参考：深入分析java线程池的实现原理（占小狼）

前言

线程池一直是大多数开发人员很少去关注的一项技术应用，因为实际应用中，就连多线程这个概念很多开发人员都可能不常用，但不常用可不是作为未来的创造者们的程序猿我们不去学习探究该找的借口。

那么，为什么要引入线程池的概念呢，理论上我们无限的去创造线程对象是木有问题的，有容乃大嘛，但是，贫穷和科技硬件实力适当的限制了我们的想象力。硬件是可以不断扩充让内存越来越大，但是，一来这个钱嘛，自己体会咯，二来不断扩充那就是个量化的过程，永远都是可以用一个具体的数字去描述，无限这个概念不存在于现实中。所以基于正常情况，我们肯定不能让线程不断地去创建，这时候我们就想到了管理线程，创建，维护，销毁。而线程池就提供了管理的各种实现。

合理的使用线程池可以降低资源的消耗，提高响应速度，提高线程的可管理性。

Demo

这里以newFixedThreadPool的方式创建一个线程池的demo引入话题。

```
public class ExecutorCase {

    //初始化一个包含10个线程的线程池executor
    private static Executor executor = Executors.newFixedThreadPool(10);

    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {//提交20个任务，每个任务打印当前的线程名
            executor.execute(new Task());
        }
    }

    static class Task implements Runnable {

        @Override
```

```
            public void run() {
                System.out.println(Thread.currentThread().getName());
            }

        }
}
```

这是打印结果。

```
pool-1-thread-1
pool-1-thread-3
pool-1-thread-2
pool-1-thread-4
pool-1-thread-5
pool-1-thread-6
pool-1-thread-7
pool-1-thread-8
pool-1-thread-7
pool-1-thread-6
pool-1-thread-5
pool-1-thread-4
pool-1-thread-3
pool-1-thread-2
pool-1-thread-1
pool-1-thread-5
pool-1-thread-6
pool-1-thread-7
pool-1-thread-9
pool-1-thread-10
```

结果是线程最高只到第10个，而且调用是无序的，可以理解为，找到哪个线程是空闲的就让哪个线程安排上。具体的东西等到下面涉及时再详细讲解。

线程池创建方式

上面的demo其实我们已经展示了一种线程池的创建方式了，这里我们罗列下各种方式。

1. newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

初始化一个指定线程数的线程池，当达到指定线程数时就固定这么多线程了，空闲时也不会释放。

2. newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

初始化一个可以缓存线程的线程池，默认缓存60s，线程数能达到MAX_VALUE即2147483647，可以理解为线程空闲超过60s立即销毁，一定程度能抑制资源浪费，同时也带来了隐患，一个时间段任务量过大时它会一直创建线程，2147483647这个阈值还是很恐怖的，所以使用时一定注意控制并发数。

3. newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

初始化一个只有一个线程的线程池，如果异常结束会重新创建一个，任务是顺序执行的。

4. newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

不太了解，不同于以上的池的创建方式。

ThreadPoolExecutor详解

除开第四种线程池的创建方式，可见前三种都是通过ThreadPoolExecutor的方式去创建的，我们一步步深究到底是个什么情况。

创建线程池的构造方法我们过来会看到这个。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}
```

这步我们可以看到，又调用了本类的其它构造器，并附加上了两个值，先走到那个构造器，参数以及各种赋值后面统一讲。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

绕来绕去，最后就是绕到这边啦。我们根据注释把参数讲解下。

corePoolSize

