线程池与阻塞队列

一、为什么要使用线程池

在Java语言中，虽然创建并启动一个线程非常方便，但是由于创建线程需要占用一定的操作系统资源，在高并发的情况下，频繁的创建和销毁线程会大量消耗CPU和内存资源，对程序性能造成很大的影响。为了避免这一问题，Java给我们提供了线程池。

线程池是一种基于池化技术思想来管理线程的工具。在线程池中维护了多个线程，由线程池统一的管理调配线程来执行任务。通过线程复用，减少了频繁创建和销毁线程的开销。

因为线程池中用到了阻塞队列，所以在讲线程池之前，我们先介绍一下阻塞队列的相关知识。

二、阻塞队列 (BlockingQueue)

1. 阻塞队列简介

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

1.1 常见阻塞场景

阻塞队列有两个常见的阻塞场景，分别是：

（1）当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。

（2）当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

支持以上两种阻塞场景的队列被称为阻塞队列。

1.2 BlockingQueue的核心方法

放入数据：

• **offer（anObject）**：表示如果可能的话，将anObject加到BlockingQueue里。即如果BlockingQueue可以容纳，则返回true，否则返回false。（本方法不阻塞当前执行方法的线程）

• **offer（E o，long timeout，TimeUnit unit）**：可以设定等待的时间。如果在指定的时间内还不能往队列中加入BlockingQueue，则返回失败。

• **put（anObject）**：将anObject加到BlockingQueue里。如果BlockQueue没有空间，则调用此方法的线程被阻断，直到BlockingQueue里面有空间再继续。

获取数据：

• **poll（）**： 取走 BlockingQueue 里排在首位的对象。若BlockingQueue为空，直接返回null。

• **poll（long timeout，TimeUnit unit）**：从BlockingQueue中取出一个队首的对象。若队列为空，则阻塞等待指定时间，如果在指定时间内， 队列一旦有数据可取，则立即返回队列中的数据；否则直到时间超时还没有数据可取，返回null。

• **take（）**：取走BlockingQueue里排在首位的对象。若BlockingQueue为空，则阻断进入等待状态，直到BlockingQueue有新的数据被加入。

• **drainTo（）**：一次性从BlockingQueue获取所有可用的数据对象（还可以指定获取数据的个数）。通 过该方法，可以提升获取数据的效率；无须多次分批加锁或释放锁。

2. Java中的阻塞队列

在Java中提供了7个阻塞队列，它们分别如下所示。

• ArrayBlockingQueue：由数组结构组成的有界阻塞队列。

• LinkedBlockingQueue：由链表结构组成的有界阻塞队列。

• PriorityBlockingQueue：支持优先级排序的无界阻塞队列。

• DelayQueue：使用优先级队列实现的无界阻塞队列。

• SynchronousQueue：不存储元素的阻塞队列。

• LinkedTransferQueue：由链表结构组成的无界阻塞队列。

• LinkedBlockingDeque：由链表结构组成的双向阻塞队列。

2.1 ArrayBlockingQueue

它是用数组实现的有界阻塞队列，并按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证线程公平地访问队列。公平访问队列就是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列。即先阻塞的生产者线程，可以先往队列里插入元素；先阻塞的消费者线程，可以先从队列里获取元素。我们可以使用以下代码创建一 个公平的阻塞队列，如下所示：

ArrayBlockingQueue fairQueue=new ArrayBlockingQueue(2000,true);

2.2 LinkedBlockingQueue

它是基于链表的阻塞队列，同ArrayListBlockingQueue类似，此队列按照先进先出（FIFO）的原则对元素进行排序，内部也维持着一个数据缓冲队列（该队列由一个链表构成）。当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到缓存容量的最大值时（LinkedBlockingQueue可以通过构造方法指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒。反之，对于消费者这端的处理也基于同样的原理。 而LinkedBlockingQueue之所以能够高效地处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步。这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。作为开发者，我们需要注意的是，如果构造一个LinkedBlockingQueue对象，而没有指定其容量大小，LinkedBlockingQueue会默认一个类似无限大小的容量（Integer.MAX_VALUE）。这样一来，如果生产者的速度一旦大于消费者的速度，也

许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。ArrayBlockingQueue和LinkedBlockingQueue是两个最普通也是最常用的阻塞队列。一般情况下，在处理多线程间的生产者-消费者问题时，使用这两个类足已。

2.3 PriorityBlockingQueue

它是一个支持优先级的无界队列。默认情况下元素采取自然顺序升序排列。这里可以自定义实现compareTo（）方法来指定元素进行排序规则；或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。但其不能保证同优先级元素的顺序。

2.4 DelayQueue

它是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed 接口。创建元素时，可以指定元素到期的时间，只有在元素到期时才能从队列中取走。

