什么是线程池

在 Java 中，如果每个请求到达就创建一个新线程，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。
如果在一个 Jvm 里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足
为了解决这个问题,就有了线程池的概念，线程池的核心逻辑是提前创建好若干个线程放在一个容器中。如果有任务需要处理，则将任务直接分配给线程池中的线程来执行就行，任务处理完以后这个线程不会被销毁，而是等待后续分配任务。同时通过线程池来重复管理线程还可以避免创建大量线程增加开销。

线程池的优势

合理的使用线程池，可以带来一些好处
1. 降低创建线程和销毁线程的性能开销
2. 提高响应速度，当有新任务需要执行是不需要等待线程创建就可以立马执行
3. 合理的设置线程池大小可以避免因为线程数超过硬件资源瓶颈带来的问题

Java 中提供的线程池 API

我相信有很多同学或多或少都接触过线程池，也可能自己也研究过线程池的原理。前面部分的内容会相对简单点，但是要想合理的使用线程池，那么势必要对线程池的原理有比较深的理解

线程池的使用

要了解一个技术，我们仍然是从使用开始。JDK 为我们提供了几种不同的线程池实现。我们先来通过一个简单的案例来引入线程池的基本使用
在 Java 中怎么创建线程池呢？下面这段代码演示了创建三个固定线程数的线程池

```
public class Test implements Runnable{
    @Override
    public void run() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
    }
    System.out.println(Thread.currentThread().getName());
}
static ExecutorService service=Executors.newFixedThreadPool(3);
public static void main(String[] args) {
    for(int i=0;i<100;i++) {
        service.execute(new Test());
    }
    service.shutdown();
}
```

Java 中提供的线程池 Api

为了方便大家对于线程池的使用，在 Executors 里面提供了几个线程池的工厂方法，这样，很多新手就不需要了解太多关于 ThreadPoolExecutor 的知识了，他们只需要直接使用 Executors 的工厂方法，就可以使用线程池：

newFixedThreadPool：该方法返回一个固定数量的线程池，线程数不变，当有一个任务提交时，若线程池中空闲，则立即执行，若没有，则会被暂缓在一个任务队列中，等待有空闲的线程去执行。

newSingleThreadExecutor: 创建一个线程的线程池，若空闲则执行，若没有空闲线程则暂缓在任务队列中。

newCachedThreadPool：返回一个可根据实际情况调整线程个数的线程池，不限制最大线程数量，若用空闲的线程则执行任务，若无任务则不创建线程。并且每一个空闲线程会在 60 秒后自动回收

newScheduledThreadPool: 创建一个可以指定线程的数量的线程池，但是这个线程池还带有延迟和周期性执行任务的功能，类似定时器。

ThreadpoolExecutor

上面提到的四种线程池的构建，都是基于 ThreadpoolExecutor 来构建的，小伙伴们打起精神来了，接下来将一起了解一下面试官最喜欢问到的一道面试题"请简单说下你知道的线程池和 ThreadPoolThread 有哪些构造参数"

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
```

```
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>());
}
```

ThreadpoolExecutor 有多个重载的构造方法，我们可以基于它最完整的构造方法来分析

先来解释一下每个参数的作用，稍后我们在分析源码的过程中再来详细了解参数的意义。

