线程池

线程池

线程池的重要性和好处

- **重用线程**，避免线程创建与销毁带来的性能开销。
- **加快响应速度**，消除了线程创建带来的延迟。
- **控制线程池的最大并发数**，避免大量线程之间因为互相抢占系统资源而导致的阻塞现象。
- **合理利用CPU和内存**，达到一个资源占用与效率之间的平衡。
- 能够**对线程进行简单管理，**进行统一的分配、调优和监控，并且提供定时执行以及制定间隔循环执行等功能。

线程池解决的问题

- 应对不确定性的**资源分配**
  - 频繁申请和调度资源带来的消耗。
  - 无法对资源的无限申请采取抑制手段。
  - 无法合理管理系统资源分布。
- 以**池化**的思想对资源统一管理

线程池的适用场合

- 比如服务器接收到大量请求时，使用线程池技术是非常合适的，可以大大减少线程的创建和销毁次数，提高服务器的工作效率。
- 实际开发中，如果需要创建5个以上的线程，那么就可以使用线程池来管理。

创建和停止线程

线程池构造函数主要参数

- int corePoolSize：**核心线程数**，线程池在完成初始化以后，默认情况下，线程池中没有线程，线程池会等待有任务到来时，再创建新的线程去完成任务。
- int maximumPoolSize：**最大线程数**，线程池有可能会在核心线程数的基础上，额外增加一些线程，但是这些新增加的线程数会有一个上限，这就是最大量maximumPoolSize。
- long keepAliveTime：存活时间，如果线程池当前的线程数多于corePoolSize，那么多余的线程**空闲时间**超过keepAliveTime，他们就会被终止。这种机制可以在线程池占用资源的过程中减少冗余消耗。默认情况下是只有多于corePoolSize的线程会被回收，除非是修改了**allowCoreThreadTimeOut**属性设置true，那么keepAliveTime也会同样作用于核心线程。
- TimeUnit unit：时间单位。
- BlockingQueue<Runnable> workQueue：**任务存储队列**，3中常见的队列类型：并发队列
  - 直接交接：SynchronousQueue
  - 无界队列：LinkedBlockingQueue
  - 有界的队列：ArrayBlockingQueue

```
1. FixedThreadPool与SingleThreadPool的Queue是LinkedBlockingQueue?
他们的线程数量已经固定了，添加的任务数量无法估计，所以是用了一个可以存储无限多任务的队列来帮助存储任务。
2. CachedThreadPool使用的Queue是SynchronousQueue?
SynchronousQueue内部实际上是不存储的，但是在CachedThreadPool这个线程池的情况下也根本不需要队列来存储，会直接启动新的线程，效率更高些，也不需要一个队列去中转了。
3. ScheduleThreadPool使用的是延迟队列DelayedWorkQueue。
```

- ThreadFactory threadFactory：当线程池需要新的线程时，会使用**线程工厂**来生成新的线程。默认使用 Executor.defaultThreadFacdtory()，创建出来的线程都在同一个线程组，拥有同样的NORM_PRIORITY优先级并且都不是守护线程。如果自己指定ThreadFactory，那么就可以改变线程名、线程组、优先级、是否守护线程等等，只是一般用默认的就足够了。
- RejectedExecutionHandler handler：由于线程池无法接受提交的任务而**拒绝策略**

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

线程的添加规则

1. 如果线程数小于**corePoolSize**，即使其他工作线程处于空闲状态，也会创建一个新线程来执行新任务。
2. 如果线程数等于或者大于corePoolSize但是少于maximumPoolSize，就将任务放入队列**workQueue**。
3. **如果队列已满，并且线程数小于maximumPoolSize，就创建一个新的线程来执行任务。**

4. 如果队列已经满了，并且线程数大于或等于maximumPoolSize，就**拒绝**这个任务。

线程池中增减线程的特点

线程池**希望保持较少数的线程数**，并且只有在负载变得很大时，才增加它。

- 通过设置corePoolSize和maximumPoolSize相同，就可以创建**固定大小**的线程池。
- 通过设置maximumPoolSize为很高的值，例如Integer.MAX_VALUE，可以允许线程池容纳任意数量的并发任务。
- 是只有在队列填满时才创建多于corePoolSize的线程，所以如果使用的是无界队列（例如LinkedBlockingQueue），那么线程数就不会超过corePoolSize。

手动创建和自动创建

手动创建相对来说会更加适合具体场景，因为可以更加明确线程池的运行规则，避免资源耗尽的风险。相反，自动创建线程池，直接调用JDK封装好的构造函数，可能会带来一些问题。

- newFixedThreadPool
  - 由于传进去的LinkedBlockingQueue是没有容量上限的，所以当传入的请求越来越多，并且无法及时处理完毕的时候，就是**请求堆积的时候，会容易造成占用大量的内存，可能会导致OOM**。

```
1 ExecutorService executorService = Executors.newFixedThreadPool(1);
```