2.5 SynchronousQueue

它是一个不存储元素的阻塞队列。每个插入操作必须等待另一个线程的移除操作，同样任何一个移除操作都等待另一个线程的插入操作。因此此队列内部其实没有任何一个元素，或者说容量是0，严格来说它并不是一种容器。由于队列没有容量，因此不能调用peek操作（返回队列的头元素）。

2.6 LinkedTransferQueue

它是一个由链表结构组成的无界阻塞TransferQueue队列。LinkedTransferQueue实现了一个重要的接口TransferQueue。该接口含有5个方法，其中有3个重要的方法，它们分别如下所示。

（1）transfer（E e）：若当前存在一个正在等待获取的消费者线程，则立刻将元素传递给消费者；如果没有消费者在等待接收数据，就会将元素插入到队列尾部，并且等待进入阻塞状态，直到有消费者线程取走该元素。

（2）tryTransfer（E e）：若当前存在一个正在等待获取的消费者线程，则立刻将元素传递给消费者；若不存在，则返回 false，并且不进入队列，这是一个不阻塞的操作。与 transfer 方法不同的是，tryTransfer方法无论消费者是否接收，其都会立即返回；而transfer方法则是消费者接收了才返回。

（3）tryTransfer（E e，long timeout，TimeUnit unit）：若当前存在一个正在等待获取的消费者线程，则立刻将元素传递给消费者；若不存在则将元素插入到队列尾部，并且等待消费者线程取走该元素。若在指定的超时时间内元素未被消费者线程获取，则返回false；若在指定的超时时间内其被消费者线程获取， 则返回true。

2.7 LinkedBlockingDeque

它是一个由链表结构组成的双向阻塞队列。双向队列可以从队列的两端插入和移出元素，因此在多线程同时入队时，也就减少了一半的竞争。由于是双向的，因此LinkedBlockingDeque多了addFirst、addLast、 offerFirst、offerLast、peekFirst、peekLast等方法。其中，以First单词结尾的方法，表示插入、获取或移除双端队列的第一个元素；以Last单词结尾的方法，表示插入、获取或移除双端队列的最后一个元素。

3. 阻塞队列实现原理

以ArrayBlockingQueue为例，我们先来看看代码，如下所示：

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
        implements BlockingQueue<E>, java.io.Serializable {

    /** The queued items */
    final Object[] items;

    /** items index for next take, poll, peek or remove */
    int takeIndex;

    /** items index for next put, offer, or add */
    int putIndex;

    /** Number of elements in the queue */
    int count;

    /** Main lock guarding all access */
    final ReentrantLock lock;

    /** Condition for waiting takes */
    private final Condition notEmpty;

    /** Condition for waiting puts */
    private final Condition notFull;
```

ArrayBlockingQueue 是维护一个 Object 类型的数组，takeIndex 和putIndex分别 表示队首元素和队尾元素的下标，count表示队列中元素的个数，lock则是一个可重入锁，notEmpty和notFull 是等待条件。

接下来我们看看关键方法put，代码如下所示：

```
    public void put(E e) throws InterruptedException {
        Objects.requireNonNull(e);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length) //说明此时队列是满的
                notFull.await();
            enqueue(e);
        } finally {
            lock.unlock();
        }
    }
```

从 put 方法的实现可以看出，它先获取了锁，并且获取的是可中断锁，然后判断当前元素个数是否等于数组的长度，如果相等，则调用notFull.await（）进行等待。当此线程被其他线程唤醒时，通过 enqueue（e）方法插入元素，最后解锁。

接下来看enqueue（e）方法，如下所示：

```
    private void enqueue(E x) {
        // assert lock.getHoldCount() == 1;
        // assert items[putIndex] == null;
        final Object[] items = this.items;
        items[putIndex] = x;
        if (++putIndex == items.length) putIndex = 0;
        count++;
        notEmpty.signal();
    }
```

插入成功后，通过notEmpty唤醒正在等待取元素的线程。再来看看take方法。

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
```

跟put方法实现类似，put方法等待的是notFull信号，而take方法等待的是notEmpty信号。在take方法中， 如果可以取元素，则通过dequeue方法取得元素。下面是dequeue方法的实现：

```
    private E dequeue() {
        // assert lock.getHoldCount() == 1;
        // assert items[takeIndex] != null;
        final Object[] items = this.items;
        @SuppressWarnings("unchecked")
        E x = (E) items[takeIndex];
        items[takeIndex] = null;
        if (++takeIndex == items.length) takeIndex = 0;
        count--;
        if (itrs != null)
            itrs.elementDequeued();
        notFull.signal();
        return x;
    }
```