```
public ThreadPoolExecutor(int corePoolSize, //核心线程数量
                          int maximumPoolSize, //最大线程数
                          long keepAliveTime, //超时时间,超出核心线程数量以外的线程空余存活时间
                          TimeUnit unit, //存活时间单位
                          BlockingQueue<Runnable> workQueue, //保存执行任务的队列
                          ThreadFactory threadFactory,//创建新线程使用的工厂
                          RejectedExecutionHandler handler //当任务无法执行的时候的处理方式)
```

这个地方有很多同学问过我，线程池初始化以后做了什么事情

线程池初始化时是没有创建线程的，线程池里的线程的初始化与其他线程一样，但是在完成任务以后，该线程不会自行销毁，而是以挂起的状态返回到线程池。直到应用程序再次向线程池发出请求时，线程池里挂起的线程就会再度激活执行任务。这样既节省了建立线程所造成的性能损耗，也可以让多个任务反复重用同一线程，从而在应用程序生存期内节约大量开销

newFixedThreadPool

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

FixedThreadPool 的核心线程数和最大线程数都是指定值，也就是说当线程池中的线程数超过核心线程数后，任务都会被放到阻塞队列中。另外 keepAliveTime 为 0，也就是超出核心线程数量以外的线程空余存活时间

而这里选用的阻塞队列是 LinkedBlockingQueue，使用的是默认容量 Integer.MAX_VALUE，相当于没有上限

这个线程池执行任务的流程如下：

1. 线程数少于核心线程数，也就是设置的线程数时，新建线程执行任务

2. 线程数等于核心线程数后，将任务加入阻塞队列

3. 由于队列容量非常大，可以一直添加

4. 执行完任务的线程反复去队列中取任务执行

用途：FixedThreadPool 用于负载比较大的服务器，为了资源的合理利用，需要限制当前线程数量

newCachedThreadPool

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

CachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程；并且没有核心线程，非核心线程数无上限，但是每个空闲的时间只有 60 秒，超过后就会被回收。

它的执行流程如下：

1. 没有核心线程，直接向 SynchronousQueue 中提交任务
2. 如果有空闲线程，就去取出任务执行；如果没有空闲线程，就新建一个
3. 执行完任务的线程有 60 秒生存时间，如果在这个时间内可以接到新任务，就可以继续活下去，否则就被回收

newSingleThreadExecutor

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行

线程池的实现原理分析

线程池的基本使用我们都清楚了，接下来我们来了解一下线程池的实现原理

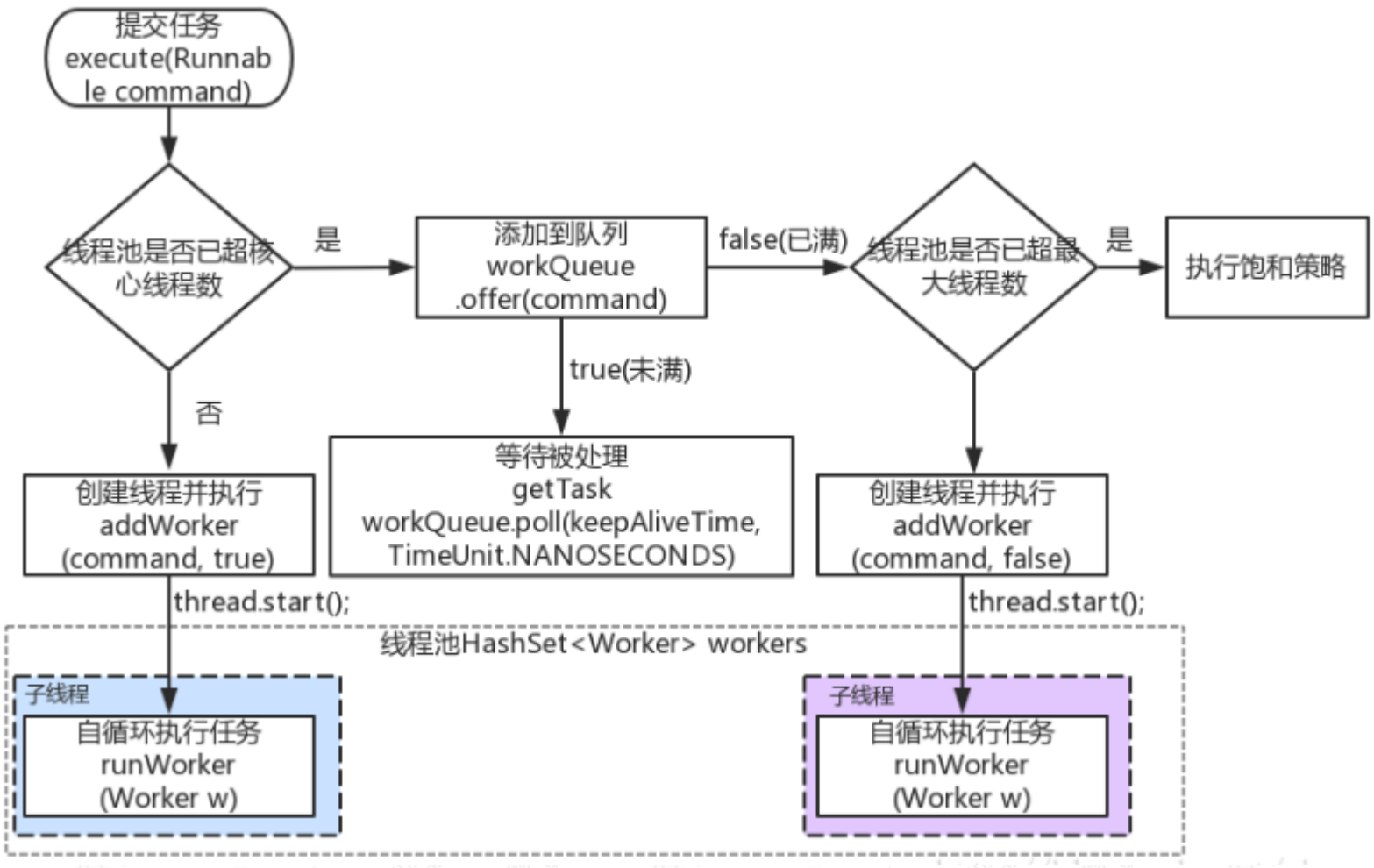
ThreadPoolExecutor 是线程池的核心，提供了线程池的实现。

ScheduledThreadPoolExecutor 继承了 ThreadPoolExecutor，并另外提供一些调度方法以支持定时和周期任务。Executers 是工具类，主要用来创建线程池对象

我们把一个任务提交给线程池去处理的时候，线程池的处理过程是什么样的呢？首先直接来

看看定义

线程池原理分析(FixedThreadPool)



源码分析

execute

基于源码入口进行分析，先看 execute 方法

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
      int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {//1.当前池中线程比核心数少，新建一个线程执行任务
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {//2.核心池已满，但任务队列未满，添加到队列中
        int recheck = ctl.get();
```

```
//任务成功添加到队列以后，再次检查是否需要添加新的线程，因为已存在的线程可能被销毁了
        if (! isRunning(recheck) && remove(command))
            reject(command);//如果线程池处于非运行状态，并且把当前的任务从任务队列中移除成功，则拒绝该任务
        else if (workerCountOf(recheck) == 0)//如果之前的线程已被销毁完，新建一个线程
            addWorker(null, false);
    }
    else if (!addWorker(command, false))//3.核心池已满，队列已满，试着创建一个新线程
        reject(command);  //如果创建新线程失败了，说明线程池被关闭或者线程池完全满了，拒绝任务
}
```