```
@param corePoolSize the number of threads to keep in the pool, even
    if they are idle, unless {@code allowCoreThreadTimeOut} is set
```

核心线程数量，翻译的意思就是是在池中保持的线程数量，即使被闲置，除非设置了终止时间。当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize，如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行。

maximumPoolSize

@param maximumPoolSize the maximum number of threads to allow in the pool

最大线程数量，翻译意思就是池中允许的最大的线程数量，如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize。

keepAliveTime

@param keepAliveTime when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.

保持存活时间，翻译意思就是当线程数量大于核心数量，多余的闲置线程在终止前将保持等待新任务直到最大时间为止。即当线程没有任务执行时，继续存活的时间，默认情况下，该参数只在线程数大于corePoolSize时才有用。

unit

@param unit the time unit for the {@code keepAliveTime} argument

单位，翻译意思就是keepAliveTime参数的时间单位。

workQueue

@param workQueue the queue to use for holding tasks before they are executed. This queue will hold only the {@code Runnable} tasks submitted by the {@code execute} method.

工作队列（阻塞队列），翻译意思就是在任务执行前被用来持有任务的队列，这个队列只会持有被execute方法提交的实现Runnable接口的任务。参数值有如下选择：

1、ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务。

2、LinkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene。

3、SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene。

4、priorityBlockingQuene：具有优先级的无界阻塞队列。

threadFactory

@param threadFactory the factory to use when the executor creates a new thread

线程工厂，翻译意思就是执行器创建一个新线程时使用的工厂。

handler

@param handler the handler to use when execution is blocked because the thread bounds and queue capacities are reached

处理者，当执行阻塞使用的处理者，因为线程绑定和队列能力到达上限了。线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

1、AbortPolicy：直接抛出异常，默认策略。

2、CallerRunsPolicy：用调用者所在的线程来执行任务。

3、DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务。

4、DiscardPolicy：直接丢弃任务。

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

线程池创建方式调用ThreadPoolExecutor赋参讲解

继续返回刚刚讨论的创建线程池的问题，我们罗列的四种方式，其中有三种都是调用了如上介绍的ThreadPoolExecutor去创建的，我们也把参数的含义介绍了一便，这里我们针对三种创建方式它们时如何给ThreadPoolExecutor赋参的。

1.newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

这里的核心线程数和最大线程数都是赋死的，并且相等的，使用LinkedBlockingQuene作为阻塞队列。至于那个0和单位毫秒，我也没太搞懂，估测就是设置0的话，就是不考虑存活时间，也就是线程不死不会主动去销毁。

2. newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

这里可以看到没有指定核心数，只是指定了最大为MAX，并设置了默认的存活时间60s，超过了直接销毁。

3. newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

这里可以看到核心数和最大数都指定为1，代表了单线程模式，其实就相当于fixed的创建方式，只不过已经默认设置了线程数为1。

4. 基于调用后然后调用本类的构造器的赋死的俩参数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}
```

threadFactory看样子是用了一个默认的线程工厂，我们一步步去看是啥。

```
public static ThreadFactory defaultThreadFactory() {
    return new DefaultThreadFactory();
}
```

```
DefaultThreadFactory() {
    SecurityManager s = System.getSecurityManager();
    group = (s != null) ? s.getThreadGroup() :
                          Thread.currentThread().getThreadGroup();
    namePrefix = "pool-" +
                  poolNumber.getAndIncrement() +
                 "-thread-";
}
```

这样就明白了我们上面demo获取的线程名是哪来的了。这里提一下，上面我们不是介绍了几种线程池创建方式吗，他们本身都还有一个重载方法，多了一个参数就是赋线程工厂的，所以这个是可以自己去定义的。怎么写呢，就模仿这个默认的呗。

```
: class DefaultThreadFactory implements ThreadFactory {
```

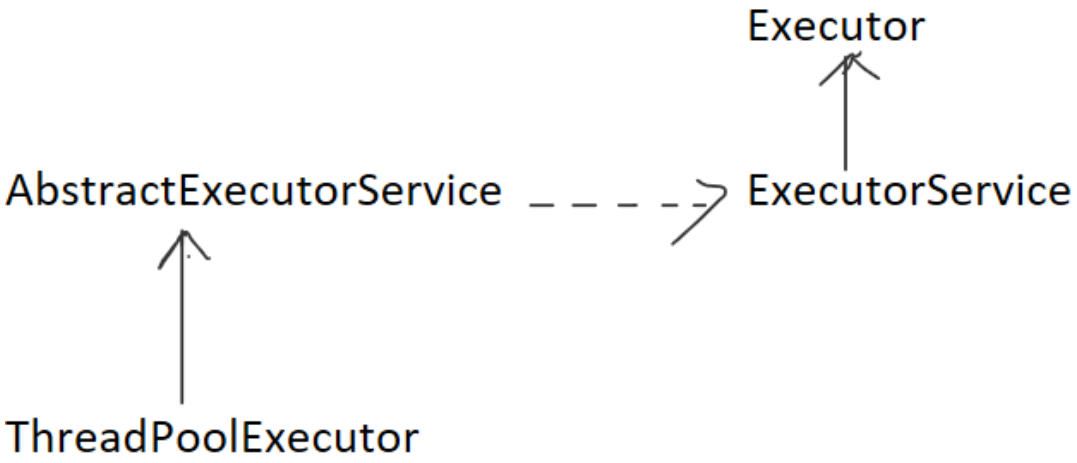
还有个参数就是handler了，看值也是个默认的，我们去看看默认的到底是啥。

```
private static final RejectedExecutionHandler defaultHandler =
    new AbortPolicy();
```

