# 多线程

相关链接：

https://segmentfault.com/a/1190000013813740

https://blog.csdn.net/javazejian/article/details/50896505

http://ifeve.com/java-concurrency-thread-directory/

## 零散

### CyclicBarrier和CountDownLatch区别

都在java.util.concurrent包下，都可以用来表示代码运行到某个点上。

CyclicBarrier的某线程运行到某个点上后，该线程立即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch的某线程运行到某个点上后，只是给该变量值-1，该线程继续运行。

CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务。

CyclicBarrier可重用；CountDownLatch不可重用，当数值为0时该CountDownLatch就不能用了。

### 多线程开发良好的实践

-1）给线程命名

-2）最小化同步范围

-3）优先使用volatile

-4）尽可能使用更高层次的开发工具而非wait()和notify()来实现线程通信，如BlockingQueue，Semeaphone

-5）优先使用并发容器而非同步容器

-6）考虑使用线程池

### Java中用到的线程调度算法是什么？

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

## 基本概念

### 进程与线程的定义

进程：一个进程是一个独立（self contained）的运行环境，它可以被看作一个程序或者一个应用。

线程：线程是在进程中独立运行的一个子任务，是操作系统能够进行运算调度的最小单位。

多线程编程好处：

提高资源利用率，CPU不会因为某个线程需要等待资源而进入空闲状态。

例如，从磁盘读取一个文件需要5秒，处理一个文件需要2秒。CPU等待第一个文件被读取完。然后开始读取第二个文件；当第二文件在被读取的时候，CPU会去处理第一个文件。

### 守护线程和用户线程

线程分为守护线程、用户线程（默认）。

setDaemon(true) //设置线程为守护线程（在调用start()方法之前设置）

守护线程：会作为进程的守护者，如果进程内没有用户线程时，守护线程也会被销毁，即使是该线程没有运行结束。

守护线程最典型的例子就是GC线程。

### 线程安全

问题：一个线程在读一个内存时，另一个线程正向该内存进行写操作，那进行读操作的那个线程将获得什么结果呢？

线程安全：如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

非线程安全：在多线程环境下，数据访问前后出现结果不一致情况。存在多线程先后修改数据。

线程安全的代码不包含竞态条件。当多个线程同时更新共享资源时会引发竞态条件。

线程安全：Vector、HashTable、StringBuffer

非线程安全：ArrayList、LinkedList、HashMap、HashSet、TreeMap、TreeSet、StringBuilder

相关链接：https://www.cnblogs.com/yjd\_hycf\_space/p/7760248.html

### 共享资源介绍

-1）局部变量

局部变量存储在线程自己的栈中，即局部变量永远不会被多个线程共享。因此基础类型的局部变量是线程安全的。

public void method1() {

long threadSafeInt = 0; //基础类型的局部变量

threadSafeInt++;

}

-2）局部的对象引用

因为引用所指的对象并没有存储在线程的栈内，而是都存在于共享堆中。如果在某个方法中创建的对象不会逃逸出该方法（附：该对象没有return，从而不会被其它方法获取）

逃逸的方式：将该对象返回

线程逃逸规则：

如果一个资源的创建、使用、销毁都在同一线程内完成，且永远不会脱离该线程的控制，则该资源的使用就是线程安全的。

资源可以是对象、数组、文件、数据库连接、套接字等等。这里的销毁指不再有引用指向对象。

-3）对象成员

对象成员存储在堆上。如果两个线程同时更新同一个对象的同一个成员，那这个代码就不是线程安全的。

当然，如果这两个线程在不同的对象实例上调用同一方法，就不会导致竞态条件。

### 线程调度器（Thread Scheduler）和时间分片（Time Slicing）

线程调度器：是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个并启动它，它的执行便依赖于线程调度器的实现。

时间分片：指的是将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间基于线程优先级或线程等待的时间。

注：线程调度由应用程序来控制更好，即不要让你的程序依赖于线程的优先级。

### 线程运行机制

一个CPU每个时刻只能执行一条线程；

操作系统给每条线程分配不同长度的时间片；(出现线程状态)

操作系统会从一堆线程中随机选取一条执行；(出现上下文切换)

每条线程用完自己的时间片后，即使任务还没完成，操作系统也会剥夺它的执行权，让另一条线程去执行。

### 线程的状态（生命周期）

分为5种：新建、就绪、运行、阻塞、死亡

1.新建态(new)：新创建一个线程对象，jvm为其分配内存空间，并初始化成员的变量

2.就绪态(Runnable)：调用该对象的start()方法，线程进入可运行线程池中，等待获取CPU的使用权。

3.运行态(Running)：就绪态的线程获取了CPU，开始执行程序代码。

4.阻塞态(Blocked)：线程因为某种原因放弃了CPU的使用权，暂时停止运行。阻塞态的线程直到进入就绪态，才有机会转到运行态。

分类：

1、等待阻塞:调用wait()方法，JVM会把该线程放入等待池中

2、同步阻塞:在获取同步锁时，该同步锁已被别的线程占用，则JVM会把该线程放入锁池中

3、其它阻塞:执行sleep()或join()方法，或者发出了I/O请求

5.死亡态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期

### 线程间的通信机制

java里面进行多线程通信的主要方式就是共享内存的方式，线程安全性问题主要关注3个：原子性、可见性、有序性。

java内存模型(JMM)解决了可见性和有序性的问题，而锁解决了原子性的问题。

在java中多线程间的通信是使用“等待/通知机制”来实现的。

分类：

synchronized锁：Object.wait/notify/notifyAll

lock锁：Condition.await/signal/signalAll

使用：

1）synchronied关键字的等待/通知机制

-1）前提：必须获取对象的对象级别锁，即synchronied修饰的函数或代码块；

-2）线程A调用了对象O的wait()方法进入等待状态；

-3）线程B调用了对象O的notify()或notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。

2）条件对象的等待/通知机制

-1）即Lock锁对象，通过锁对象的条件对象来实现等待/通知机制

-2）Condition conditionObj = ticketLock.newCondition(); //创建条件对象

-3）线程A调用了条件对象的await()方法进入等待状态

-4）线程B调用了条件对象的signal()或者signalAll()方法，线程A收到通知后从条件对象的await()方法返回，进而执行后续操作。

注:两种方式并不会立即激活一个等待线程。它们仅仅都只是解除等待线程的阻塞状态，以便这些线程可以在当前线程解锁或者退出同步方法后，通过争夺CPU执行权实现对对象的访问。

如何正确使用wait()方法？使用if还是while？

synchronized(obj) {

while (condition does not hold)

obj.wait(); //(Releases lock, and reacquires on wakeup)

… // Perform action appropriate to condition

}

### 如何检测一个线程是否持有对象监视器

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有时才会返回true。这是一个static方法，意味着“某条线程”指的是当前线程。

## 并发问题

并发需要注意的问题：上下文切换、死锁、资源限制

### 并发(concurrency)与并行(parallellism)

并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

区分：

并发(concurrency)：是指在一个cpu处理的线程中一直不断的切换任务，最终给人的感觉就像是执行了多个任务。

并行(parallellism)：在多个cpu同一时间处理多个任务

### 上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

当一条线程的时间片用完后，操作系统会暂停该线程，并保存该线程相应的信息，然后再随机选择一条新线程去执行，该过程就是“上下文切换”。

多线程不一定在多核处理器才支持的，单核处理器也支持。

CPU通过给每个线程分配一定的时间片，由于时间非常短通常是几十毫秒，所以CPU可以不停的切换线程执行任务从而达到了多线程的效果。

但是由于在线程切换的时候需要保存本次执行的信息，在该线程被CPU剥夺时间片后又再次运行恢复上次所保存的信息的过程就称为上下文切换。

线程对于异步操作很好，如果你有1000个线程应用程序中运行，上下文切换可能比你的业务实际工作还要做得多。克服这个问题的方法之一是使用'固定'的线程池。固定意思大小不变的有界的。

永远不要让它无限：您可以使用连接池，线程池，队列和许多其他机制来获得性能。可能在测试过程中，所有这些确实非常好，但是如果不限制最大值将自讨苦吃。，对不可预知的条件这是一个很好的防守的做法。