- newSingleThreadExecutor
  - 单线程的线程池，它只会用唯一的线程来执行任务。这个和FixedThreadPool的原理基本相同，只是把核心线程数和最大线程数都设置为了1，所以也会导致同样的问题，也就是请求堆积的时候，可能会占用大量的内存。
- CachedThreadPool
  - 可缓存线程，**无界线程池**，具有自动回收多余线程的功能。它的弊端在于第二个参数maximumPoolSize被设置为Integer.MAX_VALUE，这可能会创建数量非常多的线程，甚至直接导致OOM。
- ScheduleThreadPool
  - 支持**定时以及周期性任务执行**的线程池。
- helo项目中使用到的线程池
  - Helo线程池规范使用手册

因此线程池的创建，最好的方式应该是手动创建，**根据不同的业务场景**，自己设置线程池参数，比如我们的内存有多大，想给线程取什么名字等等。

```
1 new ThreadPoolExecutor(nThreads, mThreads,
2                         0L, TimeUnit.MILLISECONDS,
3                         new LinkedBlockingQueue<Runnable>());
```

线程池中线程数量的合适设定

- 一些线程池参数配置方案：

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ $The\ optimal\ pool\ size\ for\ keeping\ the\ processors\ at\ the\ desired\ utilization\ is:$ $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

但是并没有通用的线程池计算方式。并发任务的执行情况和任务类型相关，IO密集型和CPU密集型的任务运行起来的情况差异非常大，但这种占比是较难合理预估的，这导致很难有一个简单有效的通用公式帮我们直接计算出结果。

- CPU密集型（加密、计算hash等）：**最佳线程数为CPU核心数的1-2倍左右**。
- 耗时IO型（读写数据库、文件、网络读写等等）：最佳线程数一般会大于CPU核心数很多倍，以JVM线程监控显示频繁情况为依据，保证线程空闲可以衔接上，参考BrainGoetz推荐的计算方法：线程数=CPU核心数*（1+平均等待时间/平均工作时间）
- Helo CPU\IO 密集型任务划分记录 Helo线程池规范使用手册

线程池参数动态化

- 这是降低修改线程池参数的成本的一种方式，至少能够在发生故障的时候快速调整。

停止线程池的正确方法

1. shutdown：不一定马上停止，仅仅是初始化关闭过程，已经提交的任务会执行完，**新的任务无法再被提交**了。

```
public class ShutDown {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 100; i++) {
            executorService.execute(new ShutDownTask());
        }
        Thread.sleep(1500);
        executorService.shutdown();
        executorService.execute(new ShutDownTask());
        //任务已经无法提交，抛出RejectedExecutionException异常
    }
}
```

```
14
15 class ShutDownTask implements Runnable {
16     @Override
17     public void run() {
18         try {
19             Thread.sleep(500);
20             System.out.println(Thread.currentThread().getName());
21         } catch (InterruptedException e) {
22             System.out.println(Thread.currentThread().getName() + "被中断了");
23         }
24     }
25 }
```

2. isShutdown：返回是否shutdown，即使线程池中的任务还在执行。
3. isTerminated：它可返回整个线程池是不是已经完全停止了，线程和任务队列全部清空。
4. awaitTermination：测试一段时间内线程池是不是会完全停止，起到的作用是检测。这个方法在**返回之前是阻塞**的，只有当线程池中所有任务都执行完毕，或者等待时间到了，以及等待过程中被中断了抛出异常。
5. shutdownNow：**马上停止线程池**，用interrupt信号去通知正在执行的线程停止，而队列当中等待的任务则直接返回List<Runnable>，可以用于在以后通过其他方式执行。

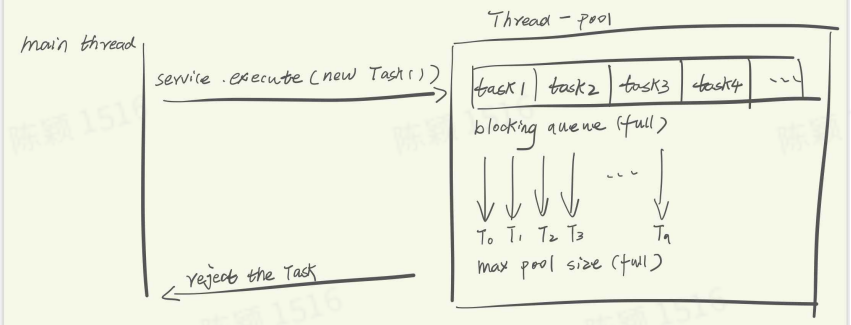
常见线程池的特点和用法

- newFixedThreadPool
- newSingleThreadPool
- CachedThreadPool
- ScheduleThreadPool
- 以及JDK1.8加入的WorkingStealingPool：这个线程池**和其他的有很大的不同**。加入的任务不是普通任务，而且是有**子任务**的情况下会适合这个场景，比如说二叉树遍历、矩阵的处理。使用这个线程池有一定的**窃取**能力，每一个线程之间是会合作的。（使用场景有限）

拒绝任务

拒绝的时机

1. 当**Executor关闭**时，提交新任务会被拒绝。
2. 当Executor对最大线程和工作队列容量使用有限边界并且**已经饱和**时。



四种拒绝策略

1. AbortPolicy：直接抛出一个**异常**。
2. DiscardPolicy：默默地把任务**丢弃**
3. DiscardOldestPolicy：丢弃队列中最老的任务，以便**腾出空间**存储添加的新任务。
4. CallerRunsPolicy：让**提交任务的这个线程去执行任务**。这种方式**避免了任务损失**，也是一种负反馈策略，给了线程池缓冲的时间。

