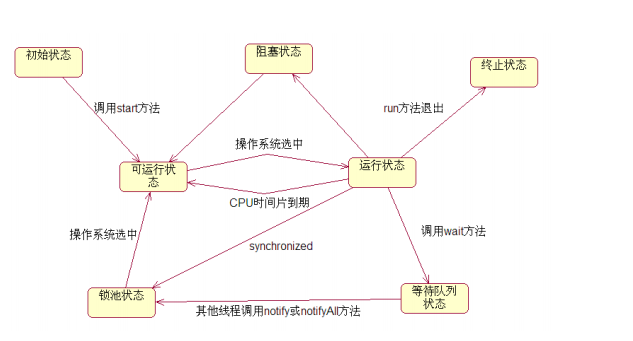
Remove()方法：移除当前线程的副本变量值。

InitialValue()方法：初始化一个threadlocal的值，每个线程都可以获取到这个值。

注意点：

建议是：每个线程只存一个变量，这样的话所有的线程存放到map中的Key都是相同的ThreadLocal，如果一个线程要保存多个变量，就需要创建多个ThreadLocal，多个ThreadLocal放入Map中时会极大的增加Hash冲突的可能。

11.线程状态变更图



三、ReentrantLock解析

<https://www.cnblogs.com/takumicx/p/9338983.html>

1.ReentrantLock和Synchronize的比较

Jdk中独占锁的实现除了使用关键字synchronized外,还可以使用ReentrantLock。虽然在性能上ReentrantLock和synchronized没有什么区别，但ReentrantLock相比synchronized而言功能更加丰富，使用起来更为灵活，也更适合复杂的并发场景。

**公平锁**：当锁可用时，在锁上等待的时间最长的线程将获得锁的使用权（讲究先到先得）。先被cpu时间片选中的线程先获得锁。大部分情况我们使用非公平锁，非公平锁的性能比公平锁的性能高，但是公平锁能够防止线程饥饿，某些情况会比较有用。

**非公平锁**：通过线程间的竞争获取锁的使用权。

**ReentrantLock可响应中断：**在死锁情况下，使用Synchronized会一直等待获取锁，造成程序无法继续运行。ReentrantLock获取锁时可以使用lockInterruptibly()方法，此方法线程可被外部中断，不必一直等待获取锁，解决死锁问题。

**获取锁时限时等待：**tryLock(time)指定时间内获取锁对象，成功返回true,失败返回false。

相同点：独占锁（同步）、可重入

不同点：ReentrantLock可以通过构造器实现公平锁和非公平锁。ReentrantLock可响应中断。ReentrantLock可限时获取锁对象。

2.同步方法:

调用lock.lock()方法的线程等价于获取了对象锁标记（对象监视器）。lock.unlock()释放锁对象。

2.Condition实现等待通知机制:

lock.newCondition()获取condition对象，并且一个lock对象可以创建多个Condition对象。在调用await()或signal()方法前需要先获取锁对象（lock.lock()），否则会抛出异常。并且线程的等待和通知需要针对于同一个condition。

object的wait()方法相当于condition中await();object的notify()方法相当于condition中的signal();object的notifyAll()方法相当于condition中的signalAll()。

注意：

当调用condition.await()方法时，线程自动释放掉当前持有的锁对象。当线程被唤醒时会自动获取到原有的锁对象。

关于condition的使用，可以参考经典案例：ArrayBlockingQueue

3.lock和unlock的实现逻辑

<https://cloud.tencent.com/developer/article/1038499>

**CAS简述：**

CAS(Compare And Swap)，即比较并交换。它是一种乐观锁（synchronized可以看做是悲观锁）是解决多线程并行情况下使用锁造成性能损耗的一种机制，CAS操作包含三个操作数——内存位置(V)、预期原值(A)和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在CAS指令之前返回该位置的值。CAS有效地说明了“我认为位置V应该包含值A；如果包含该值，则将B放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。

在JAVA中，sun.misc.Unsafe 类提供了硬件级别的原子操作来实现这个CAS。 java.util.concurrent 包下的大量类都使用了这个 Unsafe.java 类的CAS操作。