上下文切换步骤：

1）暂停正在执行的线程；

2）保存该线程的相关信息（如：执行到哪一行、程序计算的中间结果等）；

3）从就绪队列中随机选一条线程；

4）读取该线程的上下文信息，继续执行。

上下文切换是由开销的：

1）每次进行上下文切换时都需要保存当前线程的执行状态，并加载新线程先前的状态；

2）原因：如果上下文切换频繁，CPU在上下文切换上的时间占比就会上升，而真正处理任务的时间占比就会减少。

附：为了提高并发程序的执行效率，让CPU把时间花在刀刃上，我们需要减少上下文切换的次数。

减少上下文切换：

1）减少线程的数量

2）控制同一把锁上的线程数量

3）采用无锁并发编程。采用 CAS(compare and swap) 算法，仅在线程内部需要更新共享变量时使用CAS算法来更新，这种方式不会阻塞线程，并保证更新过程的安全性。（如 Atomic 包就是采用 CAS 算法）

### 任务的状态

1）需要并发执行的任务是无状态的：HASH分段

所谓无状态：是指并发执行的任务没有共享变量，它们都独立执行。对于这种类型的任务可以按照ID进行HASH分段，每段用一条线程去执行。

2）需要并发执行的任务是有状态的：CAS算法

如果任务需要修改共享变量，那么必须要控制线程的执行顺序，否则会出现安全性问题。

解决线程的执行顺序：你可以给任务加锁，保证任务的原子性与可见性，但这会引起阻塞，从而发生上下文切换；

### 死锁

当多个线程相互等待已经被对方占用的资源时，就会产生死锁。

实例：

描述：线程1给对象A加锁后再给对象B加锁，而线程2给对象B加锁后再给对象A加锁。使得线程1和线程2因为需获取的资源已被占用而发生死锁。

情况如下：

Thread1 lock A，waits for B

Thread2 lock B，waits for A

代码：

public class DeadLock {

private Object lockA; //对象A

private Object lockB; //对象B

Thread t1 = new Thread(new Runnable(){ //线程1

void run () {

synchronized (lockA) { //给对象A加锁

Thread.sleep(5000);

synchronized (lockB) { //再给对象B加锁

... //省略

}

}

}

});

Thread t2 = new Thread(new Runnable(){ //线程2

void run () {

synchronized (lockB) { //给对象B加锁

Thread.sleep(5000);

synchronized (lockA) { //再给对象A加锁

... //省略

}

}

}

});

t1.start();

t2.start();

}

如何避免死锁：

--指定获取锁的顺序！！！

1）尽量一个线程只获取一个锁

2）一个线程只占用一个资源

3）尝试使用定时锁，使用tryLock(timeout)来替换使用内部锁机制

### 重排序

重排序指编译器、处理器在不改变程序执行结果的前提下，重新排列指令的执行顺序，以达到最佳的运行效率。

数据依赖：若相邻的两条指令访问同一个变量，并且其中又一条指令执行写操作，那么这样的两条指令之间存在数据依赖。对于有数据依赖关系的指令，不会发生重排序。

编译器和处理器只会对没有依赖关系的指令进行重排序。

as-if-serial：？？？

volatile不会发生重排序

分类：

-1）编译器优化：对于没有数据依赖关系的操作，编译器在编译的过程中会进行一定程度的重排。例如，a=1和x=b，它们之间没有数据依赖关系，调换位置没影响。

-2）指令重排序：CPU优化行为，也是对不存在数据依赖关系的指令进行一定程度的重排。

## 解决线程同步

1）synchronized关键字来解决同步问题

2）在Java SE5.0引入lock锁对象的相关类

### 生产者和消费者

详情: https://blog.csdn.net/javazejian/article/details/50878665

1）一个生产者和一个消费者

-1)为保证生产烤鸭后才能消费也就是生产一只烤鸭后才能消费一只烤鸭，我们使用了等待/通知机制，wait()和notify()。

-2) 首先要创建一个对象锁，我们这里使用的重入锁ReestrantLock类，然后通过手动设置lock()和unlock()的方式去获取锁以及释放锁。为了实现等待/通知机制，我们还必须通过锁对象去创建一个条件对象Condition，然后通过锁对象的await()和signalAll()方法去实现等待以及通知操作。

2）多个生产者和多个消费者

## 锁介绍

相关链接：https://segmentfault.com/a/1190000017766364#articleHeader8

### 分类（15种）

公平锁/非公平锁

可重入锁/不可重入锁

独享锁/共享锁

互斥锁/读写锁

悲观锁/乐观锁

分段锁

偏向锁/轻量级锁/重量级锁

自旋锁

### 公平锁/非公平锁

-1）公平锁：

指多个线程按照申请锁的顺序来获取锁。

-2）非公平锁：

指多个线程获取锁的顺序并不是按照申请锁的顺序。有可能造成优先级反转或饥饿现象

ReentrantLock通过构造函数指定该锁是否是公平锁，默认非公平锁。非公平锁的优点是吞吐量比公平锁大。

Synchronized也是一种非公平锁。由于其并不像ReentrantLock通过AQS来实现线程调度，所以并没有任何方法将Synchronized变成公平锁

### 重入锁/不可重入锁

-1）可重入锁

指可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提是指同一个对象或class）。

ReentrantLock和Synchronized都是可重入锁。

实例：

synchronized void method1() throws Exception {

Thread.sleep(500);

method2();

}

synchronized void method2() throws Exception {

Thread.sleep(1000);

}

说明：如果synchronized不是可重入锁，method2可能不会被当前线程执行，可能造成死锁。

-2）不可重入锁：

与可重入锁相反，不可递归调用，递归调用就发生死锁。

实例1：

--使用自旋锁来模拟一个不可重入锁

public class UnreentrantLock {

private AtomicReference<Thread> owner = new AtomicReference<Thread>();

public void lock() {

Thread current Thread.currentThread();

//“自旋”

for (;;) {

if (!owner.compareAndSet(null, current)) {

return;

}

}

}

public void unlock() {

Thread current = Tread.currentThread();

owner.compareAndSet(current, null);

}

}

说明：使用原子引用来存放线程，同一线程两次调用lock方法，如果不执行unlock()方法释放锁，第二次调用自旋时会产生死锁。这个锁就不是可重入的，而实际上同一个线程不必每次都去释放锁再来获取锁，这样的调度切换是很耗资源的。

实例2：

--把它变成一个可重入锁

public class UnreentrantLock {

private AtomicReference<Thread> owner = new AtomicReference<Thread>();

private int state = 0;

public void lock() {

Thread current = Thread.currentThread();

if (current == owner.get()) {

state++;

return;

}

//“自旋”

for (;;) {

if (!owner.compareAndSet(null, current)) return;

}

}

public void unlock() {

Thread current = Thread.currentThread();

if (current == owner.get()) {

if (state != 0) {

state--;

} else {

owner.compareAndSet(current, null);

}

}

}

}

说明：在执行每次操作之前，判断当前锁持有者是否是当前对象，采用state计数，不用每次去释放锁。

ReentranLock可重入锁实现：

--参照nonfairTryAcquire方法。

### 独享锁/共享锁

-1）独享锁：

该锁每一次只能被一个线程所持有。

-2）共享锁：

该锁可被多个线程共有，典型的就是ReentrantReadWriteLock里的读锁，它的读锁是可以被共享的，但是它的写锁确每次只能被独占。

另外读锁的共享可保证并发读是非常高效的，但是读写和写写，写读都是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

对于Synchronized而言，当然是独享锁。

### 乐观锁/悲观锁

-1）悲观锁：

参考synchronized关键字

在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

一个线程持有锁会导致其它所有需要此锁的线程挂起。

如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

-2）乐观锁：

每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

### 自旋锁

自旋锁：指当一个线程在获取锁时，若该锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取锁才会退出循环。

cas(compare and swap)算法是一种无锁算法，是乐观锁的一种实现方式，cas算法中又涉及到自旋锁。

简单例子：

public class SpinLock {

private AtomicReference<Thread> cas = new AtomicReference<Thread>();

public void lock() {

Thread current = Thread.currentThread();

// 利用cas

while (!cas.compareAndSet(null, current)) {

// do nothing

}

}

public void unlock() {

Thread current = Thread.currentThread();

cas.compareAndSet(current, null);

}

}

说明：lock()方法利用cas，当第一个线程A获取锁时，能成功获取到，不会进入while循环；在线程A没有释放锁，另一线程B又来获取锁，此时由于不满足cas，所以就会进入while循环，不断判断是否满足cas，直到线程A调用unlock()方法释放锁。