钩子方法

- 在任务执行前后做一些事情，例如日志、埋点等。

```
/**
 * 每个任务执行前后放钩子函数
 */
public class PauseableThreadPool extends ThreadPoolExecutor {

    private boolean isPaused;
    private final ReentrantLock lock = new ReentrantLock();
    private Condition unpaused = lock.newCondition();


    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long keepAliveTime,
            TimeUnit unit,
            BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }


    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        lock.lock();
```

```
22          try {
23              while (isPaused) {
24                  unpaused.await();
25              }
26          } catch (InterruptedException e) {
27              e.printStackTrace();
28          } finally {
29              lock.unlock();
30          }
31      }
32
33      private void pause() {
34          lock.lock();
35          try {
36              isPaused = true;
37          } finally {
38              lock.unlock();
39          }
40      }
41
42      public void resume() {
43          lock.lock();
44          try {
45              isPaused = false;
46              unpaused.signalAll();
47          } finally {
48              lock.unlock();
49          }
50      }
51
52      public static void main(String[] args) throws InterruptedException {
53          PauseableThreadPool pauseableThreadPool = new PauseableThreadPool(10, 20,
    10l,
54                  TimeUnit.SECONDS, new LinkedBlockingQueue<>());
55          Runnable runnable = new Runnable() {
56              @Override
57              public void run() {
58                  System.out.println("线程被执行");
59                  try {
60                      Thread.sleep(10);
61                  } catch (InterruptedException e) {
62                      e.printStackTrace();
63                  }
64              }
```

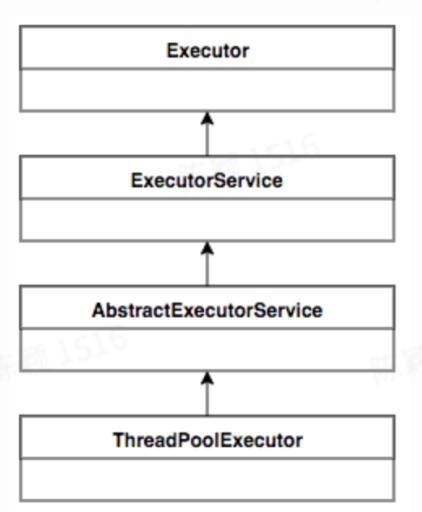
```
        };
        for (int i = 0; i < 10000; i++) {
            pauseableThreadPool.execute(runnable);
        }
        Thread.sleep(1500);
        pauseableThreadPool.pause();
        System.out.println("线程池被暂停");
        Thread.sleep(1500);
        pauseableThreadPool.resume();
        System.out.println("线程池被恢复");

    }
}
```

线程池的实现原理

总体设计

关于ThreadPoolExecutor的继承结构：线程池、ThreadPoolExecutor、ExecutorService、Executor、Executors等，这些和线程池相关的类都是什么关系？

