**引言**

在讲线程池之前，我们得先说一下线程是什么，它是一种轻量级的进程，它的创建，运行，销毁都是是需要消耗计算机资源的。当有多个任务来临时，我们得为每个任务创建线程，然后去处理。这种情况多发生于服务器应用程序上，它会接收到很多的请求，如果为每个请求都创建一个线程去处理，那么肯定会造成一种现象，为每个请求创建新线程的服务器，在创建和销毁线程上花费的时间和消耗的系统资源要比花在处理实际的用户请求的时间和资源更多，并且频繁的创建，销毁，会导致JVM不断地切换线程,从而引发内存溢出，泄漏等一系列问题。所以，我们急需一种能避免频繁的创建线程且能限制请求数的东西。线程池，应运而生，它就能很好的解决了这个问题，它限制了处理请求的线程数，并且能够维持线程的存活，避免频繁的创建销毁造成的开销，还能自我调度线程去执行任务，可以说是非常nice了。

首先让我们看看线程池里面的一些重要参数：

private final AtomicInteger ctl = new AtomicInteger(*ctlOf*(*RUNNING*, 0));

你可以把这个“ctl”理解成一个线程池，它的前三位表示线程池的状态，后29位表示线程数。通过对这个数的改变，就可以控制整个线程池。

在这里我们应该知道几个点：

1. 怎么处理并发操作的问题？

答：当线程池创建了多个线程时，每个线程都会改变线程池的内部状态，那就避不开并发的问题。因此，线程池采用了原子类AtomicInteger来修饰这个ctl,原子类是用CAS机制实现的，而CAS又是通过调用Native方法，直接使用CAS指令去修改这个数，避免了并发的问题。

1. 线程池的五种状态：

RUNNING=-1：该状态下会接受新的任务，也会处理阻塞队列的任务

SHUTDOWN=0： 该状态下不会接受新的任务，但会处理阻塞队列的任务

STOP=1：该状态下不会接受新的任务，也不会处理阻塞队列的任务，还会中断运行中的任务

TIDYING=2：所有任务都被终止了，workerCount为0，将调用terminated()方法

TERMINATED=3：调用完termianted()方法后变成该状态

以上状态也基本上对应线程池了从运行到终止的状态。

1. 会常用到的工具方法

runStateOf(int c):获取线程池c的状态

workCountOf(int c);获取线程池c的线程数量

ctlOf(int rs,int wc):根据rs(状态)与wc(线程数)打包合成一个ctl

private volatile ThreadFactory threadFactory;//线程工厂

线程池中线程的创建由该类创建(类比工厂模式)

private volatile RejectedExecutionHandler handler;//拒绝策略

线程池的线程无法接受任务时的一种策略，共四种

1. 丢弃任务，并抛出异常。
2. 丢弃任务，不抛异常。
3. 交由调入该任务的线程处理该任务。
4. 移除阻塞队列中等待时间最长的任务，并将当前任务放到队尾，再重新尝试执行任务

private final BlockingQueue<Runnable> workQueue;//阻塞队列

用来存放需要调用的任务，线程池自行获取，而不需要复杂的调度策略。

//线程队列

private final HashSet<Worker> workers = new HashSet<Worker>();

private int largestPoolSize;//记录曾达到过的最大的线程数

private long completedTaskCount;//记录已完成的任务数

private volatile long keepAliveTime;//线程空闲时的最大存活时间

该参数默认只对非核心线程生效，意指线程在没有任务执行时，最大能存活的时间。

//让keepAliveTime对核心线程是否有效的参数

private volatile boolean allowCoreThreadTimeOut;

通过设置该参数可以使最大存活时间对核心线程生效

private volatile int corePoolSize;//核心线程数

private volatile int maximumPoolSize;//最大线程数

当阻塞队列满了，才会开始创建该线程

现在，我们知道了线程池中这些参数的意义，那么让我们一起来看看线程池里面**线程**和**任务**到底是怎样运作的？

首先，我们通过观看源码，知道了线程池有四个构造器，并且很容易发现，前三个构造的方法体都是在调用第四个构造,其他参数都由线程池给默认值，通过这样的方式，让用户可以更灵活的构建线程池。（其他参数在前面都出现过，我在这里解释一下，其中的Timeunit unit是为了设置线程存活时间的一个时间单位）

