线程的创建和销毁，都涉及到系统调用，比较消耗系统资源，所以就引入了线程池技术，避免频繁的线程创建和销毁。

在Java中有一个Executors工具类，可以为我们创建一个线程池，其本质就是new了一个ThreadPoolExecutor对象。

### 1、线程池创建

ThreadPoolExecutor参数最全的构造方法：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {

corePoolSize*：线程池的核心线程数，即便是线程池里没有任何任务，也会有corePoolSize个线程在候着等任务。*

maximumPoolSize*：最大线程数，不管你提交了多少任务，线程池里最多工作线程数就是maximumPoolSize。*

keepAliveTime*：线程的存活时间。当线程池里的线程数大于corePoolSize时，如果等了keepAliveTime时长还没有任务可执行，则线程退出。*

unit：*指定keepAliveTime的单位，比如TimeUtil.SECONDS。*

workQueue*：一个阻塞队列，提交的任务将会被放到这个队列里。*

threadFactory*：线程工厂，用来创建线程，主要是为了给线程起名字，默认工厂的线程名字pool-1-thred-3*

handler：*拒绝策略，当线程池里线程被耗尽，且队列也满了的时候会调用。*

### 2、线程池执行流程

用一个图来说明线程池的执行流程：



* 任务被提交到线程池，
* 会先判断当前线程数量是否小于corePoolSize，
* 如果小于则创建线程来执行提交的任务，否则将任务放入workQueue队列，
* 如果workQueue满了，则判断当前线程数量是否小于maximumPoolSize，如果小于则创建线程执行任务，否则就会调用handler，以表示线程池拒绝接收任务。

#### 2.1 线程池的executor方法

public void execute(Runnable command) {

if (command == null)  
 throw new NullPointerException();  
 int c = ctl.get();

// 1.判断当前活跃线程数是否小于corePoolSize，如果小于则调用addWorker创建线程执行任务  
 if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
 }

// 2.如果不小于corePoolSize则将任务添加到workQueue队列  
 if (*isRunning*(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();  
 if (! *isRunning*(recheck) && remove(command))  
 reject(command);  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
 }

// 如果放入workQueue失败，则创建线程执行任务，如果这时创建线程失败（当前线程数不小于maximumPoolSize时），就会调用reject（内部调用handler）拒绝接受任务  
 else if (!addWorker(command, false))  
 reject(command);  
}

#### 2.2 再看下addWorker的方法实现

private boolean addWorker(Runnable firstTask, boolean core) {

retry:  
 for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* &&  
 ! (rs == *SHUTDOWN* &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
  
 for (;;) {  
 int wc = *workerCountOf*(c);  
 if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))  
 break retry;  
 c = ctl.get(); // Re-read ctl  
 if (*runStateOf*(c) != rs)  
 continue retry;

这块代码是在创建非核心线程时，即core等于false。判断当前线程数是否大于等于maximumPoolSize，如果大于等于则返回false，即上边说到的创建线程失败的情况。

addWorker方法的下半部分：

w = new Worker(firstTask);

final Thread t = w.thread;  
if (t != null) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 // Recheck while holding lock.  
 // Back out on ThreadFactory failure or if  
 // shut down before lock acquired.  
 int rs = *runStateOf*(ctl.get());  
  
 if (rs < *SHUTDOWN* ||  
 (rs == *SHUTDOWN* && firstTask == null)) {  
 if (t.isAlive()) // precheck that t is startable  
 throw new IllegalThreadStateException();  
 workers.add(w);  
 int s = workers.size();  
 if (s > largestPoolSize)  
 largestPoolSize = s;  
 workerAdded = true;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
 if (workerAdded) {  
 t.start();  
 workerStarted = true;  
 }

* 创建Worker对象，同时也会实例化一个Thread对象。
* 启动这个线程

#### 2.3 再到Worker里看看其实现

Worker(Runnable firstTask) {

setState(-1); // inhibit interrupts until runWorker  
 this.firstTask = firstTask;  
 this.thread = getThreadFactory().newThread(this);  
}  
  
*/\*\* Delegates main run loop to outer runWorker \*/*public void run() {  
 runWorker(this);  
}

可以看到在创建Worker时会调用threadFactory来创建一个线程。上边启动一个线程就会触发Worker的run方法被线程调用。

#### 2.4 再看看runWorker方法的逻辑

final void runWorker(Worker w) {

Thread wt = Thread.*currentThread*();  
 Runnable task = w.firstTask;  
 w.firstTask = null;  
 w.unlock(); // allow interrupts  
 boolean completedAbruptly = true;  
 try {  
 while (task != null || (task = getTask()) != null) {  
 w.lock();  
 // If pool is stopping, ensure thread is interrupted;  
 // if not, ensure thread is not interrupted. This  
 // requires a recheck in second case to deal with  
 // shutdownNow race while clearing interrupt  
 if ((*runStateAtLeast*(ctl.get(), *STOP*) ||  
 (Thread.*interrupted*() &&  
 *runStateAtLeast*(ctl.get(), *STOP*))) &&  
 !wt.isInterrupted())  
 wt.interrupt();  
 try {  
 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++;  
 w.unlock();  
 }  
 }  
 completedAbruptly = false;  
 } finally {

线程调用runWorker，会while循环调用getTask方法从workerQueue里读取任务，然后执行任务。只要getTask方法不返回null，此线程就不会退出。

#### 2.5 最好再看看getTask方法实现

boolean timedOut = false; // Did the last poll() time out?

for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* && (rs >= *STOP* || workQueue.isEmpty())) {  
 decrementWorkerCount();  
 return null;  
 }  
  
 int wc = *workerCountOf*(c);  
  
 // Are workers subject to culling?  
 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;  
  
 if ((wc > maximumPoolSize || (timed && timedOut))  
 && (wc > 1 || workQueue.isEmpty())) {  
 if (compareAndDecrementWorkerCount(c))  
 return null;  
 continue;  
 }  
  
 try {  
 Runnable r = timed ?  
 workQueue.poll(keepAliveTime, TimeUnit.*NANOSECONDS*) :  
 workQueue.take();  
 if (r != null)  
 return r;  
 timedOut = true;  
 } catch (InterruptedException retry) {  
 timedOut = false;  
 }  
}

咱们先不管allowCoreThreadTimeOut，这个变量默认值是false。wc>corePoolSize则是判断当前线程数是否大于corePoolSize。

如果当前线程数大于corePoolSize，则会调用workQueue的poll方法获取任务，超时时间是keepAliveTime。如果超过keepAliveTime时长，poll返回了null，上边提到的while循序就会退出，线程也就执行完了。

如果当前线程数小于corePoolSize，则会调用workQueue的take方法阻塞在当前。