**ThreadPoolExecutor源码分析**

1. ThreadPoolExecutor应用方式

为什么使用线程池？避免频繁创建销毁线程。

Jdk中已经提供了Executors，提供很多封装好的线程池。但不适合不同环境，所以需要自己创建线程池。

线程池使用很简单：

·构建好ThreadPoolExecutor对象即可传入指定的参数

·在执行Runnable任务时，可以直接调用execute方法执行

·在执行Callable任务时，需要有返回结果，直接调用submit方法

1. ThreadPoolExecutor核心参数

核心线程数量

最大线程数

非核心线程的空闲时间

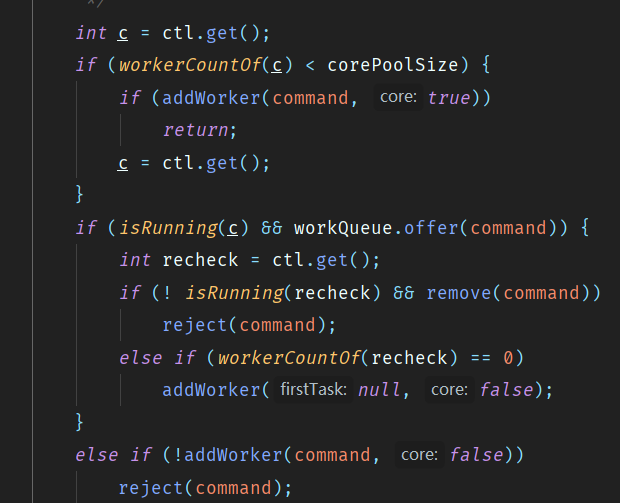
时间单位

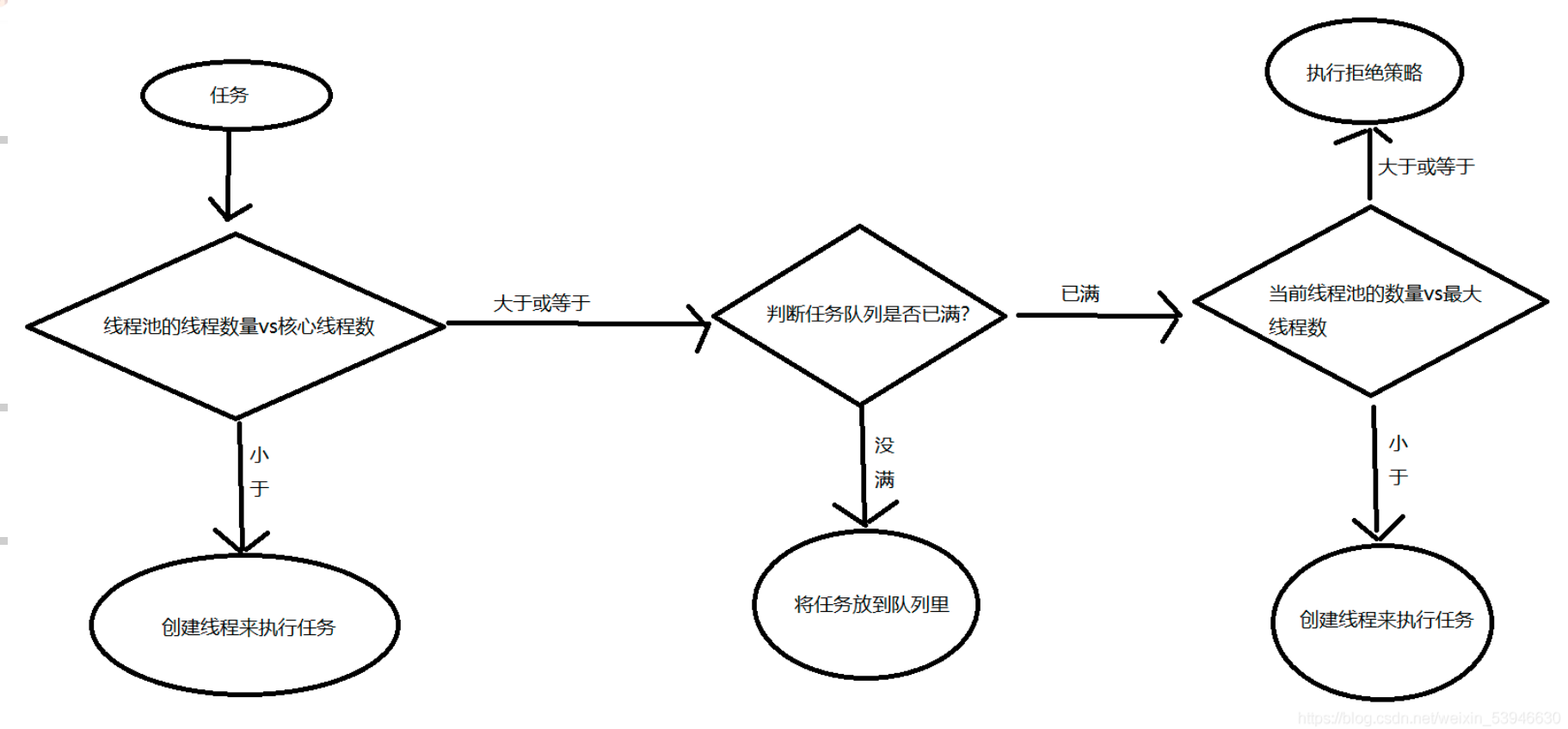
队列容器

线程工厂

拒绝策略 实现RejectedExecutionHandler接口，可以自定义拒绝策略

1. ThreadPoolExecutor 执行流程





1. ThreadPoolExecutor状态

4.1 线程池中核心属性ctl

本质int数值。

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

// Integer.SIZE =32, COUNT\_BITS =29

//ctl表述两个状态：

//1. 表示线程池当前的状态（高三位）

//2. 表示线程池当前的工作的线程个数（低29位）

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

//工作线程的最大数量

//int 32位

00100000 00000000 00000000 00000000

//由于-1 变成

00011111 11111111 11111111 11111111

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