**AQS简述:**

AbstractQueuedSynchronizer 的缩写。AQS是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。

AQS使用一个FIFO的队列表示排队等待锁的线程，队列中的每个节点都是双向链表的一环（有prev,next），头节点head不与任何节点关联，其他的节点都与等待线程关联，每个节点维护一个等待状态waitStatus。0表示正常状态，-1表示通知状态，表示该线程释放锁之后需要唤醒（通知）他的后继节点（儿子节点）。每当有一个新的线程newThread排进队列时，该节点newNode(newThread)自动成为尾部节点（tail）。如下是ReentrantLock的结构图：

|  |
| --- |
|  |

**AQS中的先进先出队列：**

|  |
| --- |
|  |

**AbstractQueuedSynchronizer中的lock方法：**

|  |
| --- |
|  |

**AbstractQueuedSynchronizer中的acquire方法：**

|  |
| --- |
|  |

**非公平锁获取锁对象nonfairTryAcquire：**

|  |
| --- |
|  |

**公平锁获取锁对象tryAcquire**

|  |
| --- |
|  |

**添加等待队列节点addWaiter:**

|  |
| --- |
|  |

**挂起线程操作acquireQueued:**

|  |
| --- |
|  |

**锁的释放unlock:**

|  |
| --- |
|  |

**尝试释放锁对象tryRelease:**

|  |
| --- |
|  |

四、ReentrantReadWriteLock解析

1.简介

ReentrantLock具有完全互斥排他的效果，即同一时间内只有一个线程执行ReentrantLock.lock()方法后的任务，这样虽然保证的线程的安全性，但是效率低下。我们说线程安全性指的是多线程并发访问临界资源，破坏了原子操作。如果两个线程只是对临界资源进行读操作，是不会有线程安全性问题的。

ReentrantReadWriteLock类中有两个内部类，ReadLock、WriteLock都实现了Lock接口。ReadLock称为共享锁，WriteLock称为排它锁。读写锁的特性如下：允许多个线程同时进行读操作，但是同一时刻只允许一个线程进行写操作。

读读共享、读写互斥、写读互斥、写写互斥

五、ThreadLocal源码解析

多线程并发访问临界资源，破坏原子操作，会导致线程安全问题。将公共变量与ThreadLocal进行绑定，线程使用该公共变量时拿到的是在该线程中的副本变量，即将线程中的数据进行了隔离，多个线程能使用该变量但互不影响。

变量值的存放路径为：Thread->ThreadLocalMap->Entry->value，ThreadLocal作为Entry的key，通过key可以拿到value。

**1.成员变量**

内部类ThreadLocalMap 中的Entry[] table 数组（数据结构），table的长度必须是2的n次方，size表示map中entry的个数，threshold表示扩容阈值。阈值通常为2/3\*length。

ThreadLocalMap中内部类Entry的value表示线程存储的副本变量值。Entry继承WeakReference,Entry的key是ThreadLocal,是弱引用，在垃圾回收时被回收。

**2.构造方法**

无参构造

**3.查找元素**

1.获取当前执行线程thread。

2.获取线程中的变量threadLocalMap。

3.通过threadLocal在map中获取对应的entry。通过key的hash值找到元素在桶中的位置，特别注意需要对entry key的判断，经常存在entry不为空，key为空的情况（key为弱引用）。

4.返回对应的value值。

**4.插入元素**

1.同上

2.同上

3.当指定索引处存在值的时候，即发生了hash冲突。（1）若key相同则更新value值，（2）若key为null时，以当前索引为中心，向前后查找一组需要清除的entry对象进行清除操作，前后边界是桶中元素为null。向后查找过程中若存在相同的key则进行替换。若没有找到key，则更新当前索引处entry为新插入的值。（3）否则元素插入的位置是下一个最靠近当前索引的位置。