# Java内存模型（JMM，Java memory model）

相关链接：

http://www.hollischuang.com/archives/1003

## 相关概念

多线程通信的主要方式就是共享内存的方式，线程安全性问题主要关注3个：原子性、可见性、有序性。

java内存模型(JMM)解决了可见性和有序性的问题，而锁解决了原子性的问题。

并发的两个关键问题：线程之间如何通信及线程之间如何同步。

通信：指线程之间以何种机制来交换信息。在命令式编程中，线程间的通信机制有两种：共享内存和消息传递。

Java内存模型的抽象：

所有实例域、静态域和数组元素存储在堆内存中，堆内存在线程之间共享。

局部变量(Local variables)、方法定义参数(formal method parameters)和异常处理器参数(exception handler parameters)不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

java线程之间的通信由java内存模型(JMM)控制。JMM决定一个线程对共享变量的写入对另一个线程可见。

JMM定义了线程和主内存之间的抽象关系：

线程之间的共享变量存储在主内存(main menory)中，每个线程都有一个私有的本地内存(local memory)，本地内存中存储了该线程以读/写共享变量的副本。

本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其它的硬件和编译器优化。

分类：主内存(main memory)和工作内存(working memory)

1）主内存：存储类成员变量(static修饰)等。

2）工作内存：java线程的本地内存，是单独给某个线程分配的，存储局部变量等，同时会复制主内存的共享变量作为本地的副本。

3）主存为所有线程共享，工作内存中存放的是该线程所需共享变量的拷贝（目的是为了减少和主内存通信的频率，提高效率）。

4）线程要对主内存中的内容进行操作时，首先需要拷贝到自己的工作内存，一般为了速度，工作内存一般是在cpu的cache中的。

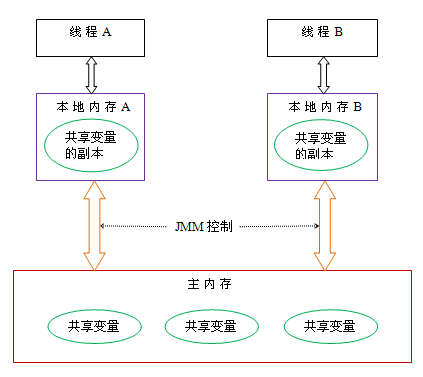
5）volatile变量不会产生工作内存的拷贝，而是直接与主内存交互。

注：Java内存模型(JMM)和JVM运行时数据区不是同一概念

-1）JMM都是围绕原子性、可见性、有序性来讲的；

-2）JMM定义了JVM如何与计算机的内存进行交互。

线程对变量的所有操作都需要在工作内存中完成，不可直接操作主内存。



从上图来看，线程A和线程B之间通信时：

-1）首先，线程A把本地内存A更新过的共享变量刷新到主内存中

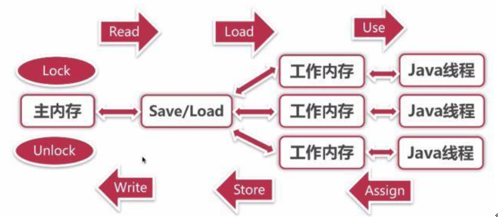
-2）然后，线程B到主内存中去读取线程A之前已更新过的共享变量。



如图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值(假设为1)临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

JMM通过控制主内存与每个线程的本地之间的交互，来提供内存可见性保证。

## 线程的原子操作



Java语言规范定义了线程的六种原子操作：

-1）read

负责从主存储器（main memory）读取到工作存储器（working memory）

-2）write

负责从工作存储器（working memory）写回到主存储器（main memory）

-3）use

表示线程引用工作存储器（working memory）的值

-4）assign

表示线程将值指定给工作存储器（working memory）

-5）lock

表示线程取得锁定

-6）unlock

表示线程解除锁定

## 原子性、可见性、有序性

### 可见性

可见性指的是一线程修改完某个共享变量后，另一线程立即能访问到该变量最新修改后的值。即一条线程对共享变量的修改，对其它线程立即可见。

可见性问题描述：

-1）当本地线程在创建时，会从主存中读取一个共享变量的副本，且仅修改该副本而并不会立即刷新到主存中去，这样其它线程并不会获取到共享变量的修改。

-2）局部变量不存在可见性问题，而共享变量就会有可见性问题。

可见性遵循一个规则：

1）当一个线程运行结束后，所有写的变量都会被flush回主存(main memory)中

2）当一个线程第一次读取某个变量时，会从主存(main memory)中读取最新的

3）volatile变量会被立即写回到主存(main memory)中

4）当一个线程释放锁后，所有的变量的变化都会flush到主存(main memory)中，然后一个使用了这个相同的同步锁的进程，就会重新加载所有的使用到的变量，这样保证了线程的可见性。

### 原子性

原子性指的是当某一个线程修改i的值的时候，从取出i到将新的i的值写给i之间不能有其他线程对i进行任何操作。

对任意单个volatile变量的读/写具有原子性。但类似于volatile++这种复合操作不具有原子性。

通过锁机制或者CAS操作可以保证操作的原子性。

# 创建线程

## 创建形式（3种）

-1. 继承Thread类，重写run()方法

new TestThread().start(); //TestThread继承了Thread类

-2. 实现Runnable接口，重写run()方法，并将对象传入thread对象中

new Thread(new TestRunnable()).start(); //TestRunnable实现Runnable接口

-3. 实现Callable接口，重写call()方法，将对象传入FutureTask对象中，再将Future对象传入Thread对象中

-1）创建Callable接口的实现类，重写call()方法

public class CallableTest implements Callable<String> { ... //省略}

-2）创建Callable对象，并将对象传入FutureTask对象中

Callable callable = new CallableTest();

FutureTask<String> task1 = new FutureTask<String>(callable);

-3）创建并启动线程

new Thread(task1).start();

## 线程相关方法描述

### Thread方法描述

实现Runnable接口比继承Thread类所具有的优势：

-1）避免java中的单继承的限制；

-2）适合多个相同的程序代码的线程去处理同一个资源；

-3）线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

Thread的静态方法：

-1）currentThread() :获取当前线程

-2）sleep() :让当前线程睡眠，交出CPU，让CPU去执行其它任务。1、该方法不会释放锁，当其它线程无法访问到该对象。2、需捕获异常

-3）yield() :交出CPU权限，让CPU去执行其他的线程。 1、不释放锁。2、只能让拥有相同优先级的线程有获取CPU执行时间的机会

Thread的成员方法：

-1）t.start()和run()

-2）t.join()、t.join(1000);

join()：把指定的线程加入到当前线程。例main是主线程，在main中创建了线程A，若在main中调用了线程A的join()方法，则在等线程A结束后才会再执行main代码

sleep和wait的区别：

-1）sleep方法来自于Thread类，该方法不会释放锁，使用时需要捕获异常。

-2）wait方法来自于Object类，该方法会释放锁。

-3）wait、notify和notifyAll方法只能在同步控制方法或同步控制块中使用，而sleep方法可以在任何地方使用。

例如：

synchronized(x){ //在同步块中使用wait方法

x.notify();

//或x.wait();

}

wait()与notify()/notifyAll() 区别:

-1）前提：调用该方法的当前线程必须持有对象的监视器锁，通俗来说，只能在同步代码块中使用这些方法。

-2）wait方法：阻塞当前线程，让出监视器锁，不再参与锁竞争，知道其它线程来通知，或者直到设定的超时等待时间到了。

-3）notify和notifyAll方法：唤醒通知那些阻塞线程，让它们从wait处返回。

如何确保main方法所在的线程在main方法退出之前执行完？

答：使用Thread类的join()方法

### Object方法描述

Object方法描述：

-1）wait()、notify()和notifyAll()方法是native方法，且是final方法，不可被重写；

-2）调用某对象的wait()方法能让当前线程阻塞，前提是当前线程已拥有此对象的锁

-3）调用某对象的notify()方法能够唤醒一个正在等待该对象锁的线程。若存在多个等待该对象锁的线程，则只能唤醒其中一个线程。

-4）调用notifyAll()能够唤醒所有正在等待这个对象锁的线程

为啥这些方法是Object类的？

由于每个对象都拥有monitor（即锁），所以让当前线程等待某个对象的锁，当然应该通过这个对象来操作了。而不是用当前线程来操作，因为当前线程可能会等待多个线程的锁，如果通过线程来操作，就非常复杂了。

如果调用某个对象的wait()方法，当前线程必须拥有这个对象的monitor（即锁），因此调用wait()方法必须在同步块或者同步方法中进行（synchronized块或者synchronized方法）。

### Condition中方法的描述

Condition中方法的描述：

-1）Condition是在java 1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。

-2）Condition是个接口，基本的方法就是await()和signal()方法。

-3）Condition依赖于Lock接口，生成一个Condition的基本代码是lock.newCondition()

-4）调用Condition的await()和signal()方法，都必须在lock保护之内，就是说必须在lock.lock()和lock.unlock之间。

## Callable和Future介绍

Callable和Future，一个产生结果，一个拿到结果；

Callable接口提供的call()方法作为线程执行体，可以有返回值，该返回值能被Future拿到，也可以声明抛出异常。

Java5提供了Future接口来代表Callable接口里call方法的返回值，并且提供了一个Future接口的实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。