哦，这就是上面提过的直接抛出异常的策略了。

执行原理

以上我们已经大致把创建线程池讲解了下，那么关键还是如何去使用了，demo的里面大家应该也关注到这个了executor.execute(new Task());那我们就来探究下这里面的过程，这里先理清一个继承的结构，上一张图。



实线表示继承，虚线表示实现，可以看到，我们调用execute其实最终是ThreadPoolExecutor去调用的，我们去看看里面的实现。

```
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();
    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);
```

具体的执行流程如下：

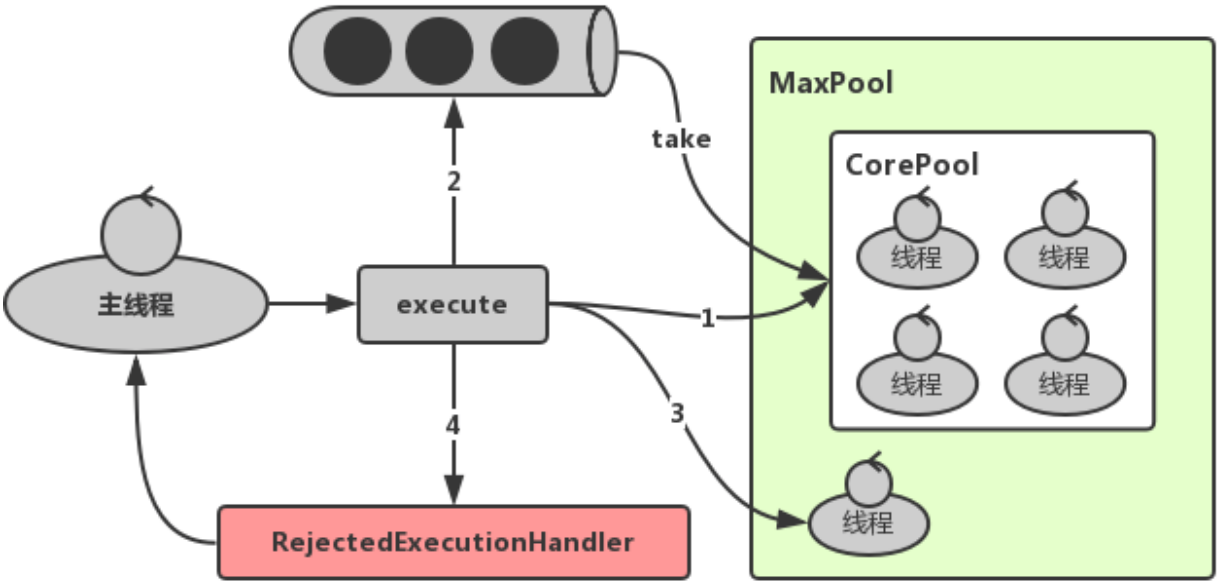
1、workerCountOf方法根据ctl的低29位，得到线程池的当前线程数，如果线程数小于corePoolSize，则执行addWorker方法创建新的线程执行任务；否则执行步骤（2）。

2、如果线程池处于RUNNING状态，且把提交的任务成功放入阻塞队列中，则执行步骤（3），否则执行步骤（4）。

3、再次检查线程池的状态，如果线程池没有RUNNING，且成功从阻塞队列中删除任务，则执行reject方法处理任务。

4、执行addWorker方法创建新的线程执行任务，如果addWoker执行失败，则执行reject方法处理任务。

贴一张图以便理解。



addWorker解读

这个方法就是负责创建线程的。代码的前半部分。

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
```

大致意思就是：

1、判断线程池的状态，如果线程池的状态值大于或等SHUTDOWN，则不处理提交的任务，直接返回。

2、通过参数core判断当前需要创建的线程是否为核心线程，如果core为true，且当前线程数小于corePoolSize，则跳出循环，开始创建新的线程。

接下来代码的后半段就是创建线程的逻辑了。

```
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
```

线程池的工作线程通过Woker类实现，在ReentrantLock锁的保证下，把Woker实例插入到HashSet后，并启动Woker中的线程，其中Worker类设计如下：

1、继承了AQS类，可以方便的实现工作线程的中止操作。

2、实现了Runnable接口，可以将自身作为一个任务在工作线程中执行。

3、当前提交的任务firstTask作为参数传入Worker的构造方法。