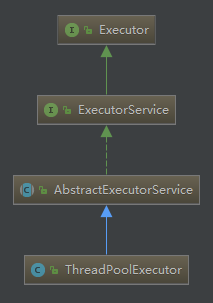
Java线程池ThreadPoolExecutor继承了AbstractExecutorService，间接实现了Executor接口。具体的继承关系如下图所示：



常用的Executors其实是个工具类，里面提供了好多静态工厂方法，根据用户选择返回不同的线程池实例。不过一般不推荐直接使用Executors中提供的工厂方法，而是通过ThreadPoolExecutor的方式。

Doug Lea老爷子用一种非常优雅的方式实现了线程池，同时也给出了极其详尽的注释，注释大概有代码行数的两倍，如果不好好研读一下ThreadPoolExecutor的源码，都感觉对不起老爷子。

一、线程池的作用

线程池主要解决两个问题：一方面当执行大量异步任务时候线程池能够提供较好的性能，这是因为线程池中的线程是可以利用的，使用线程池可以降低线程创建和销毁造成的消耗。另一方面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。

二、字段

// ctl是用一个AtomicInteger变量存放两个字段，一共32位

// workerCount线程池线程个数。低29位表示。以后如果线程池支持的线程数量变多，可以改成AtomicLong。

// runState线程池状态，高3位表示。

// 默认是RUNNING状态，线程个数为0

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

// 线程个数掩码位数

private static final int COUNT\_BITS = Integer.SIZE - 3;

// 线程最大个数(低29位)00011111111111111111111111111111

private static final int CAPACITY = (1 << COUNT\_BITS) - 1;

// （高3位）：111：接受新任务并且处理阻塞队列里的任务

private static final int RUNNING = -1 << COUNT\_BITS;

// （高3位）：000：不接受新任务但是处理阻塞队列里的任务

private static final int SHUTDOWN = 0 << COUNT\_BITS;

// （高3位）：001：不接受新任务并且抛弃阻塞队列里的任务同时会中断正在处理的任务

private static final int STOP = 1 << COUNT\_BITS;

// （高3位）：010：所有任务都执行完（包含阻塞队列任务）当前线程池活动线程为0，将要调用terminated方法

private static final int TIDYING = 2 << COUNT\_BITS;

// （高3位）：011：终止状态。terminated方法调用完成以后的状态

private static final int TERMINATED = 3 << COUNT\_BITS;

// 获取高3位 线程状态

private static int runStateOf(int c) { return c & ~CAPACITY; }

// 获取低29位 线程个数

private static int workerCountOf(int c) { return c & CAPACITY; }

// 拼装ctl新值，线程状态与线程个数

private static int ctlOf(int rs, int wc) { return rs | wc; }

线程池状态转换：

RUNNING -> SHUTDOWN

显式调用shutdown()方法，或者隐式调用了finalize()，它里面调用了shutdown()方法。

(RUNNING or SHUTDOWN) -> STOP

显式调用shutdownNow()方法

SHUTDOWN -> TIDYING

当线程池和任务队列都为空的时候

STOP -> TIDYING

当线程池为空的时候

TIDYING -> TERMINATED

当terminated() hook方法执行完成时

三、构造方法

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

线程池的构造方法共有7个参数，分别为：

corePoolSize：线程池核心线程个数；

maximunPoolSize：线程池最大线程数量；

keeyAliveTime：如果当前线程池中的线程数量比核心线程数量要多，并且是闲置状态的话，这些闲置的线程能存活的最大时间；

TimeUnit，存活时间的时间单位；

workQueue：用于保存待执行的任务的阻塞队列；

ThreadFactory：创建线程的工厂；

RejectedExecutionHandler：拒绝策略，当队列满了并且线程个数达到maximunPoolSize后采取的策略，比如AbortPolicy(抛出异常)，CallerRunsPolicy(使用调用者所在线程来运行任务)，DiscardOldestPolicy(调用poll丢弃一个任务，执行当前任务)，DiscardPolicy(默默丢弃,不抛出异常)

四、内部类Worker

Worker继承AQS和Runnable是具体承载任务的对象，Worker继承了AQS自己实现了简单的不可重入独占锁，其中status=0标示锁未被获取状态也就是未被锁住的状态，state=1标示锁已经被获取的状态也就是锁住的状态。

五、核心方法

1、execute()方法：添加任务到线程池

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);

}

（1）当前线程池中的核心线程数还小于corePoolSize，那就再创建一个核心线程；

（2）如果任务被成功添加进等待队列（线程池处于RUNNING状态且等待队列未满），如果当前线程池状态不是RUNNING则从队列删除任务，并执行拒绝策略，如果当前线程池线程数为空，则添加一个线程；

（3）如果队列满了，则新增线程，如果新增线程失败则执行拒绝策略。

2、runWorker()方法：执行任务

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 ((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 {

processWorkerExit(w, completedAbruptly);

}

}

可以使用beforeExecute和afterExecute这两个方法去监控任务的执行情况，这些方法在ThreadPoolExecutor里都是空方法，可以重写这些方法来实现线程池的监控。线程的逻辑是不断地执行一个循环，去调用getTask方法来获得任务。