ctl 的作用

在线程池中，ctl 贯穿在线程池的整个生命周期中

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

它是一个原子类，主要作用是用来保存线程数量和线程池的状态。我们来分析一下这段代码，其实比较有意思，他用到了位运算

一个 int 数值是 32 个 bit 位，这里采用高 3 位来保存运行状态，低 29 位来保存线程数量。

我们来分析默认情况下，也就是 ctlOf(RUNNING)运行状态，调用了 ctlOf(int rs,int wc)方法；

其中

```
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

其中 RUNNING =-1 << COUNT_BITS ； -1 左移 29 位. -1 的二进制是 32 个 1（1111 1111 1111 1111 1111 1111 1111 1111)

-1 的二进制计算方法

原码是 1000...001 ．高位 1 表示符号位。

然后对原码取反，高位不变得到 1111...110

然后对反码进行+1 ，也就是补码操作， 最后得到 1111...1111

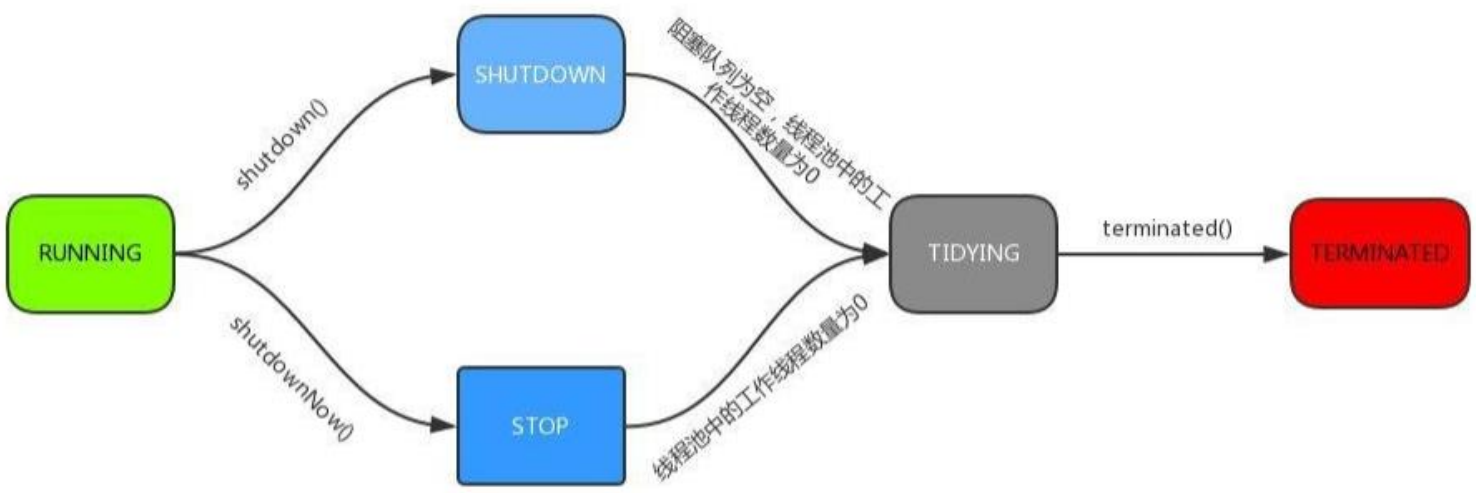
那么-1 <<左移 29 位， 也就是 【111】 表示； rs | wc 。二进制的 111 | 000 。得到的结果仍然是 111

那么同理可得其他的状态的 bit 位表示

```
private static final int COUNT_BITS = Integer.SIZE - 3; //32-3
private static final int CAPACITY   = (1 << COUNT_BITS) - 1; //将 1 的二进制向右位移 29 位,再减 1 表示最大线程容量

//运行状态保存在 int 值的高 3 位 (所有数值左移 29 位)
private static final int RUNNING    = -1 << COUNT_BITS;// 接收新任务,并执行队列中的任务
private static final int SHUTDOWN   =  0 << COUNT_BITS;// 不接收新任务,但是执行行队列中的任务
private static final int STOP       =  1 << COUNT_BITS;// 不接收新任务,不执行队列中的任务,中断正在执行中的任务
private static final int TIDYING    =  2 << COUNT_BITS; //所有的任务都已结束,线程数量为 0,处于该状态的线程池即将调用 terminated()方法
private static final int TERMINATED =  3 << COUNT_BITS;// terminated()方法执行完成
```

状态转化



addWorker

如果工作线程数小于核心线程数的话，会调用 addWorker，顾名思义，其实就是要创建一个工作线程。我们来看看源码的实现

源码比较长，看起来比较唬人，其实就做了两件事。1）才用循环 CAS 操作来将线程数加 1；2）新建一个线程并启用。

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:  //goto 语句,避免死循环
    for (;;) {
```

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

        // Check if queue empty only if necessary.
```

如果线程处于非运行状态，并且 rs 不等于 SHUTDOWN 且 firstTask 不等于空且且 workQueue 为空，直接返回 false（表示不可添加 work 状态）

1．线程池已经 shutdown 后，还要添加新的任务，拒绝

2．（第二个判断）SHUTDOWN 状态不接受新任务，但仍然会执行已经加入任务队列的任务，所以当进入 SHUTDOWN 状态，而传进来的任务为空，并且任务队列不为空的时候，是允许添加新线程的,如果把这个条件取反，就表示不允许添加 worker

```
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) { //自旋
            int wc = workerCountOf(c);//获得 Worker 工作线程数
```

//如果工作线程数大于默认容量大小或者大于核心线程数大小，则直接返回 false 表示不能再添加 worker。

```
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
```

if (compareAndIncrementWorkerCount(c))//通过 cas 来增加工作线程数，如果 cas 失败，则直接重试

```
                break retry;
            c = ctl.get();  // Re-read ctl //再次获取 ctl 的值
            if (runStateOf(c) != rs) //这里如果不想等，说明线程的状态发生了变化，继续重试
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
```