跟enqueue方法类似，在获取元素后，通过notFull的signal方法来唤醒正在等待插入元素的线程。

三、线程池

1. 线程池使用和参数讲解

1.1 线程池的使用

```
// 实例化一个线程池
ThreadPoolExecutor executor = new ThreadPoolExecutor(3, 10, 60,
        TimeUnit.SECONDS, new ArrayBlockingQueue<>(20));
// 使用线程池执行一个任务
executor.execute(() -> {
    // Do something
});
// 关闭线程池,会阻止新任务提交，但不影响已提交的任务
executor.shutdown();
// 关闭线程池，阻止新任务提交，并且中断当前正在运行的线程
executor.showdownNow();
```

1.2 线程池的参数讲解

```
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.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

这些参数的作用如下所示。

• **corePoolSize**：核心线程数。默认情况下线程池是空的，只有任务提交时才会创建线程。如果当前运行的线程数少corePoolSize，则创建新线程来处理任务；如果等于或者多于corePoolSize，则不再创建。如果调用线程池的prestartAllcoreThread方法，线程池会提前创建并启动所有的核心线程来等待任务。

• **maximumPoolSize**：线程池允许创建的最大线程数。如果任务队列满了并且线程数小于 maximumPoolSize 时，则线程池仍旧会创建新的线程来处理任务。 （最大线程数 **不包含** 任务队列中的数量）

• **keepAliveTime**：非核心线程闲置的超时时间。超过这个时间则回收。如果任务很多，并且每个任务 的执行事件很短，则可以调大keepAliveTime来提高线程的利用率。另外，如果设置 allowCoreThreadTimeOut=true时，keepAliveTime也会应用到核心线程上，

• **TimeUnit**：keepAliveTime参数的时间单位。可选的单位有天（DAYS）、小时（HOURS）、分钟（MINUTES）、秒（SECONDS）、毫秒（MILLISECONDS）等。

• **workQueue**：任务队列。如果当前线程数大于corePoolSize，则将任务添加到此任务队列中。该任务队列是BlockingQueue类型的，也就是阻塞队列。

• **ThreadFactory**：线程工厂。可以用线程工厂给每个创建出来的线程设置名字。一般情况下无须设置该参数。

• **RejectedExecutionHandler**：饱和策略。这是当任务队列和线程池都满了时所采取的应对策略，默认 是AbordPolicy，表示无法处理新任务，并抛出RejectedExecutionException异常。此外还有3种策略，它们分 别如下：

（1）CallerRunsPolicy：当提交任务到线程池中被拒绝时，会在线程池当前正在运行的Thread线程中处理被拒绝额任务。即哪个线程提交的任务哪个线程去执行。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code CallerRunsPolicy}.
     */
    public CallerRunsPolicy() { }

    /**
     * Executes task r in the caller's thread, unless the executor
     * has been shut down, in which case the task is discarded.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}
```

（2）DiscardPolicy：当提交任务到线程池中被拒绝时，线程池会丢弃这个被拒绝的任务

```
/**
 * A handler for rejected tasks that silently discards the
 * rejected task.
```

```
 */
public static class DiscardPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code DiscardPolicy}.
     */
    public DiscardPolicy() { }

    /**
     * Does nothing, which has the effect of discarding task r.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    }
}
```

（3）DiscardOldestPolicy：丢弃队列队首的任务，并执行当前的任务。

```
public static class DiscardOldestPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code DiscardOldestPolicy} for the given executor.
     */
    public DiscardOldestPolicy() { }

    /**
     * Obtains and ignores the next task that the executor
     * would otherwise execute, if one is immediately available,
     * and then retries execution of task r, unless the executor
     * is shut down, in which case task r is instead discarded.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
}
```

1.3 Java提供的线程池

ThreadPoolExecutor的构造方法中参数众多，对于初学者而言在没有了解各个参数的作用的情况下很难去配置合适的线程池。因此Java还为我们提供了一个线程池工具类Executors来快捷的创建线程池。

```
// 实例化一个单线程的线程池
ExecutorService singleExecutor = Executors.newSingleThreadExecutor();
// 创建固定线程个数的线程池
ExecutorService fixedExecutor = Executors.newFixedThreadPool(10);
// 创建一个可重用固定线程数的线程池
ExecutorService executorService2 = Executors.newCachedThreadPool();
```

但是，通常来说在实际开发中并不推荐直接使用Executors来创建线程池，而是需要根据项目实际情况配置适合自己项目的线程池，关于如何配置合适的线程池，需要我们理解线程池的各个参数以及线程池的工作原理之后才能有答案。

1.4 线程池生命周期

线程池从诞生到死亡，中间会经历RUNNING、SHUTDOWN、STOP、TIDYING、TERMINATED五个生命周期状态。

RUNNING 表示线程池处于运行状态，能够接受新提交的任务且能对已添加的任务进行处理。RUNNING状态是线程池的初始化状态，线程池一旦被创建就处于RUNNING状态。