### Callable接口

Runnable接口与Callable接口描述：

public interface Runnable {

public abstract void run(); //该方法没有任何返回值

}

public interface Callable<V> {

V call() throws Exception; //有返回值，可抛出异常

}

Runnable的run方法没有返回值，不能抛出异常。

抛出异常作用：若在任务执行过程发生异常，我们可将它向上抛给调用者，我们甚至可利用这个特性中断一个任务的执行。

注：不能直接调用call()方法，否则无法利用多线程。我们应将call方法提交给另外一个线程执行, 并在合适的时候, 判断任务是否完成, 然后获取线程的执行结果或者撤销任务, 这种思路的实现就是Future接口

一般配合ExecutorService接口来使用它：

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

### Future接口

用来代表一个异步操作的执行结果。你可以用它来判断任务是否完成、进行中断任务操作、获取执行任务的结果。

Future接口描述：

public interface Future<V> {

//获取执行结果，若任务还在执行中，就阻塞等待

V get() throws InterruptedException, ExecutionException;

//在指定时间内得到执行结果

V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;

//尝试取消一个任务的执行, 它的返回值是boolean类型, 表示取消操作是否成功

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

//一个任务已经结束, 则返回true。存在三种情况：1、正常执行；2、抛出异常；3、任务取消

boolean isDone();

}

注：cancel操作返回true并不代表任务真的就是被取消了，这取决于发动cancel状态时任务所处的状态：

-1）如果发起cancel时任务还没有开始运行，则随后任务就不会被执行；

-2）如果发起cancel时任务已经在运行了。若mayInterruptIfRunning为true，则当前在执行的任务会被中断；若为false则可以允许正在执行的任务继续运行，直到它执行完。

### FutureTask介绍

是Future接口的实现类，表示了一种抽象的可生成结果的计算。

FutureTask中所使用到的接口：Runnable、Callable、Future、RunnableFuture以及所使用到的工具类Executors、Unsafe。

FutureTask实现了Runnable和Future两个接口，因此它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

当返回值并不是立刻得到时，用另一个线程去计算返回值，而当前线程在使用这个返回值之前可以做其它的操作，等到需要这个返回值时，再通过Future得到。

构造函数：

public FutureTask(Callable<T> callable) {...}

public FutureTask(Runnable runnable, T result) {...}

示例代码：

--1）使用Callable、Future

public class Main {

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

ExecutorService executorService = Executors.newCachedThreadPool();

Task task = new Task();

Future<Integer> future = executorService.submit(task);

executorService.shutdown();

System.out.println("主线程在执行任务...");

Thread.sleep(2000);

System.out.println("task运行结果:"+future.get());

System.out.println("所有任务执行完毕");

}

}

class Task implements Callable<Integer> {

@Override

public Integer call() throws Exception {

System.out.println("子线程在执行任务...");

Thread.sleep(5000);

return 1000;

}

}

执行结果：

子线程在执行任务...

主线程在执行任务...

task运行结果:1000

所有任务执行完毕

--2）使用Callable、FutureTask

public class TestCallableAndFutureTask {

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

ExecutorService executorService = Executors.newCachedThreadPool();

Task task = new Task();

FutureTask<Integer> futureTask = new FutureTask<Integer>(task);

executorService.submit(futureTask);

executorService.shutdown();

System.out.println("主线程在执行任务...");

Thread.sleep(2000);

System.out.println("task运行结果:" + futureTask.get());

}

}

class Task implements Callable<Integer> {

@Override

public Integer call() throws Exception {

System.out.println("子线程在执行任务...");

// 模拟任务耗时

Thread.sleep(5000);

return 1000;

}

}

执行结果：

子线程在执行任务...

主线程在执行任务...

task运行结果:1000

所有任务执行完毕

实例：

public class CallableAndFuture {

public static void main(String[] args){

ExecutorService threadPool = Executors.newSingleThreadExecutor();

Future<Integer> future = threadPool.submit(new Callable<Integer>(){

public Integer call() throws Exception {

return new Random().nextInt(100);

}

});

try {

Thread.sleep(5000);// 替换其它事件执行时间

System.out.println(future.get());

} catch (Exception e) {

e.printStackTrace();

}

}

}

### RunnableFuture接口

它同时实现了RUnnable接口和Future接口

public interface RunnableFuture<V> extends Runnable, Future<V> {

void run();

}

### Executors类

Executors类是一个用于创建线程池的工厂类，这个类同时也提供了一些有用的静态方法。

因为Runnable接口没有返回值, 所以为了与Callable兼容, 我们额外传入了一个result参数, 使得返回的Callable对象的call方法直接执行Runnable的run方法, 然后返回传入的result参数。

底层：

public class Executors {

public static <T> Callable<T> callable(Runnable task, T result) {

if (task == null)

throw new NullPointerException();

return new RunnableAdapter<T>(task, result);

}

static final class RunnableAdapter<T> implements Callable<T> {

final Runnable task;

final T result;

RunnableAdapter(Runnable task, T result) {

this.task = task;

this.result = result;

}

public T call() {

task.run();

return result;

}

}

}

### ExecutorService介绍

ExecutorService继承自Executor，它的目的是为我们管理Thread对象，从而简化并发编程，Executor使我们无需显示的去管理线程的生命周期，是JDK5之后启动任务的首选方式。

AtomicLong使用:

CountDownLatch使用: new CountDownLatch(1).await();

# Volatile关键字

## 特点

记住两点：内存可见性和禁止指令重排序。

1）修饰变量。

2）不会产生工作内存的拷贝，直接与主内存交互，这样保证可见性但不保证原子性。原子性问题需synchronized或者CAS相关操作配合进行。

3）禁止进行指令重排序。

4）类似于volatile++这种复合操作不具备原子性，其操作等价于：int i = get(); i = i+1; set(i);

5）volatile变量是一个更轻量级的同步机制，因为在使用这些变量时不会发生上下文切换和线程调度等操作。

6）每次读volatie字段都会看到其它线程写入该字段的最新值。定义volatile字段是因为在某些情况由于缓存和重排序所看到的陈旧的变量值是不可接受的。

volatile保证共享变量的内存可见性原因：

volatile不会产生工作内存的拷贝，直接与主内存交互，这样保证可见性但不保证原子性。

volatile重排序规则：

读操作：前一行为volatile时，这两行代码不会发生重排序；和后一行代码都不会发生重排序。

写操作：和前一行代码都不会发生重排序；后一行代码为volatile时，这两行代码都不会发生重排序。

## 使用场景

-1）对变量的写操作不依赖于当前值。如i++，它是一个由读取－修改－写入操作序列组成的组合操作，volatile不具有原子特性。

-2）该变量没有包含在具有其他变量的不变式中。如low<up

-3）volatile最适用一个线程写，多个线程读的场合。如果有多个线程并发写操作，仍然需要使用锁或者线程安全的容器或者原子变量来代替。参考hashMap中fail-fast机制(transient volatile int modCount)

-4）结合使用volatile和synchronized实现“开销较低的读-写锁”。实际上，这些条件表明，可以被写入volatile变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

## 实例

实例一：单例模式中的双重检查锁定

//volatile是为了禁止指令重排序

private static volatile Singleton single=null;

--缺点描述: <https://segmentfault.com/a/1190000015549706#articleHeader8> ！！

## 可以创建volatile数组吗？

可以，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到volatile保护；但是如果改变数组中的元素，volatile标识符就不能起到之前的保护作用了。

## volatile能使得一个非原子操作变成原子操作吗?

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

## volatile类型变量提供了什么保证？

volatile主要有两方面的作用：1）避免指令重排序 2）可见性保证

JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。

volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。

某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低32位和高32位)，但 volatile 类型的 double 和 long 就是原子的.