1. ThreadPoolExecutor的UML类图：

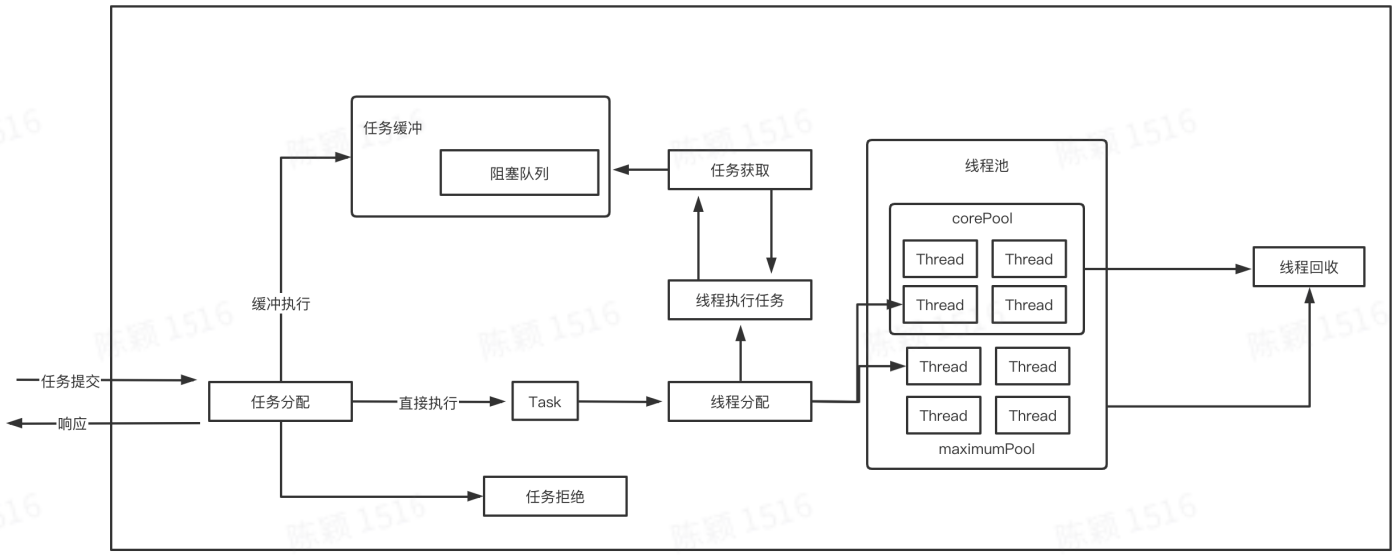


a. Executor：将任务提交和任务执行进行解耦。
b. ExecutorService：（1）扩充执行任务的能力，补充了可以为一个或一批异步任务生成Future的方法；（2）提供了管控线程池的方法，比如停止线程池的运行。
c. AbstractExecutorService：是上层的抽象类，将执行任务的流程串联了起来，保证下层的实现只需关注一个执行任务的方法即可。
d. ThreadPoolExecutor：实现最复杂的运行部分，将会一方面维护自身的生命周期，另一方面同时管理线程和任务，使两者良好结合从而执行并行任务。

2. 而Executors是个工具类，帮助快速创建线程池，最终会调用线程池创建的构造方法。

运行机制

1. ThreadPoolExecutor如何维护线程和执行任务：



线程池在内部实际上构建了一个**生产者消费者模型**，将线程和任务两者解耦，并不直接关联，从而良好的缓冲任务，复用线程。线程池的运行主要分成两部分：任务管理、线程管理。

a. 任务管理部分充当生产者的角色，当任务提交后，线程池会判断该任务后续的流转：（1）直接申请线程执行该任务；（2）缓冲到队列中等待线程执行；（3）拒绝该任务。

b. 线程管理部分是消费者，它们被统一维护在线程池内，根据任务请求进行线程的分配，当线程执行完任务后则会继续获取新的任务去执行，最终当线程获取不到任务的时候，线程就会被回收。

2. 线程池实现线程复用的原理：**相同线程执行不同任务**。源码的分析如下：

线程池中的线程会不停的检测是不是有新任务进来，然后去调用新任务的run方法。

```
executorService.execute(new Task());

public interface Executor {
    void execute(Runnable command);
}


public class ThreadPoolExecutor extends AbstractExecutorService {
    //其它省略...

    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();

```

```
        //ctl记录了线程池状态和线程数
        int c = ctl.get();
        //首先检查当前线程数量是不是小于核心线程数量
        if (workerCountOf(c) < corePoolSize) {
            //如果不够，就创建新的线程，command就是新的任务
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        //检查是不是running状态，如果是就继续放到工作队列中
        if (isRunning(c) && workQueue.offer(command)) {
            //由于在此期间线程可能会被终止了，所以再进行一次判断
            int recheck = ctl.get();
            //如果现在不是正在运行，就将这个任务删除，并且使用拒绝策略
            if (! isRunning(recheck) && remove(command))
                reject(command);
            //如果线程已经减少至0，就需要创建新的线程
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        //能够执行到这里，说明线程已经停止，
        //或者队列已经放不进去了线程数也大于核心线程数了
        //就要增加线程，直到达到最大线程数量。
        else if (!addWorker(command, false))
            reject(command);//如果已经不能再增加了，就拒绝
    }

}