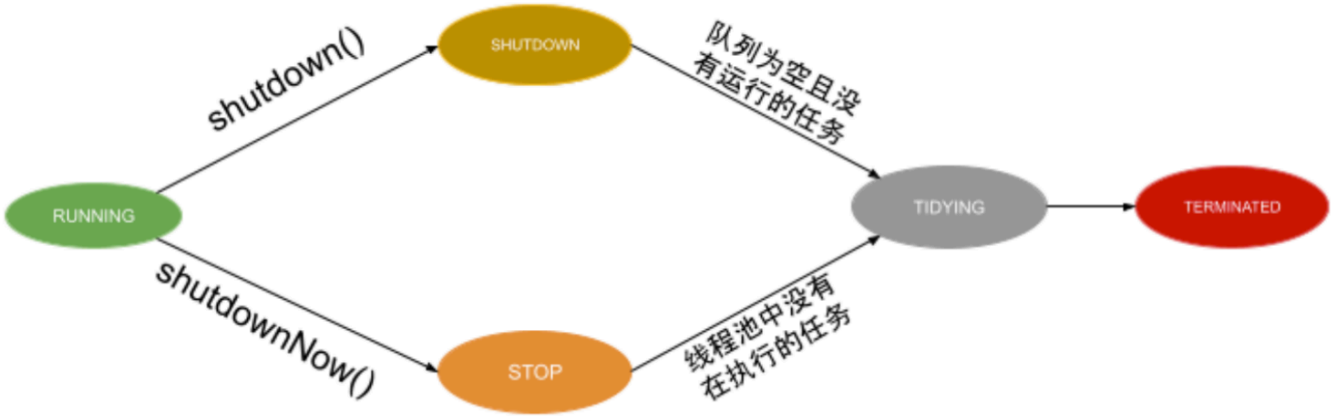
SHUTDOWN 线程处于关闭状态，不接受新任务，但可以处理已添加的任务。RUNNING状态的线程池调用shutdown后会进入SHUTDOWN状态。

STOP 线程池处于停止状态，不接收任务，不处理已添加的任务，且会中断正在执行任务的线程。RUNNING状态的线程池调用了shutdownNow后会进入STOP状态。

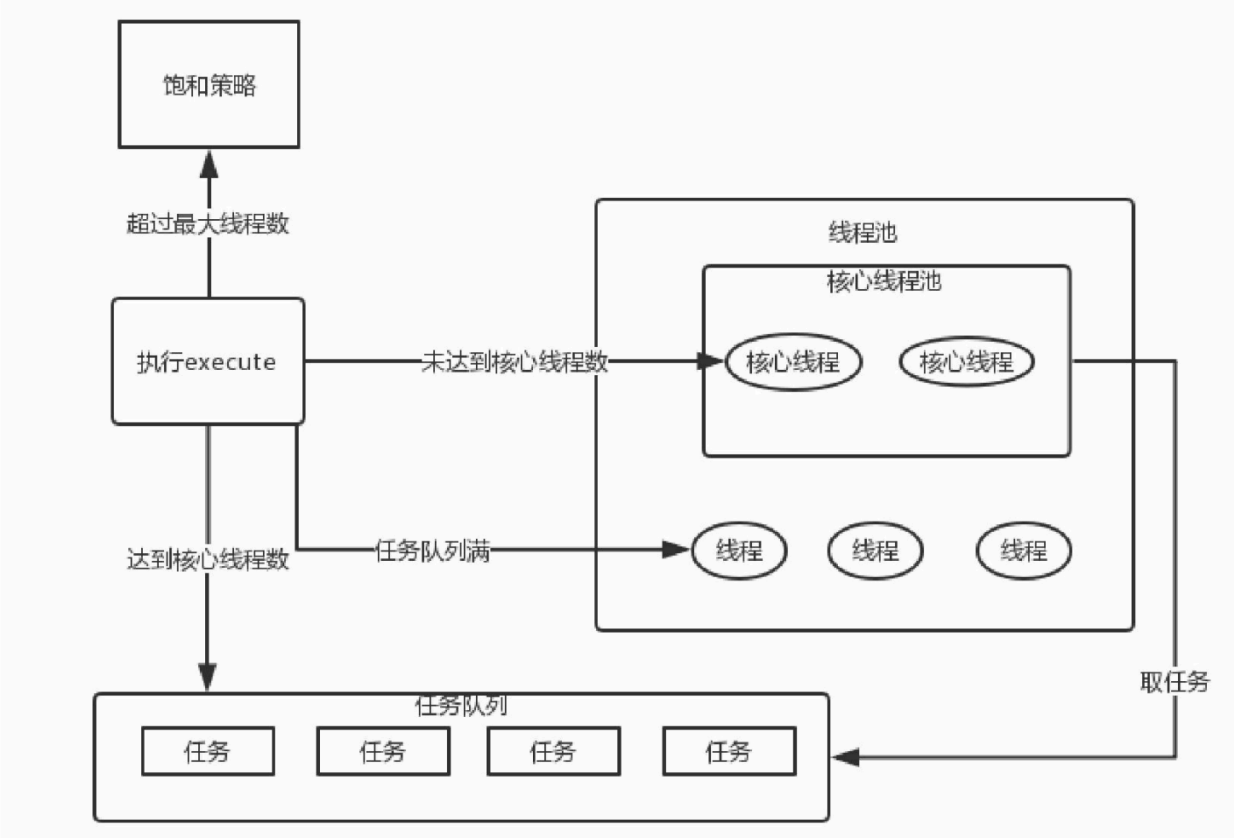
TIDYING 当所有任务已终止，且任务数量为0时，线程池会进入TIDYING。当线程池处于SHUTDOWN状态时，阻塞队列中的任务被执行完了，且线程池中没有正在执行的任务了，状态会由SHUTDOWN变为TIDYING。当线程处于STOP状态时，线程池中没有正在执行的任务时则会由STOP变为TIDYING。

