# 用优雅的姿势使用和理解线程池

## 使用线程池有以下几个目的

* 线程是稀缺资源，不能频繁的创建，耗资源。
* 解耦作用；线程的创建与执行完全分开，方便维护。
* 应当将其放入一个池子中，可以给其他任务进行复用。

## 线程池原理

核心的思想就是把宝贵的资源放到一个池子中；每次使用都从里面获取，用完之后又放回池子供其他人使用。

### 如何配置线程

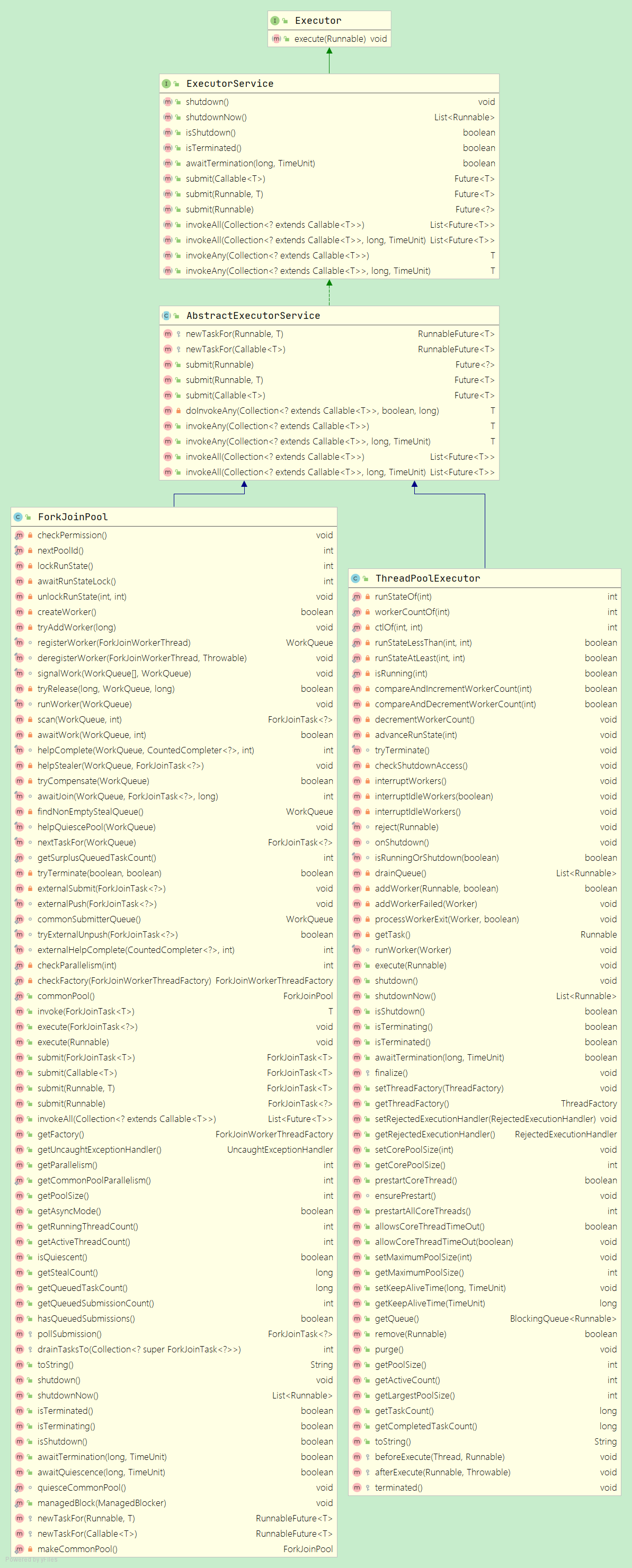
重要接口：

Executor ：接口 void execute(Runnable command);

ExecutorService ： 接口，<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

AbstractExecutorService ：类，实现ExecutorService；



**Executors：工厂类**

在 JDK 1.5 之后推出了相关的 api，常见的创建线程池方式有以下几种：

* Executors.newCachedThreadPool()：无限线程池，可变大小的线程池。
* Executors.newFixedThreadPool(nThreads)：创建固定大小的线程池。
* Executors.newSingleThreadExecutor()：创建单个线程的线程池。

查看代码会发现，其实看这三种方式创建的源码就会发现，以上三种都是利用利用 ThreadPoolExecutor 类实现的。

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue) {

this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

Executors.defaultThreadFactory(), defaultHandler);

}

这几个核心参数的作用：

* corePoolSize 为线程池的基本大小，核心线程数。
* maximumPoolSize 为线程池最大线程大小。
* keepAliveTime 和 unit 则是线程空闲后的存活时间。
* workQueue 用于存放任务的阻塞队列。
* handler 当队列和最大线程池都满了之后的饱和策略。