```

生命周期管理

1. 如何维护线程池运行状态

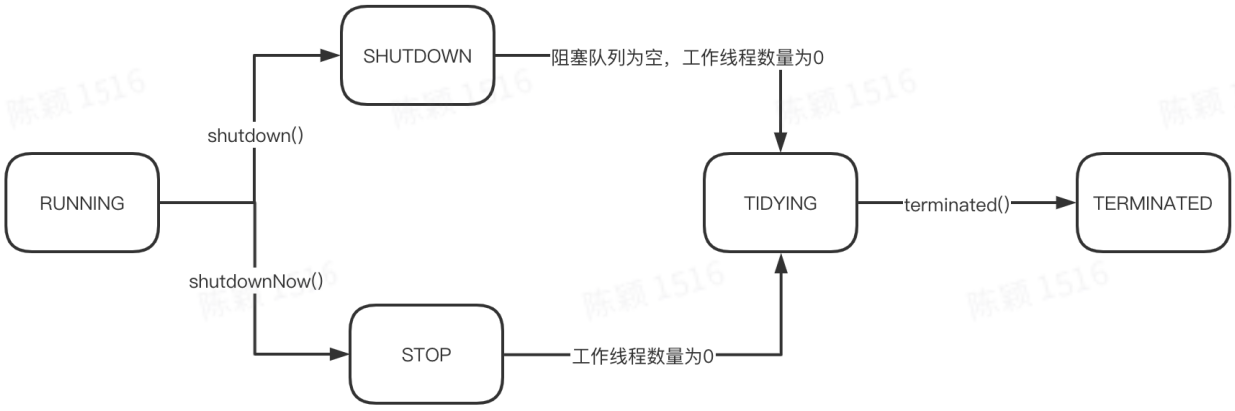
```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

线程池内部使用一个变量维护两个值：运行状态(runState)和线程数量 (workerCount)。高3位保存runState，低29位保存workerCount，两个变量之间互不干扰。用一个变量去存储两个值，可避免在做相关决策时，出现不一致的情况，不必为了维护两者的一致，而占用锁资源。线程池也提供了一些方法去获得线程池当前的运行状态、线程个数，都使用的是位运算的方式，相比于基本运算速度也快很多。

2. 线程池的五种运行状态：

a. RUNNING：接受新任务并且处理排队任务
b. SHUTDOWN：不接受新任务，但处理排队任务
c. STOP：不接受新任务，也不处理排队任务，并且会中断正在运行的线程。
d. TIDYING：中文是整洁，**所有任务都已经终止**，workerCount为零时，线程会转换到TIDYING状态，并且将运行terminated()钩子方法。
e. TERMINATED：terminated()方法运行完成。

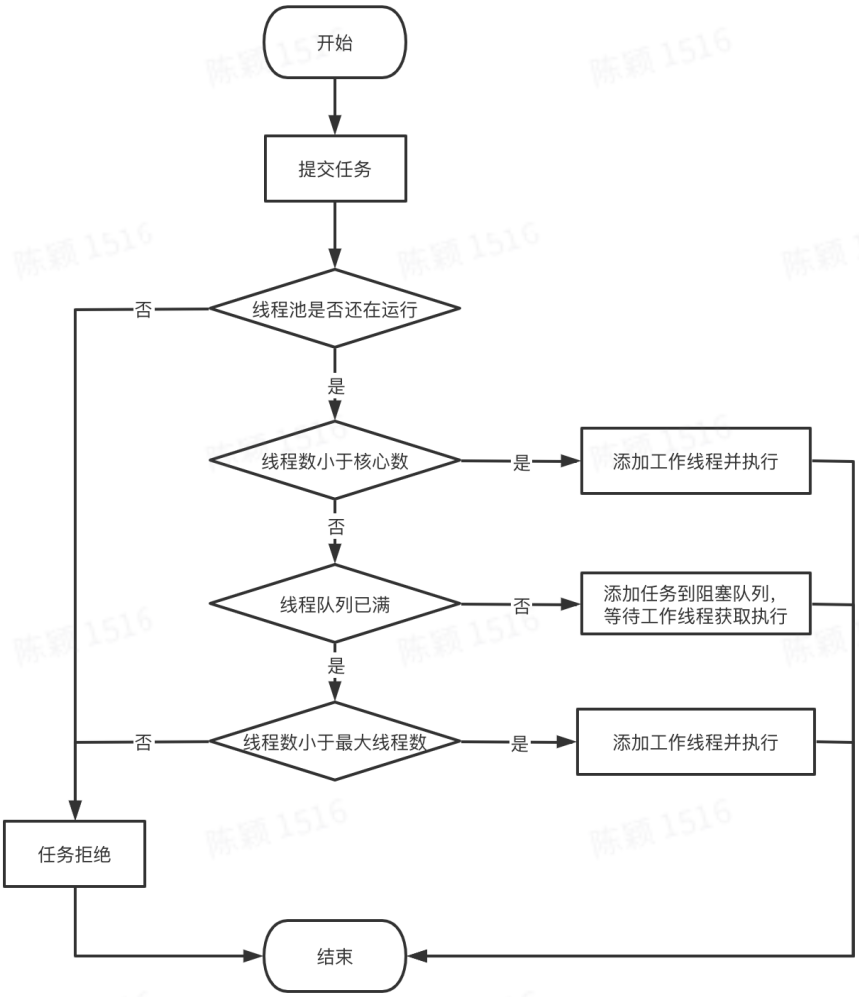
3. 线程池生命周期转换



任务执行机制

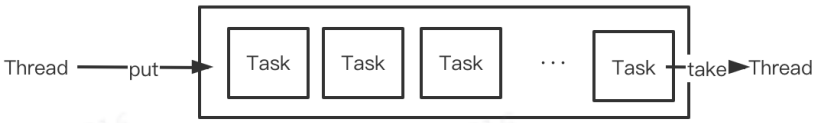
1. 任务调度

当用户提交了一个任务，接下来这个任务将如何执行都是由这个阶段决定的。并且任务的调度都是由execute方法完成的，这部分完成的工作是：检查现在线程池的运行状态、运行线程数、运行策略，决定接下来执行的流程，是直接申请线程执行，或是缓冲到队列中执行，亦或是直接拒绝该任务。



2. 任务缓冲

任务缓冲是线程池能够管理任务的核心部分。线程池的本质是对任务和线程的管理，而做到这一点最关键的思想就是将任务和线程两者解耦，不让两者直接关联，才可以做后续的分配工作。线程池中是以通过一个阻塞队列来实现的生产者消费者模式。

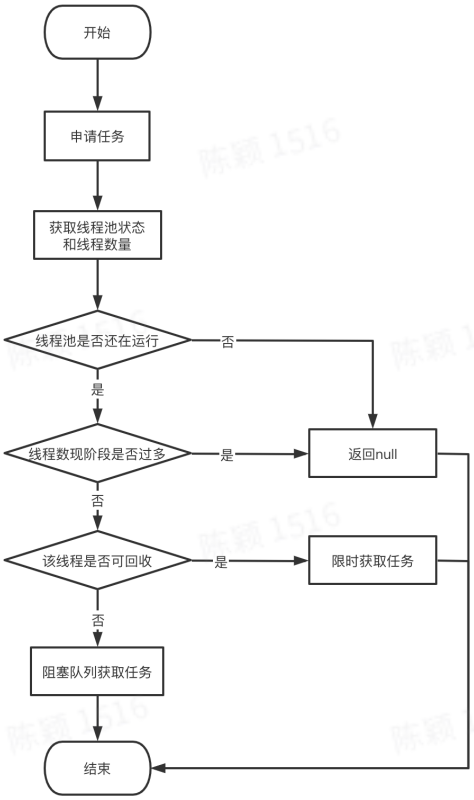


3. 任务申请

任务的执行有两种可能：一种是任务直接由新创建的线程执行；另一种是线程从任务队列中获取任务然后执行，执行完任务的空闲线程会再次去从队列中申请任务再去执行。线程需要从任务缓存模块中不断地取任务执行，帮助线程从阻塞队列中获取任务，实现线程管理模块和任务管理模块之间的通信，这部分策略由**getTask方法**实现：getTask这部分进行了多次判断，为的是**控制线程的数量**，使其符合线程池的状态。如果线程池现在不应该持有那么多线程，则会返回null值。工作线程Worker会不断接收新任务去执行，而当工作线程Worker接收不到任务的时候，就会开始被回收。