**一.线程池执行方法：**

public void execute(Runnable command){

if (command == null)//检查任务是否为空

throw new NullPointerException();

int c = ctl.get();//获取当前线程池的状态

if (workerCountOf(c) < corePoolSize) {//小于核心线程池

if (addWorker(command, true))//创建新线程执行任务，添加一个员工

return;//成功 return

c = ctl.get();//失败 再次获取线程池的状态

}

//判断线程池是否是运行状态并且添加阻塞队列是否成功

if (isRunning(c) && workQueue.offer(command)) {

int recheck = ctl.get();//再次获得状态，用于后面的双重校验

//判断线程池是否是运行状态并且从阻塞队列移除成功

if (! isRunning(recheck) && remove(command))

reject(command);//未运行,移除成功，拒绝该任务

else if (workerCountOf(recheck) == 0)//检测当前工作的线程数是否为0

addWorker(null, false);//创建一个没有任务的非核心线程

}

else if (!addWorker(command, false))//用非核心线程接受该任务

reject(command);//失败拒绝

}

大致流程是：

1、如果线程池当前线程数量少于corePoolSize，则addWorker(command, true)创建新worker线程，如创建成功返回，如没创建成功，则执行后续步骤；  
    addWorker(command, true)失败的原因可能是：  
    A、线程池已经shutdown，shutdown的线程池不再接收新任务  
    B、workerCountOf(c) < corePoolSize 判断后，由于并发，别的线程先创建了worker线程，导致workerCount>=corePoolSize  
 2、如果线程池还在running状态，将task加入workQueue阻塞队列中，如果加入成功，进行double-check，如果加入失败（可能是队列已满），则执行后续步骤；  
    double-check主要目的是判断刚加入workQueue阻塞队列的task是否能被执行  
    A、如果线程池已经不是running状态了，应该拒绝添加新任务，从workQueue中删除任务  
    B、如果线程池是运行状态，或者从workQueue中删除任务失败（刚好有一个线程执行完毕，并消耗了这个任务），确保还有线程执行任务（只要有一个就够了）  
 3、如果线程池不是running状态 或者 无法入队列，尝试开启新线程，扩容至maxPoolSize，如果addWork(command, false)失败了，拒绝当前command

接下来，让我看看线程池中工作线程以什么样的形式存在？

二.Worker

private final class Worker extends AbstractQueuedSynchronizer

implements Runnable

{

private static final long serialVersionUID = 6138294804551838833L;

final Thread thread;

Runnable firstTask;

volatile long completedTasks;

Worker(Runnable firstTask) {

setState(-1);

this.firstTask = firstTask;

this.thread = getThreadFactory().newThread(this);

}

public void run() {

runWorker(this);

}

……

}

很明显，工作线程以内部类的形式存在于线程池中，它实现了Runnable接口，又继承了AbstractQueuedSynchronizer（以下简称AQS），所以表明它是一个可执行的任务，又可以达到锁的效果。（这里想要深刻的理解worker的工作方式，必须了解AQS的机制），我在这儿作简要分析，详细的知识还得读者自行了解。

1、将AQS的state置为-1，在runWoker()前不允许中断  
2、待执行的任务会以参数传入，并赋予firstTask  
3、用Worker这个Runnable创建Thread(调用工厂创建)

之所以Worker自己实现Runnable，并创建Thread，在firstTask外包一层，是因为要通过Worker控制中断，而firstTask这个工作任务只是负责执行业务

三.添加工作者(创建线程)addWorker():

private boolean addWorker(Runnable firstTask, boolean core) {

//外层循环，负责判断线程池状态

retry:

for (;;) {

int c = ctl.get();

int rs = runStateOf(c); //状态

// Check if queue empty only if necessary.