4.当指定索引处无值的时候直接新增一个entry对象。

5.当没有清除到key=null的entry对象且size>threshold时进行rehash（）操作。

6. rehash（）操作，清除table中所有key=null的entry对象，

size>0.75threshold时进行扩容。

**5.移除元素**

1.同上

2.同上

3.将key置为null。

4.清除无效的entry对象（将value置为null）。

**6.扩容**

1.数组容量变为原来的两倍。

2.清除key=null的entry对象。

3.元素在放入新的数组时，若发生了hash冲突，则元素插入到下一个最近的索引处。

**7.清除无效的entry对象**

private int expungeStaleEntry(int staleSlot) {}

staleSlot索引处entry对应的key=null。返回值是staleSlot往后entry为null的索引。

1.清除staleSlot索引处的entry对象，

2.进行rehash操作，直到entry为null。staleSlot往后查找，当entry的key=null时，将该处的entry对象置为空，否则当entry现在所在的索引位置i与应当在索引位置h不同，将entry调整到索引h位置，或者最接近h的位置。

# 六、线程池原理ThreadPoolExecutor

<http://www.cnblogs.com/dolphin0520/p/3932921.html>

线程池是一种多线程技术，最大限度的提高cpu的利用率，提升系统的运行效率。然而线程的创建和销毁是十分消耗系统资源和时间的，线程池的作用在于集中管理线程的创建和销毁以及实现线程的复用。

1. ThreadPoolExecutor的核心构造方法

|  |
| --- |
| **public** ThreadPoolExecutor(  **int** corePoolSize, **int** maximumPoolSize, **long** keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler) {  } |

1. 关键参数解析

corePoolSize：核心线程数，类似于公司的骨干员工

maximumPoolSize：最大线程数，任务太多后，招聘的新员工

keepAliveTime：线程空闲时的存活时间

unit：时间单位

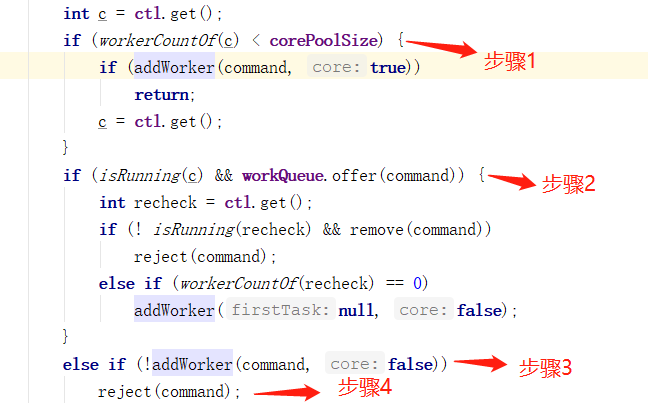
workQueue：任务队列，开发的项目

threadFactory：创建线程的工厂类

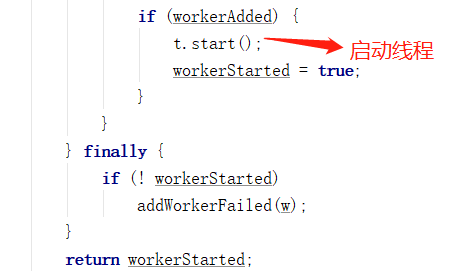
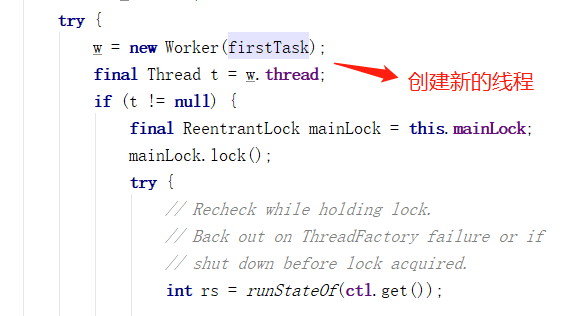
handler：拒绝策略

1. 线程池工作流程 execute方法为入口
2. 接受到任务后，先判断当前线程数是否小于核心线程数，如果小于则创建线程并执行任务。否则进入步骤2。
3. 判断当前任务是否能够添加到任务队列中，如果能则添加到队列，等待线程执行。如果不能则执行步骤3。
4. 判断当前线程数是否大于允许的最大线程数，如果小于则创建线程并执行任务，否则执行步骤4。
5. 执行拒绝策略，拒绝任务。
6. 关键方法解析