```
1 private Runnable getTask() {
2     boolean timedOut = false; // Did the last poll() time out?
3
```

```
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            //因为如果当前线程池的状态处于STOP及以上或队列为空，不能从阻塞队列中获取任务；
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
        //timed变量用于判断是否需要进行超时控制；

        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```



4. 任务拒绝

任务拒绝模块是线程池的保护部分，线程池有一个最大的容量，当线程池的任务缓存队列已满，并且线程池中的线程数目达到maximumPoolSize时，就需要拒绝掉该任务，采取任务拒绝策略，保护线程池。

拒绝策略是一个接口。

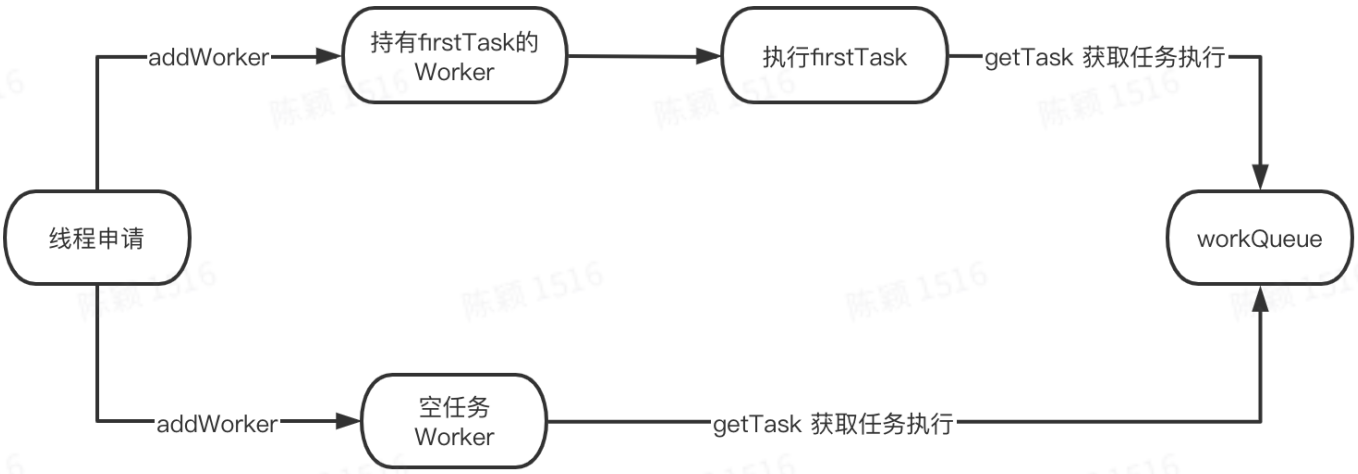
```
1 public interface RejectedExecutionHandler {
2     void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
3 }
```

Worker线程管理

1. Worker线程

线程池为了掌握线程的状态并维护线程的生命周期，设计了线程池内的工作线程Worker。

Worker这个工作线程，实现了Runnable接口，并持有一个线程thread，一个初始化的任务firstTask。thread是在调用构造方法时通过ThreadFactory来创建的线程，可以用来执行任务；firstTask用它来保存传入的第一个任务，这个任务可以有也可以为null。如果这个值是非空的，那么线程就会在启动初期立即执行这个任务，也就对应核心线程创建时的情况；如果这个值是null，那么就需要创建一个线程去执行任务列表（workQueue）中的任务，也就是非核心线程的创建。




```java
private final class Worker
    extends AbstractQueuedSynchronizer
    implements Runnable
{
    /**
     * This class will never be serialized, but we provide a
     * serialVersionUID to suppress a javac warning.
     */
    private static final long serialVersionUID = 6138294804551838833L;

    /** Thread this worker is running in.  Null if factory fails. */
    final Thread thread;
    /** Initial task to run.  Possibly null. */
    Runnable firstTask;
    /** Per-thread task counter */
    volatile long completedTasks;

    /**
     * Creates with given first task and thread from ThreadFactory.
     * @param firstTask the first task (null if none)
     */
    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }

    /** Delegates main run loop to outer runWorker  */
    public void run() {
        runWorker(this);
```

```
    }

    // Lock methods
    //
    // The value 0 represents the unlocked state.
    // The value 1 represents the locked state.

    protected boolean isHeldExclusively() {
        return getState() != 0;
    }

    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }

    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }

    void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}
```