/\*\*

\* 线程池的state越小越是运行状态，runnbale=-1，shutdown=0,stop=1,tidying=2，terminated=3

\* 1、如果线程池state已经至少是shutdown状态了

\* 2、并且以下3个条件任意一个是false

\* rs == SHUTDOWN（隐含：rs>=SHUTDOWN）false情况： 线程池状态已经超过shutdown，可能是stop、tidying、terminated其中一个，即线程池已经终止

\* firstTask == null （隐含：rs==SHUTDOWN）false情况： firstTask不为空，rs==SHUTDOWN 且 firstTask不为空，return false，场景是在线程池已经shutdown后，还要添加新的任务，拒绝

\* ! workQueue.isEmpty()（隐含：rs==SHUTDOWN，firstTask==null）false情况： workQueue为空，当firstTask为空时是为了创建一个没有任务的线程，再从workQueue中获取任务，如果workQueue已经为空，那么就没有添加新worker线程的必要了

\* return false，即无法addWorker()

\*/

if (rs >= SHUTDOWN &&

! (rs == SHUTDOWN &&

firstTask == null &&

! workQueue.isEmpty()))

return false;

//内层循环，负责worker数量+1

for (;;) {

int wc = workerCountOf(c); //worker数量

//如果worker数量>线程池最大上限CAPACITY（即使用int低29位可以容纳的最大值）

//或者( worker数量>corePoolSize 或 worker数量>maximumPoolSize )，即已经超过了给定的//边界

if (wc >= CAPACITY ||

wc >= (core ? corePoolSize : maximumPoolSize))

return false;

//调用unsafe CAS操作，使得worker数量+1，成功则跳出retry循环

if (compareAndIncrementWorkerCount(c))

break retry;

//CAS worker数量+1失败，再次读取ctl

c = ctl.get(); // Re-read ctl

//如果状态不等于之前获取的state，跳出内层循环，继续去外层循环判断

if (runStateOf(c) != rs)

continue retry;

// else CAS failed due to workerCount change; retry inner loop

// else CAS失败时因为workerCount改变了，继续内层循环尝试CAS对worker数量+1

}

}

/\*\*

\* worker数量+1成功的后续操作

\* 添加到workers Set集合，并启动worker线程

\*/

boolean workerStarted = false;

boolean workerAdded = false;

Worker w = null;

try {

final ReentrantLock mainLock = this.mainLock;

w = new Worker(firstTask); //1、设置worker这个AQS锁的同步状态state=-1

//2、将firstTask设置给worker的成员变量firstTask

//3、使用worker自身这个runnable，调用ThreadFactory创建一个线程，并设置给worker的成员变量thread

final Thread t = w.thread;

if (t != null) {

mainLock.lock();

try {

//--------------------------------------------这部分代码是上锁的

// Recheck while holding lock.

// Back out on ThreadFactory failure or if

// shut down before lock acquired.

// 当获取到锁后，再次检查

int c = ctl.get();

int rs = runStateOf(c);

//如果线程池在运行running<shutdown 或者 线程池已经shutdown，//且firstTask==null（可能是workQueue中仍有未执行完成的任务，创建没有初始任务的//worker线程执行）

//worker数量-1的操作在addWorkerFailed()

if (rs < SHUTDOWN ||

(rs == SHUTDOWN && firstTask == null)) {

if (t.isAlive()) // precheck that t is startable 线程已经启动，抛非//法线程状态异常

throw new IllegalThreadStateException();

workers.add(w);//workers是一个HashSet<Worker>

//设置最大的池大小largestPoolSize，workerAdded设置为true

int s = workers.size();

if (s > largestPoolSize)

largestPoolSize = s;

workerAdded = true;

}

}

finally {

mainLock.unlock();

}

//如果往HashSet中添加worker成功，启动线程

if (workerAdded) {

t.start();

workerStarted = true;

}

}

} finally {

//如果启动线程失败

if (! workerStarted)

addWorkerFailed(w);

}

return workerStarted;