# synchronized关键字

## 特点

修饰方法、块。一个线程在获取到监视器锁以后才能进入synchronized控制的代码块。

synchronized其实是一种悲观锁，线程持有该锁时会采用独占的方式访问这些变量。

synchronized在jdk1.6之后，已经改进优化。synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

底层实现：

-1）进入时，执行monitorenter，将计数器+1，释放锁monitorexit时，将计数器-1。

-2）当一个线程判断到计数器为0时，则当前锁空闲，可以占用；反之当前线程进入等待状态。

monitor机制：

synchronzed是在加对象锁。对象锁是一种重量级（monitor），synchronized的锁机制会根据线程竞争情况在运行时有偏向锁（单一线程）、轻量锁（多个线程访问synchronzed区域）、对象锁（重量锁，多个线程存在竞争的情况）、自旋锁等等。

synchronized本质：

线程欲进入synchronized时，会执行以下操作：

-1）强制写入主存储器（main memory）

当线程欲进入synchronized时，如果该线程的工作存储器（working memory）上有未映像到主存储器的拷贝，则这些内容会强制写入主存储器（store->write），则这些计算结果就会对其它线程可见（visible）

-2）工作存储器（working memory）的释放

当线程欲进入synchronized时，工作存储器上的工作拷贝会被全部丢弃。之后，欲引用主存储器上的值的线程，必定会从主存储器将值拷贝到工作拷贝（read->load）

线程欲退出synchronized时，会执行以下操作：

-1）强制写入主存储器（main memory）

当线程欲退出synchronized时，如果该线程的工作存储器（working memory）上有未映像到主存储器的拷贝，则这些内容会强制写入主存储器（store->write），则这些计算结果就会对其它线程可见（visible）。

# ThreadLocal类

## 相关概念

ThreadLocal用于创建线程的本地变量。目的是为了线程隔离，让每个线程都能拥有属于自己的变量空间，线程之间互不影响。副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

ThreadLocal是线程局部变量，即仅能被本线程访问，不能在线程之间进行共享访问的变量。

共享变量存在在主内存，线程要对主内存中的内容进行操作时，首先需要拷贝到自己的工作内存，因此这些变量不是线程安全的。当我们不希望其它线程共享它时，我们可以使用ThreadLocal。

当使用ThreadLocal维护变量时，其为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不会影响其它线程对应的副本。

构造方法：

new ThreadLocal()<T>; //创建一个线程本地变量

常用方法：

void set(T value); //将此线程局部变量的当前线程副本中的值设置为指定值

T get(); //返回此线程局部变量的当前线程副本中的值

void remove(); //移除此线程局部变量当前线程的值

protected T initalValue(); //返回此线程局部变量的当前线程的“初始值”。是protected方法，是为了让子类继承而设计的。

实例：

--统计某个方法的耗时时间：

public class ThreadLocalTest {

// static final类型，保证该变量不能指向其它对象

private static final ThreadLocal<Long> TIME\_THREADLOCAL = new ThreadLocal<Long>();

public static final void start() {

TIME\_THREADLOCAL.set(System.currentTimeMillis());

}

public static final long end() {

return System.currentTimeMillis() - TIME\_THREADLOCAL.get();

}

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

// AOP切入

ThreadLocalTest.start(); //执行方法前

TimeUnit.SECONDS.sleep(2); //执行方法

System.out.println(ThreadLocalTest.end()); //执行方法后

}

}

## 应用场景

实例一：

数据库连接：在多线程中，如果使用懒汉式的单例模式创建Connection对象，由于该对象是共享的，那么必须要使用同步方法保证线程安全，这样当一个线程在连接数据库时，那么另外一个线程只能等待，这样做的性能会降低。如果改为哪里要连接数据库就来进行连接，那么就会频繁的对数据库进行连接，性能还是不高。这时使用ThreadLocal既可以保证线程安全又可以保证性能。但ThreadLocal的缺点是占用了较多的空间。

例如：通过ThreadLocal保存调用线程相关的sesion信息

--这个session在不同线程中是不同的，但在同一线程中是相同的

private static final ThreadLocal threadSession = new ThreadLocal();

public static Session getSession() throws InfrastructureException {

Session s = (Session) threadSession.get(); //得到session

try {

if(s == null) {

s = getSessionFactory().openSession();

threadSession.set(s); //保存session

}

}catch (HibernateException ex) {

throw new InfrastructureException(ex);

}

return s;

}

## 原理

ThreadLocal并不是一个Thread，而是Thread的局部变量，相当于一个Map集合，key为当前线程。

在Thread中有一个成员变量ThreadLocals，该变量的类型是ThreadLocalMap，它的键是ThreadLocal、值就是变量的副本。通过ThreadLocal的get()方法可以获取该线程变量的本地副本，在get方法之前要先set。否则就要重写initialValue()方法。

线程的角度看，每个线程都保持一个对其线程局部变量副本的隐式引用，只要线程是活动的并且ThreadLocal实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被垃圾回收。

### set方法描述：

--相当于map.put(Thread.currentThread(), value);

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

### get方法描述：

--调用 ThreadLocalMap.getEntry()方法，以当前调用get方法的ThreadLocal对象为key值，获取对应的value值。

public T get() {

//获取当前线程的ThreadLocalMap对象

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null) {

ThreadLocalMap.Entry e = map.getEntry(this);

if (e != null) {

T result = (T)e.value;

return result;

}

}

return setInitialValue(); //初始的value值为null

}

## 父子线程

ThreadLocal和InheritableThreadLocal解决了单线程环境和在单线程中又创建线程（父子线程）中线程隔离的问题。

TransmittableThreadLocal主要解决了线程池中线程复用的场景。

测试:

--当我们创建一个线程,在线程内有去运行另一个线程的时候，作为子线程，如何去拿到父线程的私有属性呢?

public static void subAndThreadLocalTest() {

ThreadLocal t1 = new ThreadLocal<String>();

new Thread(new Runnable() {

@Override

public void run() {

t1.set("父线程私有变量");

System.out.println("父线程" + Thread.currentThread().getName() + ":" + t1.get());

new Thread(new Runnable() {

System.out.println("子线程" + Thread.currentThread().getName() + ":" + t1.get());

}).start();

}

}).start();

}

输出：

父线程Thread-0:父线程私有变量

子线程Thread-1:null

结果分析：

子线程并不能输出父线程的局部变量

# CAS算法

## 相关概念

compare and swap，翻译为比较和交换。

当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

CAS不通过jvm，利用CPU的cmpxchg指令（汇编指令），同时借助Java本地方法JNI来完成Java的非阻塞算法，实现原子操作。

整个java.util.concurrent都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

## CAS缺点

-1）许多线程反复尝试更新某一变量却不成功，这种循环反复的操作会给CPU带来很大的压力。

-2）CAS机制只能保证一个变量的原子操作，不能保证整个代码块的原子性。

-3）ABA问题

使用场景：使用CAS在线程冲突严重时，会大幅降低程序性能；CAS只适合于线程冲突较少的情况使用。

ABA问题描述：

-1）账号有100块钱，由于提款机硬件存在小问题，提款操作被同时提交了两次，开启了两个线程，这两个线程都是获取当前值A为100元，要更新值B为50元。

-2）理想情况下，应该是一个线程更新成功，另一线程更新失败。

-3）线程1首先执行成功，把余额从100变成50，线程2因为某种原因阻塞。

-4）这时候，你妈妈刚好给你汇款50元，开启线程3。线程2仍然阻塞状态，而线程3执行成功，把余额50变成100。

-5）线程2恢复运行，由于比较实际值也是100元，所以成功把变量值100更新为50。

分析：原本线程2应当提交失败，余额应该保持为100元，结果由于ABA问题提交成功了。

解决：利用版本号versionNumber比较可以有效解决ABA问题

CAS原子操作在concurrent包的实现：？？？

## 使用

cas机制有3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存地址V对应的值相同时，才将内存地址V对应的值修改为B，否则什么都不做。

CAS是个乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

## cas自旋

cas的自旋，在循环中做了三件事：

-1）获取当前值

-2）当前值+1

-3）进入cas操作，若成功则跳出循环，否则重复上述步骤

例如在AtomicInteger中incrementAndGet()方法：

pulic final int incrementAndGet() {

for (;;) { //cas自旋

int current = get();

int next = current + 1;

if (compareAndSet(current, next)) {

return next;

}

}

}

get()方法；

获取变量的最新值。使用volatile关键字能保证获取的当前值是内存中的最新值!

compareAndSet()方法：

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

unsafe属性：为我们提供了硬件级别的原子操作。

valueOffset对象：通过unsafe.objectFieldOffset方法得到，所代表的是AtomicInteger对象的成员变量value在内存中的偏移量。我们可以简单把valueOffset理解成value变量的内存地址。

cas机制中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。而unsafe的compareAndSwapInt方法参数包括了3个基本元素：valueOffset代表了V，expect代表了A，update代表了B。

## Unsafe介绍

这个类最大的特点在于，它提供了硬件级别的CAS原子操作。

CAS操作包含了三个操作数： 需要读写的内存位置，进行比较的原值，拟写入的新值。

在Unsafe类中，实现CAS操作的方法是： compareAndSwapXXX

例如：

// obj-目标对象 offset-对应属性的内存偏移量 expect-进行比较的原值 update-拟写入的新值

public native boolean compareAndSwapObject(Object obj, long offset, Object expect, Object update);

该方法实现了对目标对象obj中的某个成员变量（field）进行CAS操作的功能。

//获取目标field的内存偏移量offset

public native long objectFieldOffset(Field field);

//获取field对象

Class<?> k = FutureTask.class;

Field stateField = k.getDeclareField("state");

除了compareAndSwapObject，Unsafe类还提供了更为具体的对int和long类型的CAS操作：

public native boolean compareAndSwapInt(Object obj, long offset, int expect, int update);

public native boolean compareAndSwapLong(Object obj, long offset, long expect, long update);

从方法签名可以看出，这里只是把目标field的类型限定成int和long类型，而不是通用的Object.