TERMINATED 线程终止状态。处于TIDYING状态的线程执行terminated()后进入TERMINATED状态。

根据上述线程池生命周期状态的描述，可以画出如下所示的线程池生命周期状态流程示意图



2. 线程池处理流程



从图中可以看到，如果我们执行ThreadPoolExecutor的execute方法，会遇到各种情况：

（1）如果线程池中的线程数未达到核心线程数，则创建核心线程处理任务。

（2）如果线程数大于或者等于核心线程数，则将任务加入任务队列，核心线程池中的空闲线程会不断地从任务队列中取出任务进行处理。

（3）如果任务队列满了，并且线程数没有达到最大线程数，则创建非核心线程去处理任务。

（4）如果线程数超过了最大线程数，则执行饱和策略。

3. 线程池源码分析

从上面对线程池工作流程解读来看，线程池的原理似乎并没有很难。但是想要读懂线程池的源码并不容，主要原因是线程池内部运用到了大量并发相关知识，另外还与线程池中用到的位运算有关。

3.1 线程池中的位运算

在向线程池提交任务时有两个比较中要的参数会决定任务的去向，这两个参数分别是线程池的状态和线程池中的线程数。在ThreadPoolExecutor内部使用了一个AtomicInteger类型的整数ctl来表示这两个参数，代码如下：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    // Integer.SIZE = 32.所以 COUNT_BITS= 29
    private static final int COUNT_BITS = Integer.SIZE - 3;
    // 00001111 11111111 11111111 11111111 这个值可以表示线程池的最大线程容量
    private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;
    // 将-1左移29位得到RUNNING状态的值
    private static final int RUNNING    = -1 << COUNT_BITS; // 11100000 00000000
00000000 00000000
    private static final int SHUTDOWN   =  0 << COUNT_BITS; // 00000000 00000000
00000000 00000000
    private static final int STOP       =  1 << COUNT_BITS; // 00100000 00000000
00000000 00000000
    private static final int TIDYING    =  2 << COUNT_BITS; // 01000000 00000000
00000000 00000000
    private static final int TERMINATED =  3 << COUNT_BITS; // 01100000 00000000
00000000 00000000
    // 线程池运行状态和线程数
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

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

    // ...
}
```

因为涉及多线程的操作，这里为了保证原子性，ctl参数使用了AtomicInteger类型，并且通过ctlOf方法来计算出了ctl的初始值。如果不了解位运算大概很难理解上述代码的用意。

我们知道，int类型在Java中占用4byte的内存,一个byte占用8bit,所以Java中的int类型共占用32bit。对于这个32bit，我们可以进行高低位的拆分。做Android开发的同学应该都了解View测量流程中的MeasureSpec参数，这个参数将32bit的int拆分成了高2位和低30位，分别表示View的测量模式和测量值。而这里的ctl与MeasureSpec类似，ctl将32位的int拆分成了高3位和低29位，分别表示线程池的运行状态和线程池中的线程个数。

下面我们通过位运算来验证一下ctl是如何工作的

可以看到上述代码中RUNNING的值为-1左移29位，我们知道在计算机中**负数是以其绝对值的补码来表示的，而补码是由反码加1得到。**因此-1在计算机中存储形式为1的反码+1。

（正数的反码就是各位取反，补码就是他本身；负数的反码也是各位取反，补码等于它的绝对值反码 + 1）

```
1的原码：00000000 00000000 00000000 00000001
                                              +
1的反码：11111111 11111111 11111111 11111110
       ---------------------------------------
-1存储：  11111111 11111111 11111111 11111111
```

接下来对-1左移29位可以得到RUNNING的值为：

