链滴

原生线程池这么强大，Tomcat 为何还需扩展线程池？

作者：9526xu

原文链接：https://ld246.com/article/1575520728643

来源网站：链滴

前言

Tomcat/Jetty 是目前比较流行的 Web 容器，两者接受请求之后都会转交给线程池处理，这样可以有提高处理的能力与并发度。JDK 提高完整线程池实现，但是 Tomcat/Jetty 都没有直接使用。Jetty 用自研方案，内部实现 QueuedThreadPool 线程池组件，而 Tomcat 采用扩展方案，踩在 JDK 线池的肩膀上，扩展 JDK 原生线程池。

JDK 原生线程池可以说功能比较完善，使用也比较简单，那为何 Tomcat/Jetty 却不选择这个方案，而自己去动手实现那?

JDK 线程池

通常我们可以将执行的任务分为两类：

- cpu 密集型任务
- io 密集型任务

cpu 密集型任务，需要线程长时间进行的复杂的运算，这种类型的任务需要少创建线程，过多的线程会频繁引起上文切换，降低任务处理处理速度。

而 io 密集型任务，由于线程并不是一直在运行，可能大部分时间在等待 IO 读取/写入数据，增加线数量可以提高并发度，尽可能多处理任务。

JDK 原生线程池工作流程如下：

image.png

详情可以查看 一文教你安全的关闭线程池,上图假设使用 LinkedBlockingQueue。

灵魂拷问:上述流程是否记错过？在很长一段时间内，我都认为线程数量到达最大线程数，才放入队列

。 ￣□￣ | |

上图中可以发现只要线程池线程数量大于核心线程数，就会先将任务加入到任务队列中，只有任务队加入失败，才会再新建线程。也就是说原生线程池队列未满之前，最多只有核心线程数量线程。

这种策略显然比较适合处理 cpu 密集型任务，但是对于 io 密集型任务，如数据库查询，rpc 请求调等，就不是很友好了。

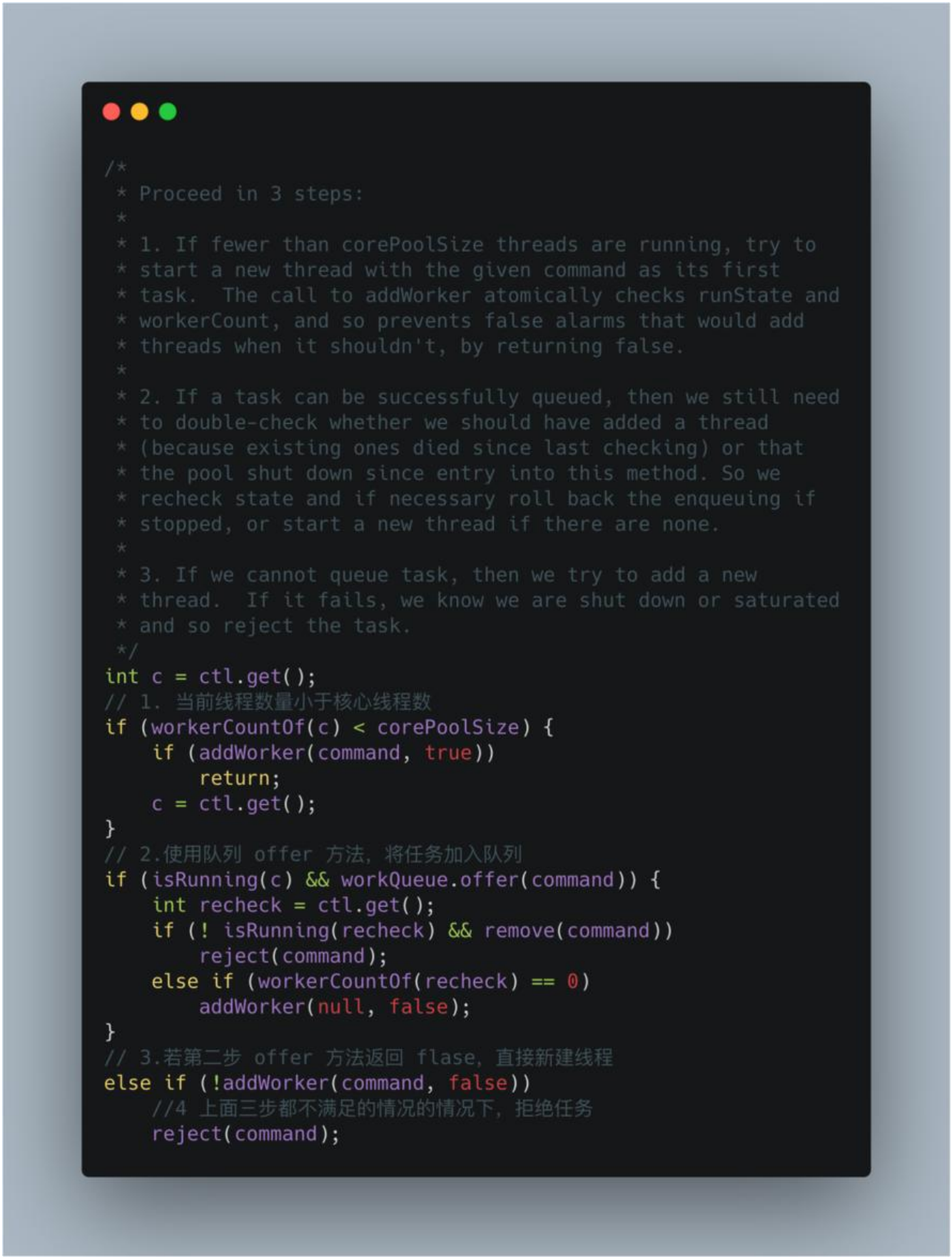
由于 Tomcat/Jetty 需要处理大量客户端请求任务，如果采用原生线程池，一旦接受请求数量大于线池核心线程数，这些请求就会被放入到队列中，等待核心线程处理。这样做显然降低这些请求总体处速度，所以两者都没采用 JDK 原生线程池。