赋值操作：

// 设置obj对象中offset偏移地址对应的int类型的field的值为指定值。这其实是Unsafe的另一个方法putIntVolatile的有序或者有延迟的版本，并且不保证值的改变被其他线程立即看到

public native void putOrderedInt(Object obj, long offset, int value);

// 设置obj对象中offset偏移地址对应的整型field的值为指定值，支持volatile store语义。

public native void putIntVolatile(Object obj, long offset, int value);

https://segmentfault.com/a/1190000016542779 未完成！！

# 线程池

相关教程：

https://www.cnblogs.com/dolphin0520/p/3932921.html

## 概念相关

定义：

在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。

java.util.concurrent.ThreadPoolExecutor类就是一个线程池。客户端调用submit(Runnable task)提交任务。

《阿里巴巴Java手册》

【强制】线程资源必须通过线程池体提供，不允许在应用中显式创建线程。

使用线程池的**好处**：减少在创建和销毁线程上所花的时间及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致内存或者“过度切换”的问题。

目的：

线程是稀缺资源，不能频繁的创建；

解耦作用：线程的创建与执行完全分开，方便维护；

应当将其放入一个池子中，可以给其它任务进行复用。

问题：

使用线程来执行异步任务时，若每个任务都创建一个新的线程去执行时存在一下问题：

1）线程的创建和销毁需要一定的开销；

2）可能会使应用处于高负荷状态而最终崩溃

解决：

在线程池中创建若干条线程。

1）有任务执行时就从线程池中获取一条线程来执行任务；

2）若某时间内任务数量超出线程池的线程数时，则后面的线程任务就进入一个等待队列等待，直到线程池有线程处于空闲时才从等待队列获取要执行的任务进行处理。

**好处**：大大减少了线程创建和销毁的开销，也会缓解我们的应用处于超负荷时的情况。

## 线程池的创建

jdk1.5引入Executors线程池框架，通过它把任务的提交和执行进行解耦，只需要定义好任务，然后提交给线程池，而不用关心该任务是如何执行、被哪个线程执行，及什么时候执行。

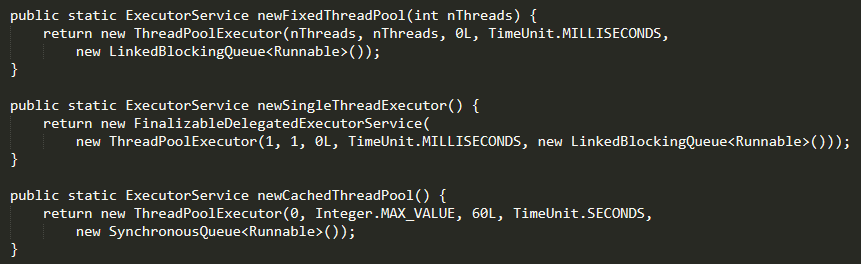
Java推荐使用Executors类提供的3个静态方法创建线程池：

1）Executors.newCachedThreadPool() //创建一个缓存池，缓存池容量大小为Integer.MAX\_VALUE。创建一个可根据需要创建新线程的线程池，若以前构造的线程可用则重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。

2）Executors.newFixedThreadPool(nThreads) //创建固定容量大小的缓冲池。创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

3）Executors.newSingleThreadExecutor() //创建容量为1的缓存池(注意它命名跟其它的不一样...)。创建一个使用单个工作线程(worker)的Executor，以无界队列的方式来运行该线程。

这三个静态方法的具体实现如下：（实际上是利用ThreadPoolExecutor类实现的）



分析：

-1）newFixedThreadPool中，corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue

-2）newSingleThreadExecutor中，corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue

-3）newCachedThreadPool中，corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

--另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

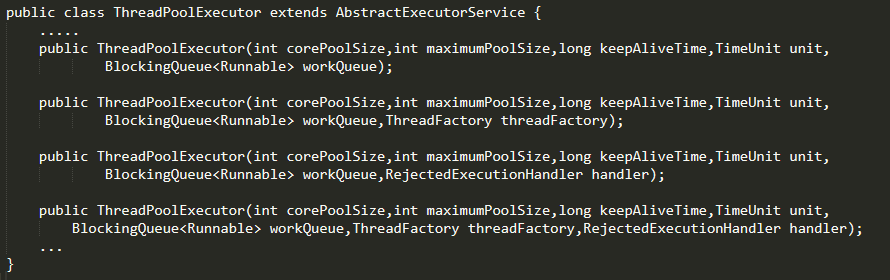
线程池中的线程初始化：

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。

## ThreadPoolExecutor类

### 构造方法

java.util.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类。有4个构造方法：



参数说明：

-1）corePoolSize：核心线程数

核心线程会一直存活，即使没有任务需要执行。

当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建线程处理。

默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，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，就会把到达的任务放到缓存队列中。

设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭。

-2）maximumPoolSize：最大线程数

当线程数>=corePoolSize，且任务队列已满时。线程池会创建新线程来处理任务

当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常

-3）keepAliveTime：线程空闲后保持存活时间

当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。

当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize

如果allowCoreThreadTimeout=true，则会直到线程数量=0

-4）workQueue：阻塞队列，用来存储等待执行的任务。

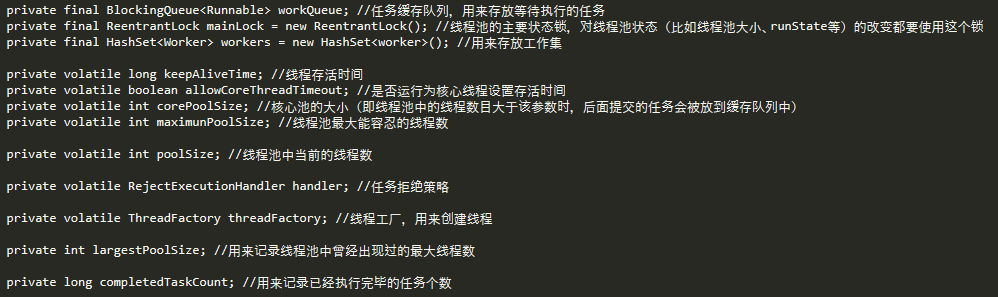
当核心线程数达到最大时，新任务会放在队列中排队等待执行。

-5）threadFactory：线程工厂，主要用来创建线程

-6）handler：拒绝处理任务的策略

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略。

### 成员变量



核心变量说明：

--corePoolSize、maximumPoolSize、largestPoolSize三个变量描述：

例子说明：假如有一个工厂，工厂里面有10个工人，每个工人同时只能做一件任务。因此只要当10个工人中有工人是空闲的，来了任务就分配给空闲的工人做；当10个工人都有任务在做时，如果还来了任务，就把任务进行排队等待；如果说新任务数目增长的速度远远大于工人做任务的速度，那么此时工厂主管可能会想补救措施，比如重新招4个临时工人进来；然后就将任务也分配给这4个临时工人做；如果说着14个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。当这14个工人当中有人空闲时，而新任务增长的速度又比较缓慢，工厂主管可能就考虑辞掉4个临时工了，只保持原来的10个工人，毕竟请额外的工人是要花钱的。这个例子中的corePoolSize就是10，而maximumPoolSize就是14（10+4）