// runState is stored in the high-order bits

高三位

private static final int RUNNING = -1 << COUNT\_BITS; 111

private static final int SHUTDOWN = 0 << COUNT\_BITS; 000

private static final int STOP = 1 << COUNT\_BITS; 001

private static final int TIDYING = 2 << COUNT\_BITS; 010

private static final int TERMINATED = 3 << COUNT\_BITS; 011

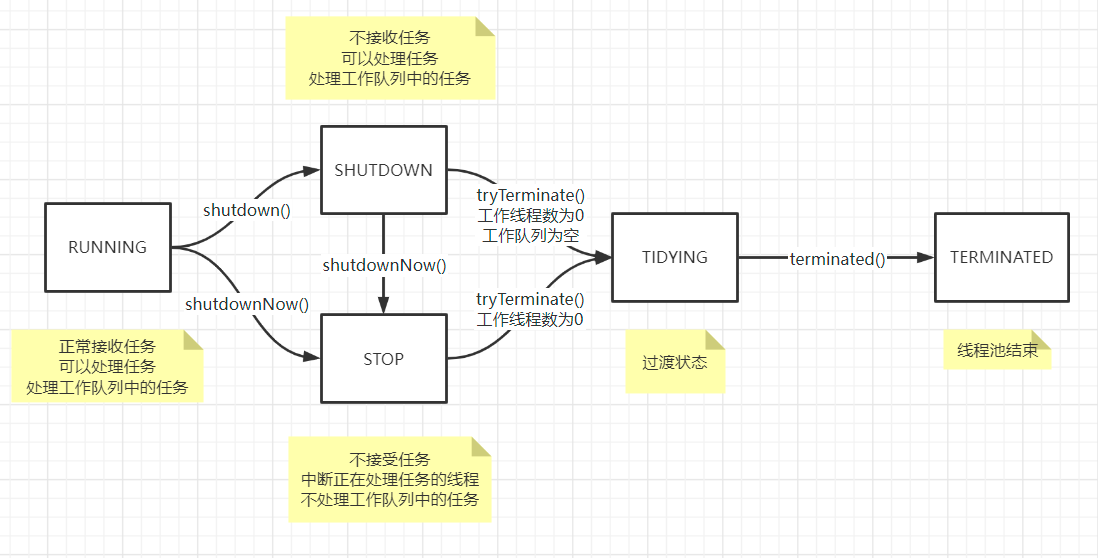
//计算当前线程池的状态

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

//计算当前线程池的工作数量

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

4.2 线程池的状态变换



Running：一旦被创建（初始化）就是此状态，可以接受新任务

Shutdown：调用线程池shutdown接口。不接受新任务，但可以处理已经添加的任务

Stop：调用shutdownNow接口。不接受新任务，不处理已添加的任务。中断正在处理的任务

Tidying：所有任务已终止，线程池会变成tidying状态，此时会调用terminated方法。

Terminated：线程池彻底终止

1. Execute方法

第一点核心：通过execute，查看线程池的整体执行流程，和一些避免并发情况的判断

第二点核心：添加无任务的线程，为了执行队列中的任务

*//获取ctl属性*

*int* c = ctl.get();