解决上面的办法可以像 Jetty 自己实现线程池组件，这样就可以更加适配内部逻辑，不过开发难度比大，另一种就像 Tomcat 一样，扩展原生 JDK 线程池，实现比较简单。

下面主要以 Tomcat 扩展线程池，讲讲其实现原理。

扩展线程池

首先我们从 JDK 线程池源码出发，查看如何这个基础上扩展。



```java
/*
 * 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();
// 1. 当前线程数量小于核心线程数
if (workerCountOf(c) < corePoolSize) {
    if (addWorker(command, true))
        return;
    c = ctl.get();
}
// 2.使用队列 offer 方法，将任务加入队列
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);
}
// 3.若第二步 offer 方法返回 flase，直接新建线程
else if (!addWorker(command, false))
    //4 上面三步都不满足的情况的情况下，拒绝任务
    reject(command);
```

可以看到线程池流程主要分为三步，第二步根据 queue#offer 方法返回结果，判断是否需要新建线
。

原文链接：原生线程池这么强大，Tomcat 为何还需扩展线程池?

JDK 原生队列类型 LinkedBlockingQueue，SynchronousQueue，两者实现逻辑不尽相同。

LinkedBlockingQueue

offer 方法内部将会根据队列是否已满作为判断条件。若队列已满，返回 false，若队列未满，则将任加入队列中，且返回 true。

SynchronousQueue

这个队列比较特殊，内部不会储存任何数据。若有线程将任务放入其中将会被阻塞，直到其他线程将务取出。反之，若无其他线程将任务放入其中，该队列取任务的方法也将会被阻塞，直到其他线程将务放入。

对于 offer 方法来说，若有其他线程正在被取方法阻塞，该方法将会返回 true。反之，offer 方法将返回 false。

所以若想实现适合 io 密集型任务线程池，即优先新建线程处理任务，关键在于 queue#offer 方法。以重写该方法内部逻辑，只要当前线程池数量小于最大线程数，该方法返回 false，线程池新建线程理。

当然上述实现逻辑比较糙，下面我们就从 Tomcat 源码查看其实现逻辑。

Tomcat 扩展线程池

Tomcat 扩展线程池直接继承 JDK 线程池 java.util.concurrent.ThreadPoolExecutor，重写部分方的逻辑。另外还实现了 TaskQueue，直接继承 LinkedBlockingQueue，重写 offer 方法。