### 执行顺序

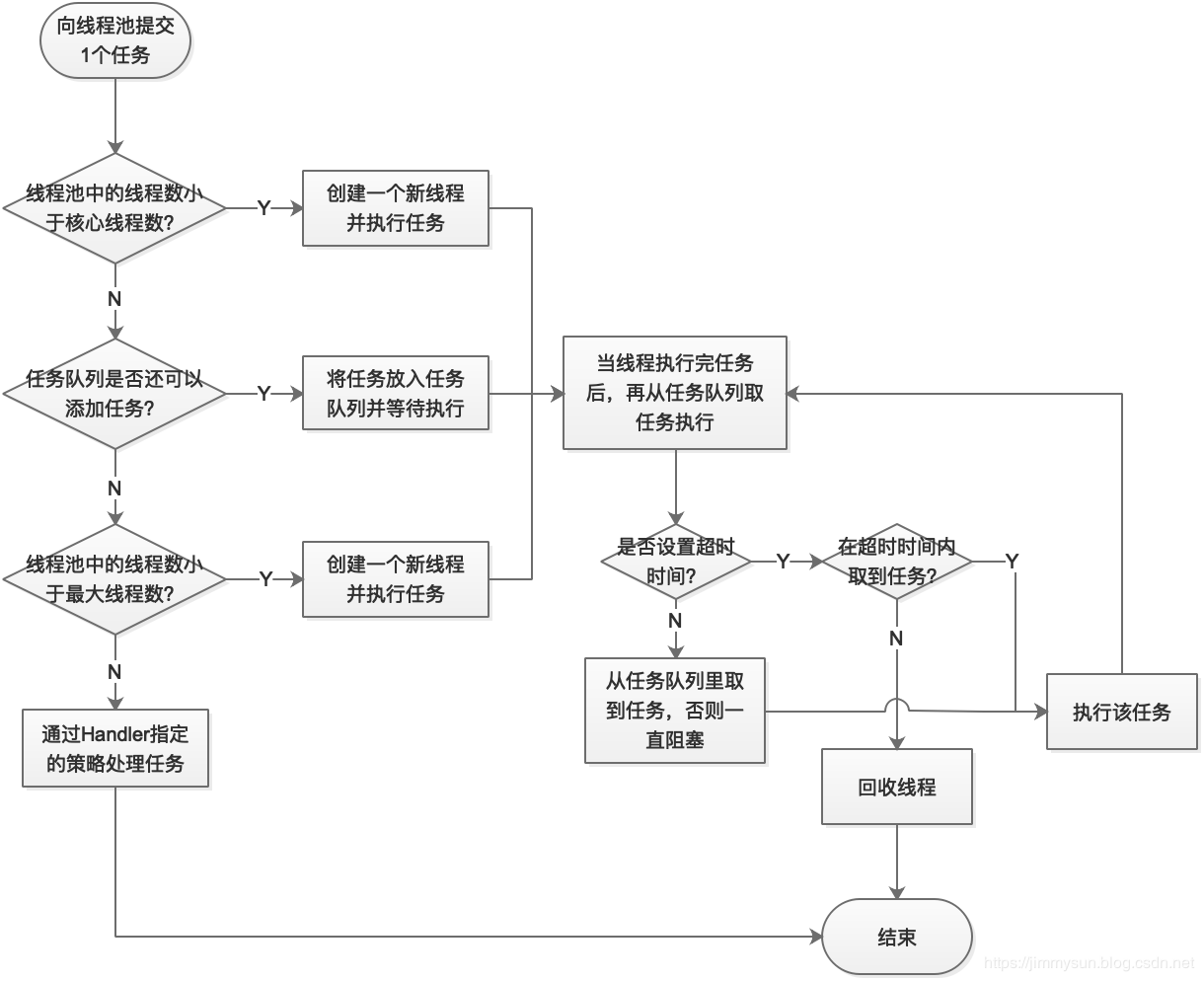
-1）当线程数小于核心线程数时，创建线程，即使池中有线程空闲，也会优先创建线程处理任务；

-2）当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列；

-3）当线程数大于等于核心线程数，且任务队列已满：1.若线程数小于最大线程数，创建线程；2.若线程数等于最大线程数，抛出异常，拒绝任务。

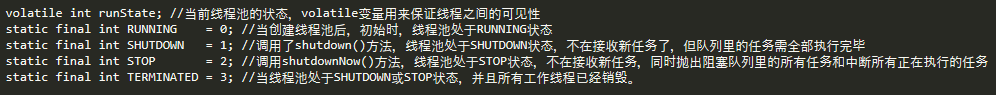
流程图：

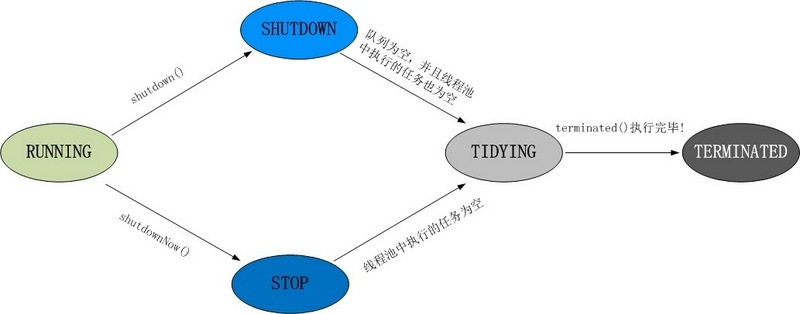
详情：<https://blog.csdn.net/u013541140/article/details/95225769>



### 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：





### 拒绝策略

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略。

分类（4种）：

-1）ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

-2）ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

-3）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）

-4）ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

### 阻塞队列

当核心线程数达到最大时，新任务会放在队列中排队等待执行。

分类：

-1）ArrayBlockinfQueue:基于数组实现，先进先出队列，创建时必须设定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。

-2）LinkedBlockingQueue:基于链表实现，先进先出队列，创建时可不需设置容量大小，默认为Integer.MAX\_VALUE.

-3）PriorityBlockingQueue:以上2种队列都是先进先出队列，而PriorityBlockingQueue是按照元素的优先级对元素进行排序，按照优先级顺序出队，每次出队的元素都是优先级最高的元素。另，PriorityBlockingQueue阻塞队列是无界阻塞队列，即大小无上限，而前面2种都是有上限的

-4）DelayQueue:基于PriorityQueue，是一种延时阻塞队列，DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue也是一个无界队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

-5）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

注：ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

### 提交任务到线程池

threadPool.execute(new Job());

execute()方法描述：

-1）如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

-2）如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

-3）如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

-4）如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

### 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()。

1）shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务。

2）shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务。

### 线程池容量的动态调整

hreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()。

### 测试实例

--直接使用ThreadPoolExecutor创建线程池测试

--1）编写线程类

class Test1Task implements Runnable {

private int taskNum;

public Test1Task(int i) {

taskNum = i;

}

@Override

public void run() {

System.out.println("正在执行线程" + taskNum);

try {

Thread.currentThread().sleep(4000);

} catch (Exception e) {

e.printStackTrace();

}

System.out.println("线程" + taskNum + "执行完毕");

}

}

--2）测试类

public class Test1 {

public static void main(String[] args) {

ThreadPoolExecutor exec = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MICROSECONDS, new ArrayBlockingQueue<Runnable>(5));

for (int i = 0; i < 15; i++) {

Test1Task task = new Test1Task(i);

exec.execute(task);

System.out.println("线程池中线程数" + exec.getPoolSize() + "，等待队列线程数："+exec.getQueue().size() + "，已执行完的任务数："+exec.getCompletedTaskCount());

}

exec.shutdown();

}

}

--3）输出

…//省略

--4）结果分析

当线程池中线程额数目大于5时，便将任务放入到任务缓存队列中，当任务缓存队列满了之后，便创建新的线程。如果上面程序中，将for循环中改成执行20个任务，就会抛出任务拒绝异常了。

## CachedthreadPool线程池

测试实例：

--1）定义线程类(实现Runnable接口)：

public class MyThread implements Runnable {

private int count = 1, number;

public MyThread(int num) {

number = num;

System.out.println("创建线程-" + number);

}

public void run() {

public void run() {

while(++count != 3) { //运行两次后退出

System.out.println("线程-" + number + "运行" + count + "次");

}

}

}

--2）编写测试类

public class CachedThreadPoolTest {

public static void main(String[] args) {

ExecutorService exec = Executors.newCachedThreadPool();

for(int i = 0; i < 5; i++) {

exec.execute(new MyThread(i));

}

exec.shutdown();

}

}

--3）测试结果

…//省略

## 功能线程池

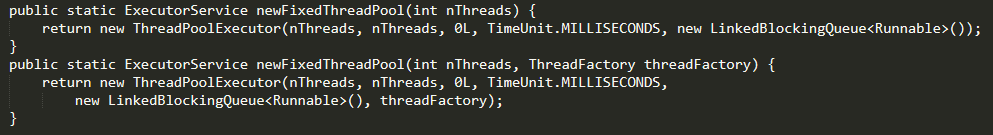
详情：https://blog.csdn.net/u013541140/article/details/95225769

Executor已经为我们封装了4种常见的功能线程池：

定长线程池（FixedThreadPool）、定时线程池（ScheduledThreadPool）、可缓存线程池（CachedThreadPool）、单线程化线程池（SingleThreadExecutor）