```
//上面这段代码主要是对 worker 数量做原子+1 操作,下面的逻辑才是正式构建一个 worker
boolean workerStarted = false; //工作线程是否启动的标识
boolean workerAdded = false;  //工作线程是否已经添加成功的标识
Worker w = null;
try {
    w = new Worker(firstTask); //构建一个 Worker，这个 worker 是什么呢？我们
可以看到构造方法里面传入了一个 Runnable 对象
    final Thread t = w.thread; //从 worker 对象中取出线程
    if (t != null) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock(); //这里有个重入锁，避免并发问题
        try {
            // Recheck while holding lock.
            // Back out on ThreadFactory failure or if
            // shut down before lock acquired.
            int rs = runStateOf(ctl.get());
            //只有当前线程池是正在运行状态，[或是 SHUTDOWN 且 firstTask 为空]，才
能添加到 workers 集合中
            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                 //任务刚封装到 work 里面，还没 start,你封装的线程就是 alive，几
个意思？肯定是要抛异常出去的
                if (t.isAlive()) // precheck that t is startable
                    throw new IllegalThreadStateException();
                workers.add(w); //将新创建的 Worker 添加到 workers 集合中
                int s = workers.size();
//如果集合中的工作线程数大于最大线程数，这个最大线程数表示线程池曾经出现过的最大线程数
                if (s > largestPoolSize)
                    largestPoolSize = s; //更新线程池出现过的最大线程数
                workerAdded = true;//表示工作线程创建成功了
            }
        } finally {
            mainLock.unlock(); //释放锁
```

```
                }
                if (workerAdded) {//如果 worker 添加成功
                    t.start();//启动线程
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w); //如果添加失败，就需要做一件事，就是递减实际工作线
程数(还记得我们最开始的时候增加了工作线程数吗)
            }
            return workerStarted;//返回结果
    }
```

Worker 类说明

我们发现 addWorker 方法只是构造了一个 Worker，并且把 firstTask 封装到 worker 中，它是做什么的呢？我们来看看

1. 每个 worker,都是一条线程,同时里面包含了一个 firstTask,即初始化时要被首先执行的任务.
2. 最终执行任务的,是 runWorker()方法

Worker 类继承了 AQS，并实现了 Runnable 接口，注意其中的 firstTask 和 thread 属性：firstTask 用它来保存传入的任务；thread 是在调用构造方法时通过 ThreadFactory 来创建的线程，是用来处理任务的线程。

在调用构造方法时，需要传入任务，这里通过 getThreadFactory().newThread(this);来新建一个线程，newThread 方法传入的参数是 this，因为 Worker 本身继承了 Runnable 接口，也就是一个线程，所以一个 Worker 对象在启动的时候会调用 Worker 类中的 run 方法。

Worker 继承了 AQS，使用 AQS 来实现独占锁的功能。为什么不使用 ReentrantLock 来实现呢？可以看到 tryAcquire 方法，它是不允许重入的，而 ReentrantLock 是允许重入的：

lock 方法一旦获取了独占锁，表示当前线程正在执行任务中；那么它会有以下几个作用

1. 如果正在执行任务，则不应该中断线程；
2. 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；

3. 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
4. 之所以设置为不可重入，是因为我们不希望任务在调用像 setCorePoolSize 这样的线程池控制方法时重新获取锁，这样会中断正在运行的线程
5.

```
    private final class Worker
        extends AbstractQueuedSynchronizer
        implements Runnable
    {
        private static final long serialVersionUID = 6138294804551838833L;

        /** Thread this worker is running in.  Null if factory fails. */
        final Thread thread; //注意了，这才是真正执行 task 的线程，从构造函数可知是由 ThreadFactury 创建的
        /** Initial task to run.  Possibly null. */
        Runnable firstTask; //这就是需要执行的 task
        /** Per-thread task counter */
        volatile long completedTasks; //完成的任务数，用于线程池统计

        Worker(Runnable firstTask) {
            setState(-1); //初始状态 -1,防止在调用 runWorker()，也就是真正执行 task 前中断 thread。
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this);
        }

        public void run() {
            runWorker(this);
        }

        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) {
            }
        }
    }
}
```

addWorkerFailed

addWorker 方法中，如果添加 Worker 并且启动线程失败，则会做失败后的处理。

这个方法主要做两件事

1. 如果 worker 已经构造好了，则从 workers 集合中移除这个 worker
2. 原子递减核心线程数（因为在 addWorker 方法中先做了原子增加）
3. 尝试结束线程池

```
private void addWorkerFailed(Worker w) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        if (w != null)
            workers.remove(w);
        decrementWorkerCount();
        tryTerminate();
    } finally {
        mainLock.unlock();
    }
}
```

runWorker 方法

前面已经了解了 ThreadPoolExecutor 的核心方法 addWorker，主要作用是增加工作线程，而 Worker 简单理解其实就是一个线程，里面重新了 run 方法，这块是线程池中执行任务的真正处理逻辑，也就是 runWorker 方法，这个方法主要做几件事

1. 如果 task 不为空,则开始执行 task
2. 如果 task 为空,则通过 getTask()再去取任务,并赋值给 task,如果取到的 Runnable 不为空,则执行该任务
3. 执行完毕后,通过 while 循环继续 getTask()取任务
4. 如果 getTask()取到的任务依然是空,那么整个 runWorker()方法执行完毕

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
```

unlock，表示当前 worker 线程允许中断，因为 new Worker 默认的 state=-1,此处是调用 Worker 类的 tryRelease()方法，将 state 置为 0，