线程池需要管理线程的生命周期，需要在线程长时间不运行的时候进行回收。线程池使用一张Hash表去持有线程的引用，这样可以通过添加引用、移除引用这样的操作来控制线程的生命周期。这个时候重要的就是如何判断线程是否在运行。

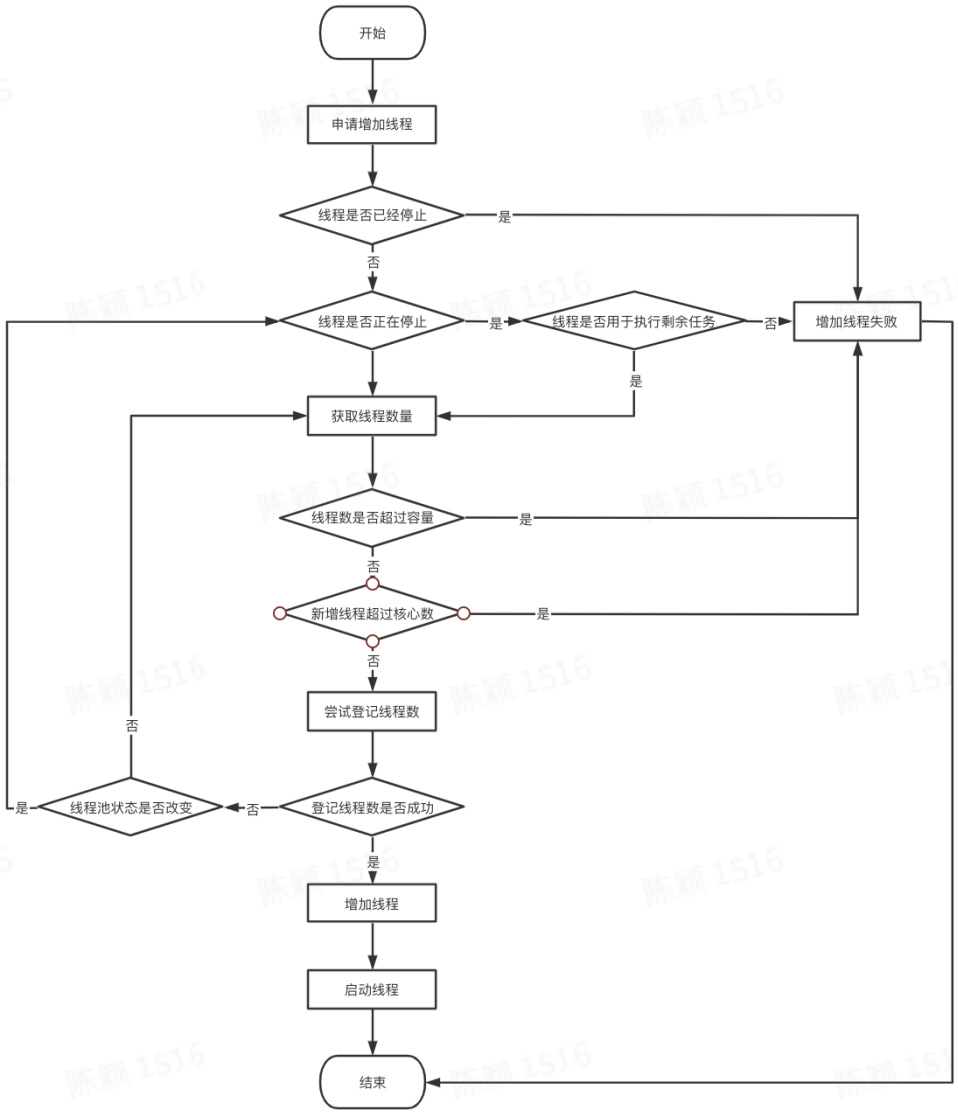
Worker是通过继承AQS，使用AQS来实现独占锁这个功能。没有使用可重入锁ReentrantLock，而是使用AQS，为的就是实现不可重入的特性去反应线程现在的执行状态。1.lock方法一旦获取了独占锁，表示当前线程正在执行任务中。 2.如果正在执行任务，则不应该中断线程。 3.如果该线程现在不是独占锁的状态，也就是空闲的状态，说明它没有在处理任务，这时可以对该线程进行中断。 4.线程池在执行shutdown方法或tryTerminate方法时会调用interruptIdleWorkers方法来中断空闲的线程，interruptIdleWorkers方法会使用tryLock方法来判断线程池中的线程是否是空闲状态；如果线程是空闲状态则可以安全回收。 AQS