//工作线程是否小于核心线程数  
*if* (*workerCountOf*(c) < corePoolSize) {

//通过add方法，添加一个核心线程去执行command任务  
 *if* (addWorker(command, *true*))

//添加成功，返回true，直接return结束  
 *return*;

//当有多个线程同时执行add方法，只有一个能够正确执行，其他的重新获取ctl属性  
 c = ctl.get();  
}

//创建核心线程失败的情况

//判断当前线程池状态是否是RUNNING，如果是RUNNING，执行offer

//将任务添加到工作队列  
*if* (*isRunning*(c) && workQueue.offer(command)) {  
 *int* recheck = ctl.get();

//判断是否是RUNNING状态，不是的话将任务将队列中移除  
 *if* (! *isRunning*(recheck) && remove(command))

//线程池状态不正确，执行拒绝策略  
 reject(command);

//是RUNNING状态或移除任务失败，判断工作线程是否为0  
 *else if* (*workerCountOf*(recheck) == 0)

//工作线程数为0，但是工作队列中有任务在排队

//添加一个空任务非核心线程，为了处理在队列中排队的人  
 addWorker(*null*, *false*);  
}

// 任务添加到任务失败，尝试创建非核心线程  
*else if* (!addWorker(command, *false*))

//添加失败，则拒绝  
 reject(command);

1. addWorker方法

添加工作线程

*private boolean* addWorker(Runnable firstTask, *boolean* core) {

// 前半部分操作 对线程池状态的判断，以及对工作线程数的判断

// 外层for循环的标识  
 retry:  
 *for* (;;) {

// 获取ctl数值  
 *int* c = ctl.get();

// 拿到线程池状态  
 *int* rs = *runStateOf*(c);  
  
 *// Check if queue empty only if necessary.*

// SHUTDOWN =0 RUNNING=-1

// 当前rs大于等于0，不是RUNNING状态，则再次做后续判断，查看当前任务是否可以不处理 *if* (rs >= SHUTDOWN &&

//不满足 任务为空，但队列有数据，对应核心点2的逻辑  
 ! (rs == SHUTDOWN && firstTask == *null* &&! workQueue.isEmpty()))  
 *return false*;  
  
 *for* (;;) {

// 获取工作线程数量  
 *int* wc = *workerCountOf*(c);

// 判断工作线程是否大于最大值  
 *if* (wc >= CAPACITY ||

// 如果是核心线程则判断是否大于核心线程数  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 *return false*;

// 以CAS的方式，对工作线程数+1，如果成功  
 *if* (compareAndIncrementWorkerCount(c))

// 直接跳出外循环，  
 *break* retry;

// 由于并发，导致CAS没有成功，则重新拿到ctl，判断和之前的rs是否一致  
 c = ctl.get(); *// Re-read ctl  
 if* (*runStateOf*(c) != rs)

// 说明并发操作导致线程池状态变化，需要重新判断状态  
 *continue* retry;  
 }  
 }

// 后半部分操作 添加工作线程，并启动工作线程

// 工作线程是否启动  
 *boolean* workerStarted = *false*;

// 工作线程是否添加  
 *boolean* workerAdded = *false*;

Worker w = *null*;  
 *try* {

// 构建工作线程，将任务扔到worker中  
 w = *new* Worker(firstTask);

// 拿到worker中绑定的thread线程  
 *final* Thread t = w.thread;  
 *if* (t != *null*) {

// 避免在添加工作线程的时候，执行了shutdown或shutdownNow方法，

// 这两个方法内部也使用了同一个mainLock  
 *final* ReentrantLock mainLock = *this*.mainLock;  
 mainLock.lock();  
 *try* {

*int* rs = *runStateOf*(ctl.get());  
 // 如果满足线程池状态为RUNNING，就添加工作  
 *if* (rs < SHUTDOWN ||

// 如果为SHUTDOWN，并且传入的任务是null  
 (rs == SHUTDOWN && firstTask == *null*)) {

//开始添加工作线程

// 判断当前线程是否属于RUN状态（健壮性判断）  
 *if* (t.isAlive()) *// precheck that t is startable  
 throw new* IllegalThreadStateException();

// 将构建好的worker添加到HashSet中  
 workers.add(w);  
 *int* s = workers.size();

// 如果现在的工作线程数大于历史最大线程数，重新赋值largeSize  
 *if* (s > largestPoolSize)  
 largestPoolSize = s;

