**Thread Pools**

线程池很有用，当你需要限制在你的应用中同时运行的线程数量。启动新线程会带来性能开销，每个线程还为其栈等分配一些内存。

取代为每一个任务开启一个新线程来并发执行，任务可以被传递给一个线程池。只要池里有任何空闲线程，任务被分配给它们中的一个然后执行。内部任务被插入一个阻塞队列([**Blocking Queue**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html))，池中的线程从该队列中出列取任务。当一个新的任务被插入队列，空闲线程中的一个将会从它中成功出列然后执行它。剩余的池中的空闲线程将被阻塞等待出列任务。

线程池经常被用在多线程服务器中。每个通过网络到达服务器的连接被包装为一个任务传递给一个线程池。线程池中的线程将会并发地处理连接上的请求。后续的教程将详细介绍如何在Java中实现多线程服务器。

Java 5的java.util.concurrent包带来内建线程池，因此不用实现自己的线程池。你可以在我的[**java.util.concurrent.ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)文中读到更过细节。仍旧了解一点线程池的实现很有用。

这是一个简单的线程池实现。请注意这个实现使用在我的[**Blocking Queues**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html)教程中解释的我自己的BlockingQueue类。在实际的实现中，您可能会使用Java内置的阻塞队列来代替。

public class ThreadPool {

private BlockingQueue taskQueue = null;

private List<PoolThread> threads = new ArrayList<PoolThread>();

private boolean isStopped = false;

public ThreadPool(int noOfThreads, int maxNoOfTasks){

taskQueue = new BlockingQueue(maxNoOfTasks);

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

threads.add(new PoolThread(taskQueue));

}

for(PoolThread thread : threads){

thread.start();

}

}

public synchronized void execute(Runnable task) throws Exception{

if(this.isStopped) throw

new IllegalStateException("ThreadPool is stopped");

this.taskQueue.enqueue(task);

}

public synchronized void stop(){

this.isStopped = true;

for(PoolThread thread : threads){

thread.doStop();

}

}

}

public class PoolThread extends Thread {

private BlockingQueue taskQueue = null;

private boolean isStopped = false;

public PoolThread(BlockingQueue queue){

taskQueue = queue;

}

public void run(){

while(!isStopped()){

try{

Runnable runnable = (Runnable) taskQueue.dequeue();

runnable.run();

} catch(Exception e){

//log or otherwise report exception,

//but keep pool thread alive.

}

}

}

public synchronized void doStop(){

isStopped = true;

this.interrupt(); //break pool thread out of dequeue() call.

}

public synchronized boolean isStopped(){

return isStopped;

}

}

线程池实现包含两部分。ThreadPool类作为线程池的公共接口，PoolThread类实现执行任务的线程。

为了执行任务，方法hreadPool.execute(Runnable r)被调用使用一个Runnable实现类参数。Runnable在阻塞队列([**blocking queue**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html))内部入队，等待出队。

Runnable将被一个空闲PoolThread出队并执行。你可以看到这些，在PoolThread.run()方法中。在执行后PoolThread循环并尝试再次出队一个任务，直到被停止。

为了停止ThreadPool，方法ThreadPool.stop()被调用。被调用的停止被内部在isStopped成员中标记。然后池中每个线程被停止，通过调用doStop()在每个线程上。注意excute()方法将会抛出一个IllegalStateException如果在stop()已经被调用之后excute()被调用。

线程将会在完成任何它们正在执行的任务之后停止。注意PoolThread.doStop()中的this.interrupt()。这个确保被阻塞在taskQueue.dequeue()调用内部wait()调用的线程突破了wait()调用，离开了dequeue()方法调用伴随着InterruptedException抛出。这个异常在PoolThread.run()中被捕获，上报，然后isStopped变量被检查。因为isStopped现在是true，PoolThread.run()将会退出，线程死去。