而 interruptIfStarted()中只有 state>=0 才允许调用中断

```
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        //注意这个 while 循环,在这里实现了 [线程复用] // 如果 task 为空，则通过 getTask 来获取任务
        while (task != null || (task = getTask()) != null) {
            w.lock(); //上锁，不是为了防止并发执行任务，为了在 shutdown()时不终止正在运行的 worker
            线程池为 stop 状态时不接受新任务，不执行已经加入任务队列的任务，还中断正在执行的任务
            //所以对于 stop 状态以上是要中断线程的
            //(Thread.interrupted() &&runStateAtLeast(ctl.get(), STOP)确保线程中断标志位为 true 且是 stop 状态以上，接着清除了中断标志
            //!wt.isInterrupted()则再一次检查保证线程需要设置中断标志位
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);//这里默认是没有实现的，在一些特定的场景中我们可以自己继承 ThreadpoolExecutor 自己重写
                Throwable thrown = null;
                try {
                    task.run(); //执行任务中的 run 方法
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
```

```
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown); //这里默认默认而也是没有实现
                }
            } finally {
                //置空任务(这样下次循环开始时,task 依然为 null,需要再通过 getTask()
取) + 记录该 Worker 完成任务数量 + 解锁
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
        //1.将入参 worker 从数组 workers 里删除掉;
        //2.根据布尔值 allowCoreThreadTimeOut 来决定是否补充新的 Worker 进数组
workers
    }
}
```

getTask

worker 线程会从阻塞队列中获取需要执行的任务，这个方法不是简单的 take 数据，我们来分析下他的源码实现

你也许好奇是怎样判断线程有多久没有活动了，是不是以为线程池会启动一个监控线程，专门监控哪个线程正在偷懒？想太多，其实只是在线程从工作队列 poll 任务时，加上了超时限制，如果线程在 keepAliveTime 的时间内 poll 不到任务，那我就认为这条线程没事做，可以干掉了，看看这个代码片段你就清楚了

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?
    for (;;) {//自旋
        int c = ctl.get();
```

```
int rs = runStateOf(c);
```

* 对线程池状态的判断，两种情况会 workerCount-1，并且返回 null

1．线程池状态为 shutdown，且 workQueue 为空（反映了 shutdown 状态的线程池还是要执行 workQueue 中剩余的任务的）

2．线程池状态为 stop（shutdownNow()会导致变成 STOP）（此时不用考虑 workQueue 的情况）

```
// Check if queue empty only if necessary.
if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
    decrementWorkerCount();
    return null;//返回 null，则当前 worker 线程会退出
}

int wc = workerCountOf(c);
// timed 变量用于判断是否需要进行超时控制。
// allowCoreThreadTimeOut 默认是 false，也就是核心线程不允许进行超时；
// wc > corePoolSize，表示当前线程池中的线程数量大于核心线程数量；
// 对于超过核心线程数量的这些线程，需要进行超时控制
boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
```

1． 线程数量超过 maximumPoolSize 可能是线程池在运行时被调用了 setMaximumPoolSize()被改变了大小，否则已经 addWorker()成功不会超过 maximumPoolSize

2. timed && timedOut 如果为 true，表示当前操作需要进行超时控制，并且上次从阻塞队列中获取任务发生了超时.其实就是体现了空闲线程的存活时间

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

根据 timed 来判断，如果为 true，则通过阻塞队列 poll 方法进行超时控制，如果在 keepaliveTime 时间内没有获取到任务，则返回 null.

否则通过 take 方法阻塞式获取队列中的任务

```
    Runnable r = timed ?
```

```
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)//如果拿到的任务不为空，则直接返回给 worker 进行处理
                return r;
            timedOut = true;//如果 r==null，说明已经超时了，设置 timedOut=true，在
下次自旋的时候进行回收
        } catch (InterruptedException retry) {
            timedOut = false; // 如果获取任务时当前线程发生了中断，则设置 timedOut 为
false 并返回循环重试
        }
    }
```

这里重要的地方是第二个 if 判断，目的是控制线程池的有效线程数量。由上文中的分析可以知道，在执行 execute 方法时，如果当前线程池的线程数量超过了 corePoolSize 且小于 maximumPoolSize，并且 workQueue 已满时，则可以增加工作线程，但这时如果超时没有获取到任务，也就是 timedOut 为 true 的情况，说明 workQueue 已经为空了，也就说明了当前线程池中不需要那么多线程来执行任务了，可以把多于 corePoolSize 数量的线程销毁掉，保持线程数量在 corePoolSize 即可。

什么时候会销毁？当然是 runWorker 方法执行完之后，也就是 Worker 中的 run 方法执行完，由 JVM 自动回收。

getTask 方法返回 null 时，在 runWorker 方法中会跳出 while 循环，然后会执行 processWorkerExit 方法。

processWorkerExit

runWorker 的 while 循环执行完毕以后，在 finally 中会调用 processWorkerExit，来销毁工作线程。

到目前为止，我们已经从 execute 方法中输入了 worker 线程的创建到执行以及最后到销毁的全部过程。那么我们继续回到 execute 方法.我们只分析完 addWorker 这段逻辑，继续来看后面的判断

execute 后续逻辑分析

如果核心线程数已满，说明这个时候不能再创建核心线程了，于是走第二个判断

第二个判断逻辑比较简单，如果线程池处于运行状态并且任务队列没有满，则将任务添加到

队列中

第三个判断，核心线程数满了，队列也满了，那么这个时候创建新的线程也就是（非核心线程）

如果非核心线程数也达到了最大线程数大小，则直接拒绝任务