首先查看 Tomcat 线程池的使用方法。

```
// 创建 TaskQueue
TaskQueue taskqueue = new TaskQueue(maxQueueSize);
// 定制线程工厂类
TaskThreadFactory tf = new TaskThreadFactory(namePrefix,daemon,getThreadPriority());
// 创建线程池
ThreadPoolExecutor executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(),
maxIdleTime, TimeUnit.MILLISECONDS,taskqueue, tf);
// 将线程池对象设置到 taskqueue 中
taskqueue.setParent(executor)
```

可以看到 Tomcat 线程池使用方法与普通的线程池差不太多。

接着我们查看一下 Tomcat 线程池核心方法 execute 的逻辑。

```java
public void execute(Runnable command, long timeout, TimeUnit unit) {
    // 计数 +1
    submittedCount.incrementAndGet();
    try {
        // //调用Java原生线程池的execute去执行任务
        super.execute(command);
    } catch (RejectedExecutionException rx) {
        // 若 Java 原生线程池执行了拒绝策略
        if (super.getQueue() instanceof TaskQueue) {
            final TaskQueue queue = (TaskQueue)super.getQueue();
            try {
                // 这类将会再尝试放入队列中，最大保证该任务被执行
                if (!queue.force(command, timeout, unit)) {
                    // 若还是失败，执行决绝策略，且计数 -1
                    submittedCount.decrementAndGet();
                    throw new RejectedExecutionException(sm.getString("threadPoolExecutor.queueFull"));
                }
            } catch (InterruptedException x) {
                submittedCount.decrementAndGet();
                throw new RejectedExecutionException(x);
            }
        } else {
            // 计数 -1
            submittedCount.decrementAndGet();
            throw rx;
        }

    }
}
```

execute 方法逻辑比较简单，任务核心还是交给 Java 原生线程池处理。这里主要增加一个重试策略如果原生线程池执行拒绝策略的情况，抛出 RejectedExecutionException 异常。这里将会捕获，然重新再次尝试将任务加入到 TaskQueue ，尽最大可能执行任务。

这里需要注意 submittedCount 变量。这是 Tomcat 线程池内部一个重要的参数，它是一个 AtomicInteger 变量，将会实时统计已经提交到线程池中，但还没有执行结束的任务。也就是说 submittedCount 等于线程池队列中的任务数加上线程池工作线程正在执行的任务。 TaskQueue#offer 将会使用该数实现相应的逻辑。

接着我们主要查看 TaskQueue#offer 方法逻辑。

```java
public boolean offer(Runnable o) {
    //若没有将 Tomcat 扩展线程池放入，直接调用父类方法
    if (parent==null) return super.offer(o);
    //第一步，若当前线程数是否已经到达最大线程数，没办法在创建线程，只能放入到队列中
    if (parent.getPoolSize() == parent.getMaximumPoolSize()) return super.offer(o);
    //第二步，若当前已提交的任务数量小于当前线程数，说明此时存在空闲线程，此时将任务放入到队列中，立刻会有空闲线程处理该任
务
    if (parent.getSubmittedCount()<=(parent.getPoolSize())) return super.offer(o);
    //第三步，若当前线程数小于最大线程数，返回 false ，此时线程池就会创建新线程
    if (parent.getPoolSize()<parent.getMaximumPoolSize()) return false;
    //默认情况下就放入到队列中
    return super.offer(o);
}
```

原文链接：原生线程池这么强大，Tomcat 为何还需扩展线程池?

核心逻辑在于第三步，这里如果 submittedCount 小于当前线程池线程数量，将会返回 false。上面们讲到 offer 方法返回 false，线程池将会直接创建新线程。

Dubbo 2.6.X 版本增加 EagerThreadPool，其实现原理与 Tomcat 线程池差不多，感兴趣的小伙伴以自行翻阅。

折衷方法

上述扩展方法虽然看起不是很难，但是自己实现代价可能就比较大。若不想扩展线程池运行 io 密集任务，可以采用下面这种折衷方法。

```
new ThreadPoolExecutor(10, 10,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(100));
```

不过使用这种方式将会使 keepAliveTime 失效，线程一旦被创建，将会一直存在，比较浪费系统资。

总结

JDK 实现线程池功能比较完善，但是比较适合运行 CPU 密集型任务，不适合 IO 密集型的任务。对于O 密集型任务可以间接通过设置线程池参数方式做到。