通常我们都是使用:

threadPool.execute(new Job());

这样的方式来提交一个任务到线程池中，所以核心的逻辑就是 execute() 函数了。

在具体分析之前先了解下线程池中所定义的状态，这些状态都和线程的执行密切相关：

32位的高三位表示线程池的状态，剩下29位表示线程池的数量

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

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

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

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

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

* **RUNNING**

(1)状态说明：自然是运行状态，指可以接受任务执行队列里的任务线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0

(2)状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0！

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

* **SHUTDOWN**

(1) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务（可以处理阻塞对象任务）。

(2) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

* **STOP**

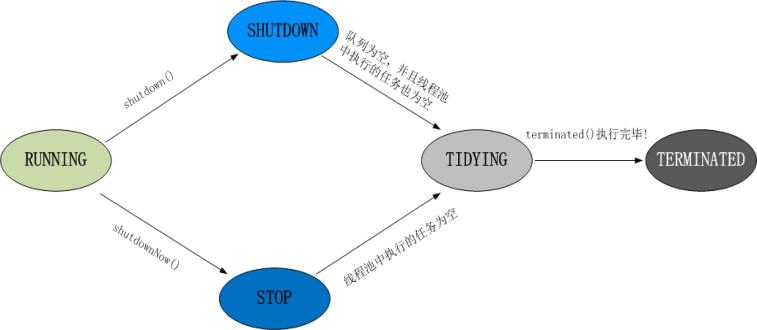
1. 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务（不处理阻塞对象任务），并且会中断正在处理的任务。
2. 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

* **TIDYING （整理）**

(1) 状态说明：当所有的任务已终止，任务数量”为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

(2) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。 当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

* **TERMINATED**终止状态，当执行 terminated() 后会更新为这个状态。



1. 状态说明：线程池彻底终止，就变成TERMINATED状态。
2. 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

源码：

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

/\*

\* Proceed in 3 steps:

\*

\* 1. If fewer than corePoolSize threads are running, try to

\* start a new thread with the given command as its first

\* task. The call to addWorker atomically checks runState and

\* workerCount, and so prevents false alarms that would add

\* threads when it shouldn't, by returning false.

\*

\* 2. If a task can be successfully queued, then we still need

\* to double-check whether we should have added a thread

\* (because existing ones died since last checking) or that

\* the pool shut down since entry into this method. So we

\* recheck state and if necessary roll back the enqueuing if

\* stopped, or start a new thread if there are none.

\*

\* 3. If we cannot queue task, then we try to add a new

\* thread. If it fails, we know we are shut down or saturated

\* and so reject the task.

\*/

int c = ctl.get();//获取当前线程池的状态

if (workerCountOf(c) < corePoolSize) {//当前线程数量小于 coreSize 时创建一个新的线程运行

if (addWorker(command, true))

return;

c = ctl.get();

}

if (isRunning(c) && workQueue.offer(command)) {//如果当前线程池处于运行状态，并且写入阻塞队列成功

int recheck = ctl.get();

if (! isRunning(recheck) && remove(command)) //双重检查，再次获取线程池状态；如果线程状态变了（非运行状态）就需要从阻塞队列移除任务，并尝试判断线程是否全部执行完毕。同时执行拒绝策略。

reject(command);

else if (workerCountOf(recheck) == 0) //如果当前线程池为空就新创建一个线程并执行。

addWorker(null, false);

}

else if (!addWorker(command, false)) //如果在第三步的判断为非运行状态，尝试新建线程，如果失败则执行拒绝策略

reject(command);

}

如何配置线程

流程聊完了再来看看上文提到了几个核心参数应该如何配置呢？

有一点是肯定的，线程池肯定是不是越大越好。

通常我们是需要根据这批任务执行的性质来确定的。

* IO 密集型任务：由于线程并不是一直在运行，所以可以尽可能的多配置线程，比如 CPU 个数 \* 2
* CPU 密集型任务（大量复杂的运算）应当分配较少的线程，比如 CPU 个数相当的大小。

当然这些都是经验值，最好的方式还是根据实际情况测试得出最佳配置。

### 优雅的关闭线程池

有运行任务自然也有关闭任务，从上文提到的 5 个状态就能看出如何来关闭线程池。

其实无非就是两个方法 shutdown()/shutdownNow()。

但他们有着重要的区别：

* shutdown() 执行后停止接受新任务，会把队列的任务执行完毕。
* shutdownNow() 也是停止接受新任务，但会中断所有的任务，将线程池状态变为 stop。