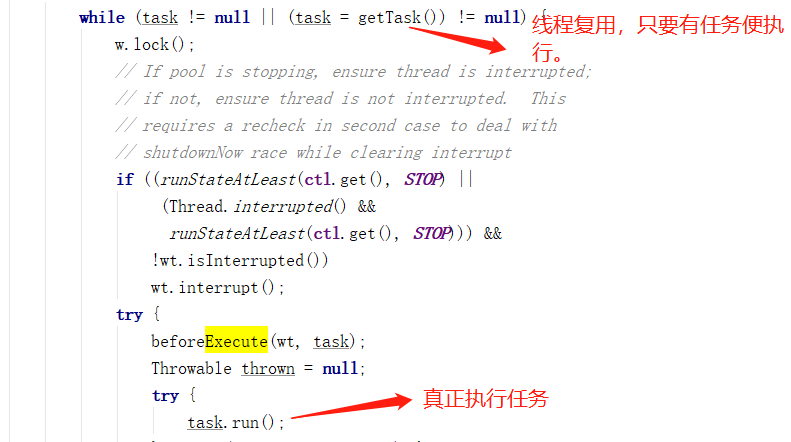
ThreadPoolExecutor#execute为主流程方法



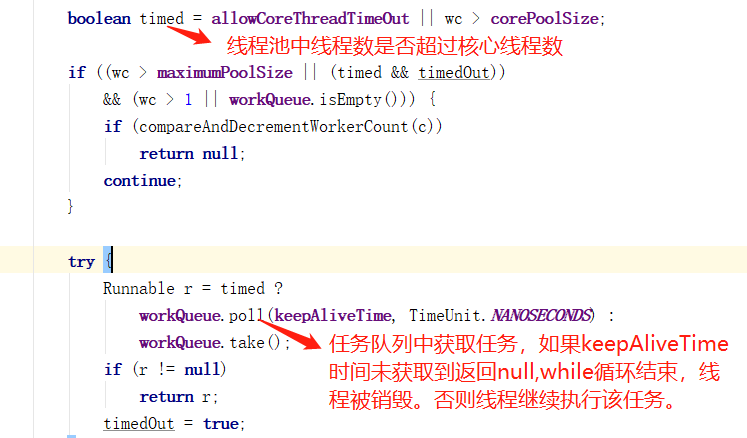
ThreadPoolExecutor#addWorker创建新的线程并执行任务。



ThreadPoolExecutor#runWorker执行任务。



ThreadPoolExecutor#getTask获取任务。



1. 简单使用线程池

newSingleThreadExecutor：只有一个线程，任务队列为Integer.MAX\_VALUE。

newCachedThreadPool：最大线程数为Integer.MAX\_ VALUE。

newFixedThreadPool：线程数固定，任务队列为Integer.MAX\_ VALUE。

场景：比如接受到异步通知后立即响应，内部逻辑可以使用线程池完成。

也可使用mq完成。

|  |
| --- |
| **class** ThreadPoolTest {  **public static void** main(String[] args) { *// ExecutorService executor = Executors.newSingleThreadExecutor(); // ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.newCachedThreadPool(); // ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.newFixedThreadPool(5);* ThreadPoolExecutor executor = **new** ThreadPoolExecutor(5, 10, 200, TimeUnit.***MILLISECONDS***,  **new** ArrayBlockingQueue<>(5));  **for** (**int** i = 0; i < 16; i++) {  MyTask myTask = **new** MyTask(i);  executor.execute(myTask);  System.***out***.println(**"线程池中线程数目："** + executor.getPoolSize() + **"，队列中等待执行的任务数目："** +  executor.getQueue().size() + **"，已执行玩别的任务数目："** + executor.getCompletedTaskCount());  }  executor.shutdown();  } } **class** MyTask **implements** Runnable {  **private int taskNum**;  **public** MyTask(**int** num) {  **this**.**taskNum** = num;  }  @Override  **public void** run() {  System.***out***.println(**"正在执行task "** + **taskNum**);  **try** {  Thread.*currentThread*().*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"task "** + **taskNum** + **"执行完毕"**);  } } |