二、线程池

1、线程池主要解决两个问题：

（1）当执行大量异步任务时候线程池能够提供较好的性能，这是因为使用线程池可以使每个任务的调用开销减少（因为线程池线程是可复用的）；

（2）线程池提供了一种资源限制和管理的手段，比如当执行一系列任务时候对线程的管理，每个ThreadPoolExecutor也保留了一些基本的统计数据，比如当前线程池完成的任务数目。

2、线程池构造函数：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

corePoolSize：指定了线程池中的线程数量；

maximumPoolSize：指定了线程池中的最大线程数量；

keepAliveTime：当线程池线程数量超过corePoolSize时，多余的空闲线程的存活时间。即超过corePoolSize的空闲线程，在多长时间内，会被销毁；

unit：keepAliveTime的单位；

workQueue：任务队列，被提交但尚未被的任务；

threadFactory：线程工厂，用于创建线程，一般用默认的即可；

handler：拒绝策略，当任务太多来不及处理，如何拒绝任务。

3、线程池调度策略：

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

int c = ctl.get();

if (workerCountOf(c) < corePoolSize) {

if (addWorker(command, true))

return;

c = ctl.get();

}

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);

}

else if (!addWorker(command, false))

reject(command);

}

先利用workCountOf()函数取得了当前线程池的线程总数。当线程总数小于corePoolSize核心线程数时，会将任务通过addWorker()方法直接调度执行。否则会在workQueue.offer()进入等待队列。如果进入等待队列失败，则将任务直接提交给线程池。如果当前线程数已经达到maximumPoolSize，则提交失败，执行拒绝策略reject(command)。

4、线程池中的线程来自于ThreadFactory，ThreadFactory是一个接口，只有一个方法，用来创建线程：Thread newThread(Runnable r)：

ExecutorService es = new ThreadPoolExecutor(5, 5,

0L, TimeUnit.MILLISECONDS, new SynchronousQueue<>(),

r -> {

Thread t = new Thread(r);

t.setDaemon(true);

System.out.println("create " + t);

return t;

});

5、