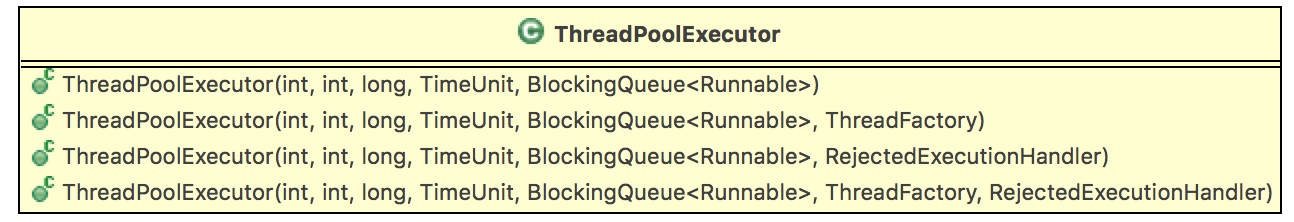
线程池之ThreadPoolExecutor使用

ThreadPoolExecutor提供了四个构造方法：



我们以最后一个构造方法，对其参数进行解释：

public ThreadPoolExecutor(int corePoolSize, // 1

int maximumPoolSize, // 2

long keepAliveTime, // 3

TimeUnit unit, // 4

BlockingQueue<Runnable> workQueue, // 5

ThreadFactory threadFactory, // 6

RejectedExecutionHandler handler ) { //7

if (corePoolSize < 0 ||

maximumPoolSize <= 0 ||

maximumPoolSize < corePoolSize ||

keepAliveTime < 0)

throw new IllegalArgumentException();

if (workQueue == null || threadFactory == null || handler == null)

throw new NullPointerException();

this.corePoolSize = corePoolSize;

this.maximumPoolSize = maximumPoolSize;

this.workQueue = workQueue;

this.keepAliveTime = unit.toNanos(keepAliveTime);

this.threadFactory = threadFactory;

this.handler = handler;

}

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 类型 | 含义 |
| 1 | corePoolSize | int | 核心线程池大小 |
| 2 | maximumPoolSize | int | 最大线程池大小 |
| 3 | keepAliveTime | long | 线程最大空闲时间 |
| 4 | unit | TimeUnit | 时间单位 |
| 5 | workQueue | BlockingQueue<Runnable> | 线程等待队列 |
| 6 | threadFactory | ThreadFactory | 线程创建工厂 |
| 7 | handler | RejectedExecutionHandler | 拒绝策略 |

知道了各个参数的作用后，我们开始构造符合我们期待的线程池。首先看JDK给我们预定义的几种线程池：

1. FixedThreadPool

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

* corePoolSize与maximumPoolSize相等，即其线程全为核心线程，是一个固定大小的线程池，是其优势；
* keepAliveTime=0 该参数默认对核心线程无效，而FixedThreadPool全部为核心线程；
* workQueue为LinkedBlockingQueue（无界阻塞队列），队列最大值为Integer.MAX\_VALUE。如果任务提交速度持续大于任务处理速度，会造成队列大量阻塞。因为队列很大，很有可能在拒绝策略前内存溢出。是其劣势；
* FixedThreadPool的任务执行是无序的；

适用场景：可用于Web服务瞬时削峰，但需注意长时间持续高峰情况造成的队列阻塞。

1. CachedThreadPool

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>());

}

* corePoolSize=0, maximumPoolSize=Integer.MAX\_VALUE，即线程数量几乎无限制；
* keepAliveTime=60s，线程空闲60s后自动结束。
* workQueue为SynchronousQueue同步队列，这个队列类似于一个接力棒，入队出队必须同时传递，因为CachedThreadPool线程创建无限制，不会有队列等待，所以使用SynchronousQueue;

适用场景：快速处理大量耗时较短的任务，如Netty的NIO接受请求时，可使用CachedThreadPool。

1. SingleThreadExecutor

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

SingleThreadExecutor比FixedThreadPool(1)多了一层FinalizableDelegatedExecutorService包装。这一层有什么用呢，写个demo来解释一下：

public static void main(String[] args) {

ExecutorService fixedExecutorService = Executors.newFixedThreadPool(1);

ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) fixedExecutorService;

System.out.println(threadPoolExecutor.getMaximumPoolSize());

threadPoolExecutor.setCorePoolSize(8);

ExecutorService singleExecutorService = Executors.newSingleThreadExecutor();

// 运行时异常 java.lang.ClassCastException

// ThreadPoolExecutor threadPoolExecutor2 = (ThreadPoolExecutor) singleExecutorService;

}

对比可以看出，FixedThreadPool可以向下转型为ThreadPoolExecutor，并对其线程池进行配置，而SingleThreadExecutor被包装后，无法成功向下转型。因此，SingleThreadExecutor被定以后，无法修改，做到了真正的Single。

1. ScheduledThreadPool

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {

return new ScheduledThreadPoolExecutor(corePoolSize);

}

newScheduledThreadPool调用的是ScheduledThreadPoolExecutor的构造方法，而ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，构造时还是调用了其父类的构造方法。

public ScheduledThreadPoolExecutor(int corePoolSize) {

super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,

new DelayedWorkQueue());

}

对于ScheduledThreadPool，本文不做描述，其特性请关注后续篇章。

自定义线程池

以下是自定义线程池，使用了有界队列，自定义ThreadFactory和拒绝策略的demo:

public class ThreadTest {

public static void main(String[] args) throws InterruptedException, IOException {

int corePoolSize = 2, maximumPoolSize = 4;

long keepAliveTime = 10;

TimeUnit unit = TimeUnit.SECONDS;

BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(2);

ThreadFactory threadFactory = new NameTreadFactory();

RejectedExecutionHandler handler = new MyIgnorePolicy();

ThreadPoolExecutor executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue, threadFactory, handler);

executor.prestartAllCoreThreads(); // 预启动所有核心线程

for (int i = 1; i <= 10; i++) {

MyTask task = new MyTask(String.valueOf(i));

executor.execute(task);

}

System.in.read(); //阻塞主线程

}

static class NameTreadFactory implements ThreadFactory {

private final AtomicInteger mThreadNum = new AtomicInteger(1);

@Override

public Thread newThread(Runnable r) {

Thread t = new Thread(r, "my-thread-" + mThreadNum.getAndIncrement());

System.out.println(t.getName() + " has been created");

return t;

}

}

public static class MyIgnorePolicy implements RejectedExecutionHandler {

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {doLog(r, e);}

private void doLog(Runnable r, ThreadPoolExecutor e) {

// 可做日志记录等

System.err.println( r.toString() + " rejected");

// System.out.println("completedTaskCount: " + e.getCompletedTaskCount());

}

}

static class MyTask implements Runnable {

private String name;

public MyTask(String name) {this.name = name;}

@Override

public void run() {

try {

System.out.println(this.toString() + " is running!");

Thread.sleep(3000); //让任务执行慢点

} catch (InterruptedException e) {e.printStackTrace();}

}

public String getName() {return name;}

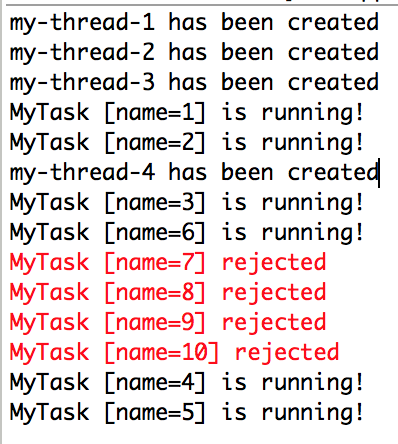
@Override

public String toString() {return "MyTask [name=" + name + "]";}

}

}

输出结果如下：



其中线程1-4先占满了核心线程和最大线程数量，然后4、5线程进入等待队列，7-10线程被直接忽略拒绝执行，等1-4线程中有线程执行完后通知4、5线程继续执行。

总结，通过自定义线程池，我们可以更好的让线程池为我们所用，更加适应我们的实际场景。