2. Worker线程增加

```
private boolean addWorker(Runnable firstTask, boolean core) {
//由于线程执行过程中，各种情况都有可能处于，通过自旋的方式来保证worker的增加;
        retry:
        for (int c = ctl.get();;) {
            // Check if queue empty only if necessary.
            if (runStateAtLeast(c, SHUTDOWN)
                && (runStateAtLeast(c, STOP)
                    || firstTask != null
                    || workQueue.isEmpty()))
                return false;

            for (;;) {
                if (workerCountOf(c)
                    >= ((core ? corePoolSize : maximumPoolSize) & COUNT_MASK))
                    return false;
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                c = ctl.get();  // Re-read ctl
                if (runStateAtLeast(c, SHUTDOWN))
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop
            }
        }

        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            //这个Worker，就是主要的工作相关的类
            w = new Worker(firstTask);
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
```

```
35                 try {
36                     // Recheck while holding lock.
37                     // Back out on ThreadFactory failure or if
38                     // shut down before lock acquired.
39                     int c = ctl.get();
40 
41                     if (isRunning(c) ||
42                         (runStateLessThan(c, STOP) && firstTask == null)) {
43                         if (t.isAlive()) // precheck that t is startable
44                             throw new IllegalThreadStateException();
45                         workers.add(w);
46                         int s = workers.size();
47                         if (s > largestPoolSize)
48                             largestPoolSize = s;
49                         workerAdded = true;
50                     }
51                 } finally {
52                     mainLock.unlock();
53                 }
54                 if (workerAdded) {
55                     t.start();
56                     workerStarted = true;
57                 }
58             }
59         } finally {
60             if (! workerStarted)
61                 addWorkerFailed(w);
62         }
63         return workerStarted;
64     }
```

增加线程是通过线程池中的addWorker方法，功能就是增加一个线程，该方法不考虑线程池是在哪个阶段增加的该线程，这个分配线程的策略是在上个步骤完成的，该步骤仅仅完成增加线程，并使它运行，最后返回是否成功这个结果。addWorker方法有两个参数：firstTask、core。firstTask参数用于指定新增的线程执行的第一个任务，该参数可以为空；core参数为true表示在新增线程时会判断当前活动线程数是否少于corePoolSize，false表示新增线程前需要判断当前活动线程数是否少于maximumPoolSize。



3. Worker线程回收

线程池中线程的销毁依赖JVM自动的回收，线程池做的工作是根据当前线程池的状态维护一定数量的线程引用，防止这部分线程被JVM回收，当线程池决定哪些线程需要回收时，只需要将其引用消除。Worker被创建出来后，就会不断地进行轮询，然后获取任务去执行，核心线程可以无限等待获取任务，非核心线程要限时获取任务。当Worker无法获取到任务，也就是获取的任务为空时，循环会结束，Worker会主动消除自身在线程池内的引用。

```
    //这个方法反应了线程的复用
    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        //获取到Runable类型的任务
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            //只要这个任务不为空，就去执行，getTask()就是从阻塞队列中去取出任务
```

```
            //while循环意味着整个worker不会停止，不停的执行任务，取出新的任务执行
            while (task != null || (task = getTask()) != null) {
                w.lock();
                // If pool is stopping, ensure thread is interrupted;
                // if not, ensure thread is not interrupted.  This
                // requires a recheck in second case to deal with
                // shutdownNow race while clearing interrupt
                if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    try {
                        //执行Runable的run方法
                        task.run();
                        afterExecute(task, null);
                    } catch (Throwable ex) {
                        afterExecute(task, ex);
                        throw ex;
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
            //获取不到任务的时候主动回收自己
        }
    }
```

4. Worker线程执行任务

上面runWorker方法中的执行流程是：1.while循环不断地通过getTask()方法获取任务。2.getTask()方法从阻塞队列中取任务。 3.如果线程池正在停止，那么要保证当前线程是中断状态，否则要保证当前线程不是中断状态。 4.执行任务。 5.如果getTask结果为null则跳出循环，执行processWorkerExit()方法，销毁线程。

线程池使用的注意点

- 避免任务的堆积
- 避免线程数过度增加
- 排查线程泄露：线程执行完毕却不能回收，往往是任务逻辑有问题，导致线程结束不了。

线程池有什么缺点呢

线程池使用面临的核心的问题在于：线程池的参数并不好配置。一方面线程池的运行机制不是很好理解，配置合理需要强依赖个人经验和知识；另一方面，线程池执行的情况和任务类型相关性较大，IO密集型和CPU密集型的任务运行起来的情况差异非常大，这导致业界并没有一些成熟的经验策略帮助开发人员参考。

协程与线程池

使用线程池是为了获取并发性，对于获取并发性，不用线程池而采用协程框架是其中的一个替换方案。但是在JVM平台上的协程框架也是在线程池基础上进行调度的。

参考文档

- 《深入理解Java虚拟机》
- 《实战Java高并发程序设计》
- 《Java并发编程实践》
- 更好的使用JAVA线程池
- Helo线程池使用梳理
- Helo线程池规范使用手册
- Helo线程池梳理

TODO

- [ ] 代码注释补充
- [ ] SAFEGUARD线程池有机会作为兜底吗