```
if (isRunning(c) && workQueue.offer(command)) {//2.核心池已满，但任务队列未满，添加到队列中
        int recheck = ctl.get();
//任务成功添加到队列以后，再次检查是否需要添加新的线程，因为已存在的线程可能被销毁了
        if (! isRunning(recheck) && remove(command))
            reject(command);//如果线程池处于非运行状态，并且把当前的任务从任务队列中移除成功，则拒绝该任务
        else if (workerCountOf(recheck) == 0)//如果之前的线程已被销毁完，新建一个线程
            addWorker(null, false);
    }
else if (!addWorker(command, false)) //3.核心池已满，队列已满，试着创建一个新线程
        reject(command);  //如果创建新线程失败了，说明线程池被关闭或者线程池完全满了，拒绝任务
```

拒绝策略

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

线程池的注意事项

分析完线程池以后，我们再来了解一下线程池的注意事项

阿里开发手册不建议使用线程池

不止一个同学问我说阿里开发手册上不建议使用线程池？估计这些同学都是没有认真看手册

的。手册上是说

线程池的构建不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式。

分析完原理以后，大家自己一定要有一个答案。我来简单分析下，用 Executors 使得用户不需要关心线程池的参数配置，意味着大家对于线程池的运行规则也会慢慢的忽略。这会导致一个问题，比如我们用 newFixdThreadPool 或者 singleThreadPool.允许的队列长度为 Integer.MAX_VALUE，如果使用不当会导致大量请求堆积到队列中导致 OOM 的风险

而 newCachedThreadPool，允许创建线程数量为 Integer.MAX_VALUE，也可能会导致大量线程的创建出现 CPU 使用过高或者 OOM 的问题

而如果我们通过 ThreadPoolExecutor 来构造线程池的话，我们势必要了解线程池构造中每个参数的具体含义，使得开发者在配置参数的时候能够更加谨慎。不至于像有些同学去面试的时候被问到：构造一个线程池需要哪些参数，都回答不上来

如何合理配置线程池的大小

如何合理配置线程池大小，也是很多同学反馈给我的问题，我也简单说一下。线程池大小不是靠猜，也不是说越多越好。

在遇到这类问题时，先冷静下来分析

1. 需要分析线程池执行的任务的特性： CPU 密集型还是 IO 密集型
2. 每个任务执行的平均时长大概是多少，这个任务的执行时长可能还跟任务处理逻辑是否涉及到网络传输以及底层系统资源依赖有关系

如果是 CPU 密集型，主要是执行计算任务，响应时间很快，cpu 一直在运行，这种任务 cpu 的利用率很高，那么线程数的配置应该根据 CPU 核心数来决定，CPU 核心数=最大同时执行线程数，加入 CPU 核心数为 4，那么服务器最多能同时执行 4 个线程。过多的线程会导致上下文切换反而使得效率降低。那线程池的最大线程数可以配置为 cpu 核心数+1

如果是 IO 密集型，主要是进行 IO 操作，执行 IO 操作的时间较长，这是 cpu 出于空闲状态，导致 cpu 的利用率不高，这种情况下可以增加线程池的大小。这种情况下可以结合线程的等待时长来做判断，等待时间越高，那么线程数也相对越多。一般可以配置 cpu 核心数的 2 倍。

一个公式：线程池设定最佳线程数目 = ((线程池设定的线程等待时间+线程 CPU 时间）/线程 CPU 时间 ）* CPU 数目

这个公式的线程 cpu 时间是预估的程序单个线程在 cpu 上运行的时间(通常使用 loadrunner 测试大量运行次数求出平均值)

线程池中的线程初始化

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：prestartCoreThread()：初始化一个核心线程； prestartAllCoreThreads()：初始化所有核心线程

```
ThreadPoolExecutor tpe=(ThreadPoolExecutor)service;
tpe.prestartAllCoreThreads();
```

线程池的关闭

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown()和 shutdownNow()，其中：shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务 shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

线程池容量的动态调整

ThreadPoolExecutor 提供了动态调整线程池容量大小的方法： setCorePoolSize()和 setMaximumPoolSize()，setCorePoolSize：设置核心池大小 setMaximumPoolSize：设置线程池最大能创建的线程数目大小

任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列，即 workQueue，它用来存放等待执行的任务。workQueue 的类型为 BlockingQueue，通常可以取下面三种类型：

1. ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；
2. LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为 Integer.MAX_VALUE；
3. SynchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

线程池的监控

如果在项目中大规模的使用了线程池，那么必须要有一套监控体系，来指导当前线程池的状态，当出现问题的时候可以快速定位到问题。而线程池提供了相应的扩展方法，我们通过重写线程池的 beforeExecute、afterExecute 和 shutdown 等方式就可以实现对线程的监控，简单给大家演示一个案例

```
public class Demo1 extends ThreadPoolExecutor {

    // 保存任务开始执行的时间,当任务结束时,用任务结束时间减去开始时间计算任务执行时间
    private ConcurrentHashMap<String,Date> startTimes;

    public Demo1(int corePoolSize, int maximumPoolSize, long
keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue);
        this.startTimes=new ConcurrentHashMap<>();
    }

    @Override
    public void shutdown() {
        System.out.println("已经执行的任务数:
"+this.getCompletedTaskCount()+"," +
                "当前活动线程数:"+this.getActiveCount()+",当前排队线程
数:"+this.getQueue().size());
        System.out.println();
        super.shutdown();
    }

    //任务开始之前记录任务开始时间
    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        startTimes.put(String.valueOf(r.hashCode()),new Date());
        super.beforeExecute(t, r);
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
        Date startDate = startTimes.remove(String.valueOf(r.hashCode()));
        Date finishDate = new Date();
```