// 工作线程添加成功true  
 workerAdded = *true*;  
 }  
 } *finally* {  
 mainLock.unlock();  
 }  
 *if* (workerAdded) {

// 启动线程执行  
 t.start();  
 workerStarted = *true*;  
 }  
 }  
 } *finally* {

// 如果启动工作线程失败！  
 *if* (! workerStarted)

// 移除workers中的工作线程，工作线程数-1，尝试将线程池状态改为TIDYING  
 addWorkerFailed(w);  
 }  
 *return* workerStarted;  
}

*private void* addWorkerFailed(Worker w) {  
 *final* ReentrantLock mainLock = *this*.mainLock;  
 mainLock.lock();  
 *try* {

// 如果之前创建的工作线程成功，则将hashSet的工作线程移除  
 *if* (w != *null*)  
 workers.remove(w);

// 工作线程数减一  
 decrementWorkerCount();

// 将线程池状态 TIDYING  
 tryTerminate();  
 } *finally* {  
 mainLock.unlock();  
 }  
}

1. Worker对象

*// 继承AQS，基于state做判断，是否可以线程中断*

*// 实现Runnable，存储需要执行的任务*

*private final class* Worker *extends* AbstractQueuedSynchronizer *implements* Runnable  
{

// 初始化构建出来的  
 *final* Thread thread;  
 // 需要执行的任务

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) {

// 刚刚初始化的线程不允许被中断（还没有执行runWorker）  
 setState(-1); *// inhibit interrupts until runWorker*

// 将任务赋值（可能是null） *this*.firstTask = firstTask;

// 构建thread对象  
 *this*.thread = getThreadFactory().newThread(*this*);  
 }  
  
 // 重写run *public void* run() {  
 runWorker(*this*);  
 }

// 中断线程不是立即让线程挺自豪，只是将thread的中断标识设置为true

// 以下都是判断状态 *protected boolean* isHeldExclusively() {  
 *return* getState() != 0;  
 }  
  
 *protected boolean* tryAcquire(*int* unused) {  
 *if* (compareAndSetState(0, 1)) {  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 *return true*;  
 }  
 *return false*;  
 }

// 将state置为0，在runWorker中，为了表示当前线程允许被中断  
 *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) {  
 }  
 }  
 }  
}

1. runWorker方法

*final void* runWorker(Worker w) {

// 拿到当前工作线程  
 Thread wt = Thread.*currentThread*();

// 拿到worker中封装的任务给task  
 Runnable task = w.firstTask;

// 将worker中的task归为null  
 w.firstTask = *null*;

// 允许中断，将state 置为 0，说明线程可以中断  
 w.unlock(); *// allow interrupts*

// 任务执行时，勾子函数是否出现异常 *boolean* completedAbruptly = *true*;  
 *try* {

// 获取任务的第一种方式，就是执行execute、submit时，传入的任务直接处理

// 获取任务的第二种方式，在工作队列中拿到任务  
 *while* (task != *null* || (task = getTask()) != *null*) {

// 加锁，在shutdown状态下，当前线程不许被中断

// 并且worker内部实现的锁，并不是可重入锁，因为在中断时，也需要对worker进行lock，不能获取当前线程的任务  
 w.lock();  
 // 如果线程池状态为STOP，必须将当前线程中断

*if* ((*runStateAtLeast*(ctl.get(), STOP) || (Thread.*interrupted*() &&  
 *runStateAtLeast*(ctl.get(), STOP))) &&

// 查询当前线程中断标记是否为false，如果是false，执行interrupt  
 !wt.isInterrupted())

// 将中断标记设置为true  
 wt.interrupt();  
 *try* {

// 勾子函数 前置函数，可以自定义  
 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* {

// 勾子函数 后置函数，可以自定义  
 afterExecute(task, thrown);  
 }  
 } *finally* {  
 task = *null*;  
 w.completedTasks++;  
 w.unlock();  
 }  
 }

// 说明 勾子函数正确没有抛出异常  
 completedAbruptly = *false*;  
 } *finally* {  
 processWorkerExit(w, completedAbruptly);  
 }  
}

1. getTask方法

如何从工作队列中获取任务

*private* Runnable getTask() {

// 标识（非核心线程可以干掉）  
 *boolean* timedOut = *false*; *// Did the last poll() time out?  
 // 死循环   
 for* (;;) {  
 *int* c = ctl.get();  
 *int* rs = *runStateOf*(c);  
  
 *// Check if queue empty only if necessary.  
 if* (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {  
 decrementWorkerCount();  
 *return null*;  
 }

// ==============判断巩固走线程数量==============

*int* wc = *workerCountOf*(c);  
  
 *// Are workers subject to culling?  
 boolean* timed = allowCoreThreadTimeOut || wc > corePoolSize;  
  
 *if* ((wc > maximumPoolSize || (timed && timedOut))  
 && (wc > 1 || workQueue.isEmpty())) {

// 基于CAS移除掉当前线程  
 *if* (compareAndDecrementWorkerCount(c))

// 返回null，交给pocessWorkerExit移除当前工作线程  
 *return null*;  
 *continue*;  
 }

// ================== 从队列中拿任务 =================  
 *try* {

Runnable r = timed ?