```
// 高三位表示线程状态，即高三位为111表示RUNNING
11100000 00000000 00000000 00000000
```

而AtomicInteger初始线程数量是0，因此ctlOf方法中的“|”运算如下：

```
RUNNING：   11100000 00000000 00000000 00000000
                                                |
线程数为0：  00000000 00000000 00000000 00000000
          ---------------------------------------
得到ctl：    11100000 00000000 00000000 00000000
```

通过RUNNING|0(线程数)即可得到ctl的初始值。同时还可以通过以下方法将ctl拆解成运行状态和线程数：

```
    // 00001111 11111111 11111111 11111111
    private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;
    // 获取线程池运行状态
    private static int runStateOf(int c)     { return c & ~COUNT_MASK; }
    // 获取线程池中的线程数
    private static int workerCountOf(int c)  { return c & COUNT_MASK; }
```

假设此时线程池为RUNNING状态，且线程数为0，验证一下runStateOf是如何得到线程池的运行状态的：

```
COUNT_MASK:  00001111 11111111 11111111 11111111

~COUNT_MASK: 11110000 00000000 00000000 00000000
                                                &
ctl:         11100000 00000000 00000000 00000000
            ----------------------------------------
RUNNING:     11100000 00000000 00000000 00000000
```

如果不理解上边的验证流程没有关系，只要知道通过runStateOf方法可以得到线程池的运行状态，通过workerCountOf可以得到线程池中的线程数即可。

接下来我们进入线程池的源码的源码分析环节。

3.2 ThreadPoolExecutor的execute

向线程池提交任务的方法是execute方法，execute方法是ThreadPoolExecutor的核心方法，以此方法为入口来进行剖析，execute方法的代码如下：

```
    public void execute(Runnable command) {
         if (command == null)
             throw new NullPointerException();
         // 获取ctl的值
         int c = ctl.get();
         // 1.线程数小于corePoolSize
         if (workerCountOf(c) < corePoolSize) {
             // 线程池中线程数小于核心线程数，则尝试创建核心线程执行任务
             if (addWorker(command, true))
                 return;
             c = ctl.get();
         }
         // 2.到此处说明线程池中线程数大于核心线程数或者创建线程失败
         if (isRunning(c) && workQueue.offer(command)) {
             // 如果线程是运行状态并且可以使用offer将任务加入阻塞队列未满，offer是非阻塞操作。
             int recheck = ctl.get();
             // 重新检查线程池状态，因为上次检测后线程池状态可能发生改变，如果非运行状态就移除任务并执行拒绝策略
             if (! isRunning(recheck) && remove(command))
                 reject(command);
             // 如果是运行状态，并且线程数是0，则创建线程
             else if (workerCountOf(recheck) == 0)
                 // 线程数是0，则创建非核心线程，且不指定首次执行任务，这里的第二个参数其实没有实际意义
                 addWorker(null, false);
```

```
        }
        // 3.阻塞队列已满，创建非核心线程执行任务
        else if (!addWorker(command, false))
            // 如果失败，则执行拒绝策略
            reject(command);
    }
```

execute方法中的逻辑可以分为三部分：

- 1.如果线程池中的线程数小于核心线程，则直接调用addWorker方法创建新线程来执行任务。
- 2.如果线程池中的线程数大于核心线程数，则将任务添加到阻塞队列中，接着再次检验线程池的运行状态，因为上次检测过之后线程池状态有可能发生了变化，如果线程池关闭了，那么移除任务，执行拒绝策略。如果线程依然是运行状态，但是线程池中没有线程，那么就调用addWorker方法创建线程，注意此时传入任务参数是null，即不指定执行任务，因为任务已经加入了阻塞队列。创建完线程后从阻塞队列中取出任务执行。
- 3.如果第2步将任务添加到阻塞队列失败了，说明阻塞队列任务已满，那么则会执行第三步，即创建非核心线程来执行任务，如果非核心线程创建失败那么就执行拒绝策略。

可以看到，代码的执行逻辑和我们在之前分析的线程池的工作流程是一样的。

接下来看下execute方法中创建线程的方法addWoker，addWoker方法承担了核心线程和非核心线程的创建，通过一个boolean参数core来区分是创建核心线程还是非核心线程。先来看addWorker方法前半部分的代码：

```
    // 返回值表示是否成功创建了线程
    private boolean addWorker(Runnable firstTask, boolean core) {
        // 这里做了一个retry标记，相当于goto.
        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 (;;) {
                // 根据core来确定创建最大线程数，超过最大值则创建线程失败，注意这里的最大值可能有三个corePoolSize、maximumPoolSize和线程池线程的最大容量
                int wc = workerCountOf(c);
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
```