### 定长线程池（FixedThreadPool）

创建方法源码：



特点：只有核心线程数，线程数量固定，执行完立即回收，任务队列链表结构的有界队列。

应用场景：控制线程最大并发数。

测试：

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

Runnable task = new Runnable() {

public void run() {

System.out.println(Thread.currentThread().getName() + "::执行任务");

}

};

fixedThreadPool.execute(task); // pool-1-thread-1::执行任务

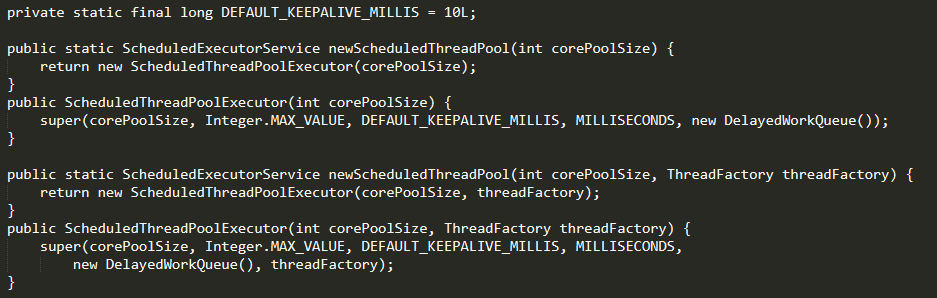
fixedThreadPool.execute(task); // pool-1-thread-3::执行任务

fixedThreadPool.execute(task); // pool-1-thread-1::执行任务

fixedThreadPool.execute(task); // pool-1-thread-2::执行任务

### 定时线程池（ScheduledThreadPool）

创建方法的源码：



特点：核心线程数量固定，非核心线程数量无限，执行完闲置10ms后回收，任务队列为延时阻塞队列。

应用场景：执行定时或周期性的任务。

测试：

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

Runnable task = new Runnable() {

public void run() {

System.out.println(Thread.currentThread().getName() + "：：执行任务");

}

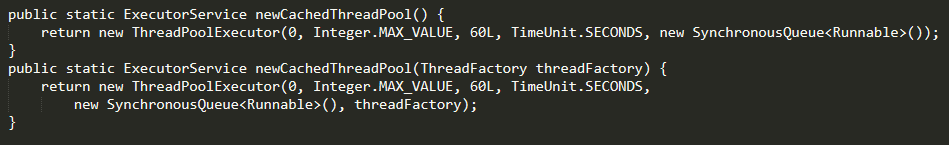
};

scheduledThreadPool.schedule(task, 1, TimeUnit.SECONDS);//延迟1s后执行

scheduledThreadPool.scheduleAtFixedRate(task, 10, 1000, TimeUnit.MILLISECONDS);//延迟10ms后、每隔1000ms执行任务

### 可缓存线程池（CachedThreadPool）

创建方法的源码：



特点：无核心线程，非核心线程数量无限，执行完闲置60s后回收，任务队列为不存储元素的阻塞队列。

应用场景：执行大量、耗时少的任务

测试：

ExecutorService fixedThreadPool = Executors.newCachedThreadPool();

Runnable task = new Runnable() {

public void run() {

System.out.println(Thread.currentThread().getName() + "::执行任务");

}

};

fixedThreadPool.execute(task); // pool-1-thread-1::执行任务

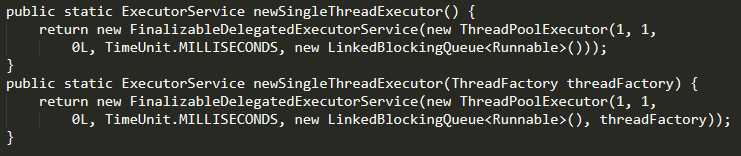
fixedThreadPool.execute(task); // pool-1-thread-4::执行任务

fixedThreadPool.execute(task); // pool-1-thread-3::执行任务

fixedThreadPool.execute(task); // pool-1-thread-2::执行任务

### 单线程化线程池（SingleThreadExecutor）

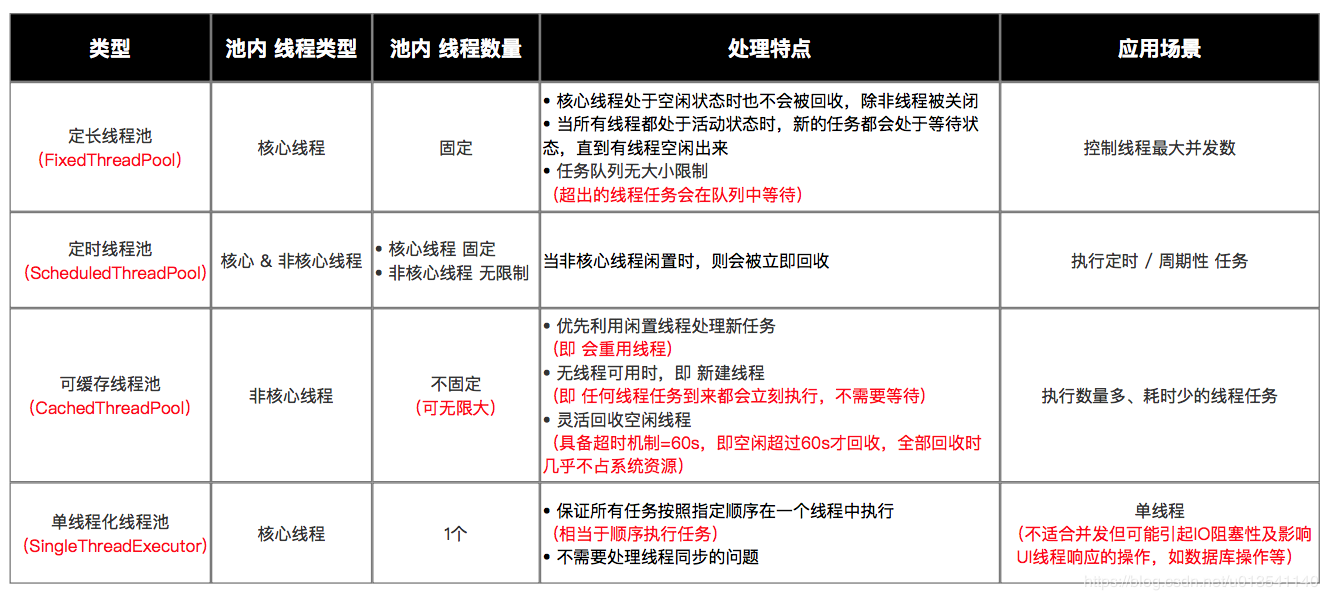
创建方法的源码：



特点：只有1个核心线程，无非核心线程，执行完立即回收，任务队列为链表结构的有界队列。

应用场景：不适合并发但可能引起IO阻塞性及影响UI线程响应的操作，如数据库操作、文件操作等。

### 4种线程池对比



Executors的4个功能线程池虽然方便，但现在已经不建议使用了，而是建议直接通过使用ThreadPoolExecutor的方式。

FixedThreadPool和SingleThreadExecutor：主要问题是堆积的请求处理队列均采用LinkedBlockingQueue，可能会耗费非常大的内存，甚至OOM。

CachedThreadPool和ScheduledThreadPool：主要问题是线程数最大数是Integer.MAX\_VALUE，可能会创建数量非常多的线程，甚至OOM。

# 读写锁原理:

读写锁，即存在两把锁，可以用AQS的同步状态标识其中的一把锁，再引入一个新的属性表示另一把锁，但是这样做就变成了二元并发安全问题，使问题变得更加复杂。

ReentrantReadWriteLock采用“按位切割”的方式，将32位的int型state变量分为高16位和低16位，高16位代表读状态，低16位代表写状态，读锁共享、写锁互斥。

对于写锁，用低16位表示线程的重入次数；而读锁因为可以同时有多个线程，所以重入次数需要通过其它方式来记录，那就是ThreadLocal变量。

假设状态变量是c：

1）读状态就是c>>>16（无符号右移16位），即抹掉低的16位，剩下的就是c的高16位；

2）写状态是c&((1 << 16) -1)，即c&00000000000000001111111111111111，与运算之后，高16位被抹掉，剩下的就是c的低16位。

3）如果读线程申请读锁，当前写锁重入次数不为0时，则等待，否则可以马上分配；如果是写线程申请写锁，当前状态为0则可以马上分配，否则等待。

2.读锁的获取和释放

https://www.toutiao.com/a6600608017446601223/?tt\_from=mobile\_qq&utm\_campaign=client\_share&timestamp=1536853538&app=news\_article\_lite&utm\_source=mobile\_qq&iid=39241532035&utm\_medium=toutiao\_android&group\_id=6600608017446601223 --未完成