//阻塞一定时间从工作队列中拿任务（可以理解非核心线程）  
 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :

// 一直阻塞（可以理解核心线程）  
 workQueue.take(); // 如果数量为0，有await操作  
 *if* (r != *null*)  
 *return* r;

// 从队列获取任务时超时了，当timeOut=true时，再次循环的时候，timed && timedOut可能成为true，然后执行基于CAS移除掉当前线程。  
 timedOut = *true*;  
 } *catch* (InterruptedException retry) {  
 timedOut = *false*;  
 }  
 }  
}

1. processWorkerExit方法

移除当前工作线程操作

*private void* processWorkerExit(Worker w, *boolean* completedAbruptly) {  
 // 一般是勾子函数中抛出异常，

*if* (completedAbruptly) *// If abrupt, then workerCount wasn't adjusted* decrementWorkerCount();  
  
 *final* ReentrantLock mainLock = *this*.mainLock;  
 mainLock.lock();  
 *try* {  
 completedTaskCount += w.completedTasks;  
 workers.remove(w);  
 } *finally* {  
 mainLock.unlock();  
 }

// 尝试将线程池状态修改  
 tryTerminate();  
  
 *int* c = ctl.get();  
 *if* (*runStateLessThan*(c, STOP)) {

// 进到这，说明线程池状态时RUNNING、SHUTDOWN  
 *if* (!completedAbruptly) {

// 正常状态移除当前工作线程  
 *int* min = allowCoreThreadTimeOut ? 0 : corePoolSize;

// 如果任务不为空，设置工作线程最小值为1  
 *if* (min == 0 && ! workQueue.isEmpty())  
 min = 1;

// 还有工作线程在线程池中  
 *if* (*workerCountOf*(c) >= min)  
 *return*; *// replacement not needed* }

// 1. 说明是不正常的方式移除了当前工作线程，再添加一个工作线程

// 2. 线程池工作队列不为空，并且没有工作线程，再添加一个工作线程  
 addWorker(*null*, *false*);  
 }  
}

Spring中的多线程知识点

1. @Async

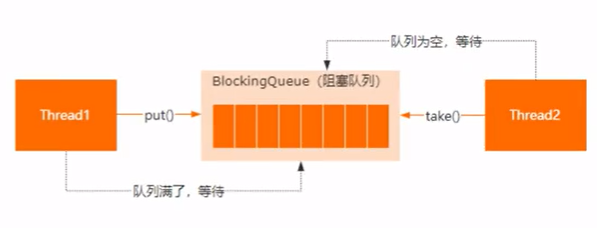
创建代理方法，在线程池中运行。

失效：

·异步方法和调用者方法在同一个类中

原因：spring在扫描bean的时候会扫描方法上是否包含Async，如果包含，spring会为这个bean动态生成一个代理类（proxy），代理类是继承原来的bean的。所以当有此注解的方法被调用时，实际上是由代理类来调用的，代理类在调用时增加了异步操作。然而，如果是同一个类下的方法调用了此方法，那么该方法没有通过代理类调用，而是直接调用的原来的bean的方法，没有增加异步操作，也就导致Async失效。

1. Volatile
2. ArrayBlockingQueue



数组结构的阻塞队列，数组长度有限，使用循环数组。

在队列的基础上增加两个附加操作。

·当队列为空时，获取元素的线程会等待队列变为非恐。

·当队列满了，存储元素的线程会等待队列为可用。

可以很好的实现生产者和消费者模型

主要技术：队列元素存储和线程阻塞和唤醒，阻塞和唤醒用到了JUC包里的ReentrantLock和Condition。Condition相当于wait/notify

1. 线程池如何知道一个线程的任务完成了？

·通过submit进行任务的执行，再通过Future获得任务的执行结果，任务没有完成前，future.get()方法会一直阻塞，直到任务结束。所以当get方法正常返回，就说明任务完成。

·通过引入CountDownLatch作为计数器， await()阻塞线程，countDown()进行倒计时，当countDown值为0的时候，说明正在执行的线程完成。