```
                    return false;
                // 通过CAS来将线程数+1，如果成功则跳出循环，执行下边逻辑
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                c = ctl.get();  // Re-read ctl
                // 线程池的状态发生了改变，退回retry重新执行
                if (runStateOf(c) != rs)
                    continue retry;
            }
        }

        // ...省略后半部分

        return workerStarted;
    }
```

这部分代码会通过是否创建核心线程来确定线程池中线程数的值，如果是创建核心线程，那么最大值不能超过corePoolSize,如果是创建非核心线程那么线程数不能超过maximumPoolSize，另外无论是创建核心线程还是非核心线程，最大线程数都不能超过线程池允许的最大线程数CAPACITY(有可能设置的maximumPoolSize大于CAPACITY)。如果线程数大于最大值就返回false，创建线程失败。

接下来通过CAS将线程数加1，如果成功那么就break retry结束无限循环，如果CAS失败了则就continue retry从新开始for循环，注意这里的retry不是Java的关键字，是一个可以任意命名的字符。

接下来，如果能继续向下执行则开始执行创建线程并执行任务的工作了，看下addWorker方法的后半部分代码：

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

        // ...省略前半部分

        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            // 实例化一个Worker,内部封装了线程
            w = new Worker(firstTask);
            // 取出新建的线程
            final Thread t = w.thread;
            if (t != null) {
                // 这里使用ReentrantLock加锁保证线程安全
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    int c = ctl.get();
```

```
                    // 拿到锁重新检查线程池状态，只有处于RUNNING状态或者处于SHUTDOWN并且
firstTask==null时候才会创建线程
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        // 线程不是处于NEW状态，说明线程已经启动，抛出异常
                        if (t.isAlive())
                            throw new IllegalThreadStateException();
                        // 将线程加入线程队列，这里的worker是一个HashSet
                        workers.add(w);
                        workerAdded = true;
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    // 开启线程执行任务
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }
```

这部分逻辑其实比较容易理解，就是创建Worker并开启线程执行任务的过程，Worker是对线程的封装，创建的worker会被添加到ThreadPoolExecutor中的HashSet中。也就是线程池中的线程都维护在这个名为workers的HashSet中并被ThreadPoolExecutor所管理，HashSet中的线程可能处于正在工作的状态，也可能处于空闲状态，一旦达到指定的空闲时间，则会根据条件进行回收线程。

我们知道，线程调用start后就会开始执行线程的逻辑代码，执行完后线程的生命周期就结束了，那么线程池是如何保证Worker执行完任务后仍然不结束的呢？当线程空闲超时或者关闭线程池又是怎样进行线程回收的呢？这个实现逻辑其实就在Worker中。看下Worker的代码：

```
 private final class Worker
        extends AbstractQueuedSynchronizer
        implements Runnable
    {
        // 执行任务的线程
        final Thread thread;
```

```
        // 初始化Worker时传进来的任务，可能为null，如果不空，则创建和立即执行这个task，对应核心线程创建的情况
        Runnable firstTask;

        Worker(Runnable firstTask) {
            // 初始化时设置setate为-1
            setState(-1); // inhibit interrupts until runWorker
            this.firstTask = firstTask;
            // 通过线程工程创建线程
            this.thread = getThreadFactory().newThread(this);
        }

        // 线程的真正执行逻辑
        public void run() {
            runWorker(this);
        }

        // 判断线程是否是独占状态，如果不是意味着线程处于空闲状态
        protected boolean isHeldExclusively() {
            return getState() != 0;
        }

        // 获取锁
        protected boolean tryAcquire(int unused) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }
        // 释放锁
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }
        // ...
    }
```

Worker是位于ThreadPoolExecutor中的一个内部类，它继承了AQS，使用AQS来实现了独占锁的功能，但是并没支持可重入。这里使用不可重入的特性来表示线程的执行状态，即可以通过isHeldExclusively方法来判断，如果是独占状态，说明线程正在执行任务，如果非独占状态，说明线程处于空闲状态。

另外，Worker还实现了Runnable接口，因此它的执行逻辑就是在run方法中，run方法调用的是线程池中的runWorker(this)方法。任务的执行逻辑就在runWorker方法中，它的代码如下：

```
    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        // 取出Worker中的任务，可能为空
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            // task不为null或者阻塞队列中有任务，通过循环不断的从阻塞队列中取出任务执行
            while (task != null || (task = getTask()) != null) {
                w.lock();
                // ...
                try {
                    // 任务执行前的hook点
                    beforeExecute(wt, task);
                    try {
                        // 执行任务
                        task.run();
                        // 任务执行后的hook点
                        afterExecute(task, null);
                    } catch (Throwable ex) {
                        afterExecute(task, ex);
                        throw ex;
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            // 超时没有取到任务，则回收空闲超时的线程
            processWorkerExit(w, completedAbruptly);
        }
    }
```