```
        long diff = finishDate.getTime() - startDate.getTime();
        // 统计任务耗时、初始线程数、核心线程数、正在执行的任务数量、
        // 已完成任务数量、任务总数、队列里缓存的任务数量、
        // 池中存在的最大线程数、最大允许的线程数、线程空闲时间、线程池是否关闭、线程池
是否终止
        System.out.print("任务耗时:"+diff+"\n");
        System.out.print("初始线程数:"+this.getPoolSize()+"\n");
        System.out.print("核心线程数:"+this.getCorePoolSize()+"\n");
        System.out.print("正在执行的任务数量:"+this.getActiveCount()+"\n");
        System.out.print("已经执行的任务
数:"+this.getCompletedTaskCount()+"\n");
        System.out.print("任务总数:"+this.getTaskCount()+"\n");
        System.out.print("最大允许的线程数:"+this.getMaximumPoolSize()+"\n");
        System.out.print("线程空闲时
间:"+this.getKeepAliveTime(TimeUnit.MILLISECONDS)+"\n");
        System.out.println();
        super.afterExecute(r, t);
    }

    public static ExecutorService newCachedThreadPool() {
        return new Demo1(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new
SynchronousQueue ());
    }
}
```

测试脚本

```
public class Test implements Runnable{

    private static ExecutorService es =Demo1.newCachedThreadPool();
    @Override
    public void run() {
        try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) throws Exception {
        for (int i = 0; i < 100; i++) {
            es.execute(new Test());
        }
        es.shutdown();
    }
```

Callable/Future 使用及原理分析

很多同学应该关注到了。线程池的执行任务有两种方法，一种是 submit、一种是 execute；这两个方法是有区别的，那么基于这个区别我们再来看看。

execute 和 submit 区别

1. execute 只可以接收一个 Runnable 的参数
2. execute 如果出现异常会抛出
3. execute 没有返回值

1. submit 可以接收 Runable 和 Callable 这两种类型的参数，
2. 对于 submit 方法，如果传入一个 Callable，可以得到一个 Future 的返回值
3. submit 方法调用不会抛异常，除非调用 Future.get

这里，我们重点了解一下 Callable/Future，可能很多同学知道他是一个带返回值的线程，但是具体的实现可能不清楚。

Callable/Future 案例演示

Callable/Future 和 Thread 之类的线程构建最大的区别在于，能够很方便的获取线程执行完以后的结果。首先来看一个简单的例子

```
public class CallableDemo implements Callable<String> {
```

```
    @Override
    public String call() throws Exception {
        //Thread.sleep(3000);//阻塞案例演示
        return "hello world";
    }
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        CallableDemo callableDemo=new CallableDemo();
        FutureTask futureTask=new FutureTask(callableDemo);
        new Thread(futureTask).start();
        System.out.println(futureTask.get());
    }
}
```

想一想我们为什么需要使用回调呢？那是因为结果值是由另一线程计算的，当前线程是不知道结果值什么时候计算完成，所以它传递一个回调接口给计算线程，当计算完成时，调用这个回调接口，回传结果值。

这个在很多地方有用到，比如 Dubbo 的异步调用，比如消息中间件的异步通信等等...

利用 FutureTask、Callable、Thread 对耗时任务（如查询数据库）做预处理，在需要计算结果之前就启动计算。

所以我们来看一下 Future/Callable 是如何实现的

Callable/Future 原理分析

在刚刚实现的 demo 中，我们用到了两个 api，分别是 Callable 和 FutureTask。

Callable 是一个函数式接口，里面就只有一个 call 方法。子类可以重写这个方法，并且这个方法会有一个返回值

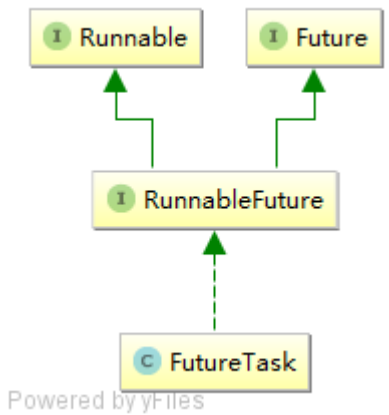
```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
```

```
    V call() throws Exception;
}
```

FutureTask

FutureTask 的类关系图如下，它实现 RunnableFuture 接口，那么这个 RunnableFuture 接口的作用是什么呢。

在讲解 FutureTask 之前，先看看 Callable, Future, FutureTask 它们之间的关系图，如下：




```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

RunnableFuture 是一个接口，它继承了 Runnable 和 Future 这两个接口，Runnable 太熟悉了，那么 Future 是什么呢?

Future 表示一个任务的生命周期，并提供了相应的方法来判断是否已经完成或取消，以及获取任务的结果和取消任务等。

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    // 当前的 Future 是否被取消，返回 true 表示已取消
    boolean isCancelled();
    // 当前 Future 是否已结束。包括运行完成、抛出异常以及取消，都表示当前 Future 已结束
    boolean isDone();
     // 获取 Future 的结果值。如果当前 Future 还没有结束，那么当前线程就等待，
    // 直到 Future 运行结束，那么会唤醒等待结果值的线程的。
    V get() throws InterruptedException, ExecutionException;
```

```
    // 获取 Future 的结果值。与 get() 相比较多了允许设置超时时间
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

分析到这里我们其实有一些初步的头绪了，FutureTask 是 Runnable 和 Future 的结合，如果我们把 Runnable 比作是生产者，Future 比作是消费者，那么 FutureTask 是被这两者共享的，生产者运行 run 方法计算结果，消费者通过 get 方法获取结果。