}

addWorker()总共有四种方式：

addWorker(command,true);//将command任务以核心线程运行

addWorker(command,false);//将command任务以非核心线程线程

addWorker(null,true);//创建一个核心线程

addWorker(null,true);//创建一个非核心线程

执行流程：

1、判断线程池当前是否为可以添加worker线程的状态，可以则继续下一步，不可以return false：  
    A、线程池状态>shutdown，可能为stop、tidying、terminated，不能添加worker线程  
     B、线程池状态==shutdown，firstTask不为空，不能添加worker线程，因为shutdown状态的线程池不接收新任务  
    C、线程池状态==shutdown，firstTask==null，workQueue为空，不能添加worker线程，**因为firstTask为空是为了添加一个没有任务的线程再从workQueue获取task**，而workQueue为空，说明添加无任务线程已经没有意义(**从这一步就可以看出线程池不是通过调度任务来执行的，而是通过线程主动去阻塞队列取任务来执行的**)  
 2、线程池当前线程数量是否超过上限（corePoolSize 或 maximumPoolSize），超过了return false，没超过则对workerCount+1，继续下一步  
 3、在线程池的ReentrantLock保证下，向Workers Set中添加新创建的worker实例，添加完成后解锁，并启动worker线程，如果这一切都成功了，return true，如果添加worker入Set失败或启动失败，调用addWorkerFailed()逻辑

四.工作者运行方法：

final void runWorker(Worker w) {

Thread wt = Thread.currentThread();//获得当前线程

Runnable task = w.firstTask;//获取任务

w.firstTask = null;

w.unlock(); //new Worker() state==-1，此处是调用Worker类的tryRelease()

//方法，将state置为0，而interruptIfStarted()中只有

//state>=0才允许调用中断

boolean completedAbruptly = true; //是否“突然完成”，如果是由于异常

//导致的进入finally，那么//completedAbruptly==true就是突然完成的

try {

while (task != null || (task = getTask()) != null) {

w.lock();//加锁

if ((runStateAtLeast(ctl.get(), STOP) ||

(Thread.interrupted() &&

runStateAtLeast(ctl.get(), STOP))) &&

!wt.isInterrupted())

wt.interrupt();//当前线程调用interrupt()中断

try {

//执行前（子类实现）

beforeExecute(wt, task);

Throwable thrown = null;

try {

task.run();

} catch (RuntimeException x) {

thrown = x; throw x;

} catch (Error x) {

thrown = x; throw x;

} catch (Throwable x) {

thrown = x; throw new Error(x);

} finally {

afterExecute(task, thrown);

}

} finally {

task = null;

w.completedTasks++;//任务书加1

w.unlock();

}

}

completedAbruptly = false;

} finally {

   //处理worker的退出

processWorkerExit(w, completedAbruptly);

}

}

执行流程：

1.Worker线程启动后，通过Worker类的run()方法调用runWorker(this)  
2.执行任务之前，首先worker.unlock()，将AQS的state置为0，允许中断当前worker线程  
3.开始执行firstTask，调用task.run()，在执行任务前会上锁wroker.lock()，在执行完任务后会解锁，为了防止在任务运行时被线程池一些中断操作中断  
4.在任务执行前后，可以根据业务场景自定义beforeExecute() 和 afterExecute()方法  
5.无论在beforeExecute()、task.run()、afterExecute()发生异常上抛，都会导致worker线程终止，进入processWorkerExit()处理worker退出的流程  
6.如正常执行完当前task后，会通过getTask()从阻塞队列中获取新任务，当队列中没有任务，且获取任务超时，那么当前worker也会进入退出流程。

 线程池里面的方法还有很多，我在这里就不赘述了，感兴趣的读者可以下去自行分析。

Java提供了四种线程池：

**CachedThreadPool()**

可缓存线程池：

1. 线程数无限制
2. 有空闲线程则复用空闲线程，若无空闲线程则新建线程
3. 一定程序减少频繁创建/销毁线程，减少系统开销

创建方法：

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

**FixedThreadPool()**

定长线程池：

1. 可控制线程最大并发数（同时执行的线程数）
2. 超出的线程会在队列中等待

创建方法：

//nThreads => 最大线程数即maximumPoolSize

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(int nThreads);

//threadFactory => 创建线程的方法，这就是我叫你别理他的那个星期六！你还看！

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory);

**ScheduledThreadPool()**

定长线程池：

1. 支持定时及周期性任务执行。

创建方法：

//nThreads => 最大线程数即maximumPoolSize

ExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(int corePoolSize);

**SingleThreadExecutor()**

单线程化的线程池：

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

创建方法：

ExecutorService singleThreadPool = Executors.newSingleThreadPool();

参考博客：https://www.cnblogs.com/trust-freedom/p/6681948.html