可以看到，runWorker的核心逻辑就是不断通过getTask方法从阻塞队列中获取任务并执行，通过这样的方式实现了线程的复用，避免了创建线程。这里要注意的是这里是一个“生产者-消费者”模式，getTask是从阻塞队列中取任务，所以如果阻塞队列中没有任务的时候就会处于阻塞状态。

getTask中通过判断是否要回收线程而设置了等待超时时间，如果阻塞队列中一直没有任务，那么在等待keepAliveTime时间后会抛出异常。最终会走到上述代码的finally方法中，意味着有线程空闲时间超过了keepAliveTime时间，那么调用processWorkerExit方法移除Worker。processWorkerExit方法中没有复杂难以理解的逻辑，这里就不再贴代码了。我们重点看下getTask中是如何处理的，代码如下：

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

       for (;;) {
           int c = ctl.get();
           // ...


           // Flag1. 如果配置了allowCoreThreadTimeOut==true或者线程池中的线程数大于核心线程数，则timed为true，表示开启指定线程超时后被回收
           boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

           // ...

           try {
               // Flag2. 取出阻塞队列中的任务,注意如果timed为true，则会调用阻塞队列的poll方法，并设置超时时间为keepAliveTime，如果超时没有取到任务则会抛出异常。
               Runnable r = timed ?
                   workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                   workQueue.take();
               if (r != null)
                   return r;
               timedOut = true;
           } catch (InterruptedException retry) {
               timedOut = false;
           }
       }
   }
```

重点看getTask是如何处理空闲超时的逻辑的。我们知道，回收线程的条件是线程大于核心线程数或者配置了allowCoreThreadTimeOut为true,当线程空闲超时的情况下就会回收线程。上述代码在Flag1处先判断了如果线程池中的线程数大于核心线程数，或者开启了allowCoreThreadTimeOut，那么就需要开启线程空闲超时回收。

所有在Flag2处，timed为true的情况下调用了阻塞队列的poll方法，并传入了超时时间为keepAliveTime，如果在keepAliveTime时间内，阻塞队列一直为null那么会抛出异常，结束runWorker的循环。进而执行runWorker方法中回收线程的操作。

这里需要我们理解阻塞队列poll方法的使用，poll方法接受一个时间参数，是一个阻塞操作，在给定的时间内没有获取到数据就会抛出异常。其实说白了，阻塞队列就是一个使用ReentrantLock实现的“生产者-消费者”模式。

3.3 ThreadPoolExecutor的拒绝策略

上一小节中我们多次提到线程池的拒绝策略，它是在reject方法中实现的。实现代码也非常简单,代码如下：

```
    final void reject(Runnable command) {
        handler.rejectedExecution(command, this);
    }
```

通过调用handler的rejectedExecution方法实现。这里其实就是运用了策略模式，handler是一个RejectedExecutionHandler类型的成员变量，RejectedExecutionHandler是一个接口，只有一个rejectedExecution方法。在实例化线程池时构造方法中传入对应的拒绝策略实例即可。前文已经提到了Java提供的几种默认实现分别为DiscardPolicy、DiscardOldestPolicy、CallerRunsPolicy以及AbortPolicy。

3.4 ThreadPoolExecutor的shutdown

调用shutdown方法后，会将线程池标记为SHUTDOWN状态，上边execute的源码可以看出，只有线程池是RUNNING状态才接受任务，因此被标记位SHUTDOWN后，再提交任务会被线程池拒绝。shutdown的代码如下：

```
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //检查是否可以关闭线程
            checkShutdownAccess();
            // 将线程池状态置为SHUTDOWN状态
            advanceRunState(SHUTDOWN);
            // 尝试中断空闲线程
            interruptIdleWorkers();
            // 空方法，线程池关闭的hook点
            onShutdown();
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }

    private void interruptIdleWorkers() {
        interruptIdleWorkers(false);
    }
```

修改线程池为SHUTDOWN状态后，会调用interruptIdleWorkers去中断空闲线程线程，具体实现逻辑是在interruptIdleWorkers(boolean onlyOne)方法中，如下：

```
    private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers) {
                Thread t = w.thread;
                // 尝试tryLock获取锁，如果拿锁成功说明线程是空闲状态
                if (!t.isInterrupted() && w.tryLock()) {
                    try {
                        // 中断线程
                        t.interrupt();
                    } catch (SecurityException ignore) {
                    } finally {
                        w.unlock();
                    }
                }
                if (onlyOne)
                    break;
            }
        } finally {
            mainLock.unlock();
        }
    }
```

shutdown的逻辑比较简单，里边做了两件比较重要的事情，即先将线程池状态修改为SHUTDOWN，接着遍历所有Worker，将空闲的Worker进行中断。