作为生产者消费者模式，有一个很重要的机制，就是如果生产者数据还没准备的时候，消费者会被阻塞。当生产者数据准备好了以后会唤醒消费者继续执行。

这个有点像我们上次可分析的阻塞队列，那么在 FutureTask 里面是基于什么方式实现的呢?

state 的含义

表示 FutureTask 当前的状态，分为七种状态

```
private static final int NEW          = 0; // NEW 新建状态，表示这个 FutureTask 还没有开始运行
// COMPLETING 完成状态， 表示 FutureTask 任务已经计算完毕了
// 但是还有一些后续操作，例如唤醒等待线程操作，还没有完成。
private static final int COMPLETING   = 1;
// FutureTask 任务完结，正常完成，没有发生异常
private static final int NORMAL       = 2;
// FutureTask 任务完结，因为发生异常。
private static final int EXCEPTIONAL  = 3;
// FutureTask 任务完结，因为取消任务
private static final int CANCELLED    = 4;
// FutureTask 任务完结，也是取消任务，不过发起了中断运行任务线程的中断请求
private static final int INTERRUPTING = 5;
// FutureTask 任务完结，也是取消任务，已经完成了中断运行任务线程的中断请求
private static final int INTERRUPTED  = 6;
```

run 方法

```
public void run() {
// 如果状态 state 不是 NEW，或者设置 runner 值失败
```

```
// 表示有别的线程在此之前调用 run 方法，并成功设置了 runner 值
// 保证了只有一个线程可以运行 try 代码块中的代码。
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {/ 只有 c 不为 null 且状态 state 为 NEW 的情
况
            V result;
            boolean ran;
            try {
                result = c.call(); //调用 callable 的 call 方法，并获得返回结果
                ran = true;//运行成功
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex); //设置异常结果，
            }
            if (ran)
                set(result);//设置结果
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
```

}

其实 run 方法作用非常简单，就是调用 callable 的 call 方法返回结果值 result，根据是否发生异常，调用 set(result)或 setException(ex)方法表示 FutureTask 任务完结。

不过因为 FutureTask 任务都是在多线程环境中使用，所以要注意并发冲突问题。注意在 run 方法中，我们没有使用 synchronized 代码块或者 Lock 来解决并发问题，而是使用了 CAS 这个乐观锁来实现并发安全，保证只有一个线程能运行 FutureTask 任务

get 方法

get 方法就是阻塞获取线程执行结果，这里主要做了两个事情

1. 判断当前的状态，如果状态小于等于 COMPLETING，表示 FutureTask 任务还没有完结，所以调用 awaitDone 方法，让当前线程等待。
2. report 返回结果值或者抛出异常

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);
    return report(s);
}
```

awaitDone

如果当前的结果还没有被执行完，把当前线程线程和插入到等待队列

```
private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    final long deadline = timed ? System.nanoTime() +
nanos : 0L;
    WaitNode q = null;
    boolean queued = false; // 节点是否已添加
    for (;;) {
        // 如果当前线程中断标志位是 true，
        // 那么从列表中移除节点 q，并抛出 InterruptedException 异
常
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
```

```
            }

            int s = state;
            if (s > COMPLETING) { // 当状态大于 COMPLETING 时，表
示 FutureTask 任务已结束。
                if (q != null)
                    q.thread = null;   // 将节点 q 线程设置为 null，
因为线程没有阻塞等待
                return s;
            }// 表示还有一些后序操作没有完成，那么当前线程让出执行权
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            //表示状态是 NEW，那么就需要将当前线程阻塞等待。
            // 就是将它插入等待线程链表中，
            else if (q == null)
                q = new WaitNode();
            else if (!queued)
              // 使用 CAS 函数将新节点添加到链表中，如果添加失败，那么
queued 为 false，
              // 下次循环时，会继续添加，知道成功。
                queued = UNSAFE.compareAndSwapObject(this,
waitersOffset,
                                                     q.next =
waiters, q);
            else if (timed) {// timed 为 true 表示需要设置超时
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos); // 让当前线
程等待 nanos 时间
            }
```

```
            else
                LockSupport.park(this);
        }
    }
```

被阻塞的线程，会等到 run 方法执行结束之后被唤醒

report

report 方法就是根据传入的状态值 s，来决定是抛出异常，还是返回结果值。这个两种情况都表示 FutureTask 完结了

```
private V report(int s) throws ExecutionException {
    Object x = outcome;//表示 call 的返回值
    if (s == NORMAL) // 表示正常完结状态，所以返回结果值
        return (V)x;
// 大于或等于 CANCELLED，都表示手动取消 FutureTask 任务，
// 所以抛出 CancellationException 异常
    if (s >= CANCELLED)
        throw new CancellationException();
// 否则就是运行过程中，发生了异常，这里就抛出这个异常
    throw new ExecutionException((Throwable)x);
}
```

线程池对于 Future/Callable 的执行

我们现在再来看线程池里面的 submit 方法，就会很清楚了。

```
public class CallableDemo implements Callable<String> {

    @Override
    public String call() throws Exception {
        //Thread.sleep(3000);//阻塞案例演示
        return "hello world";
    }

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService es=Executors.newFixedThreadPool(1);
```

```
        CallableDemo callableDemo=new CallableDemo();
        Future future=es.submit(callableDemo);
        System.out.println(future.get());
    }
}
```

AbstractExecutorService.submit

调用抽象类中的 submit 方法，这里其实相对于 execute 方法来说，只多做了一步操作，就是封装了一个 RunnableFuture

```
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

ThreadpoolExecutor.execute

然后调用 execute 方法，这里面的逻辑前面分析过了，会通过 worker 线程来调用过 ftask 的 run 方法。而这个 ftask 其实就是 FutureTask 里面最终实现的逻辑