个独占锁来控制同时只有一个线程可以进行入队和出队操作。另外，前者只使用了一个notEmpty条件变量而没有使用notFull，这是因为前者是无界队列，执行put操作时永远不会处于await状态，所以也不需要被唤醒。而take方法是阻塞方法，并且是可被中断的。当需要存放有优先级的元素时该队列比较有用。

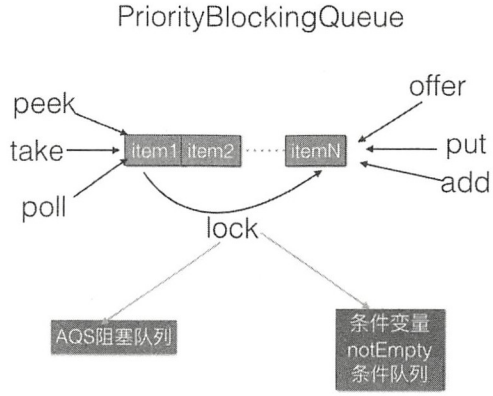

图7-33

7.5　DelayQueue原理探究

DelayQueue并发队列是一个无界阻塞延迟队列，队列中的每个元素都有个过期时间，当从队列获取元素时，只有过期元素才会出队列。队列头元素是最快要过期的元素。

7.5.1　DelayQueue类图结构

DelayQueue类图结构如图7-34所示。

由该图可知，DelayQueue内部使用PriorityQueue存放数据，使用ReentrantLock实现线程同步。另外，队列里面的元素要实现Delayed接口，由于每个元素都有一个过期时间，所以要实现获知当前元素还剩下多少时间就过期了的接口，由于内部使用优先级队列来实现，所以要实现元素之间相互比较的接口。

```
public interface Delayed extends Comparable<Delayed> {

    long getDelay(TimeUnit unit);
}
```

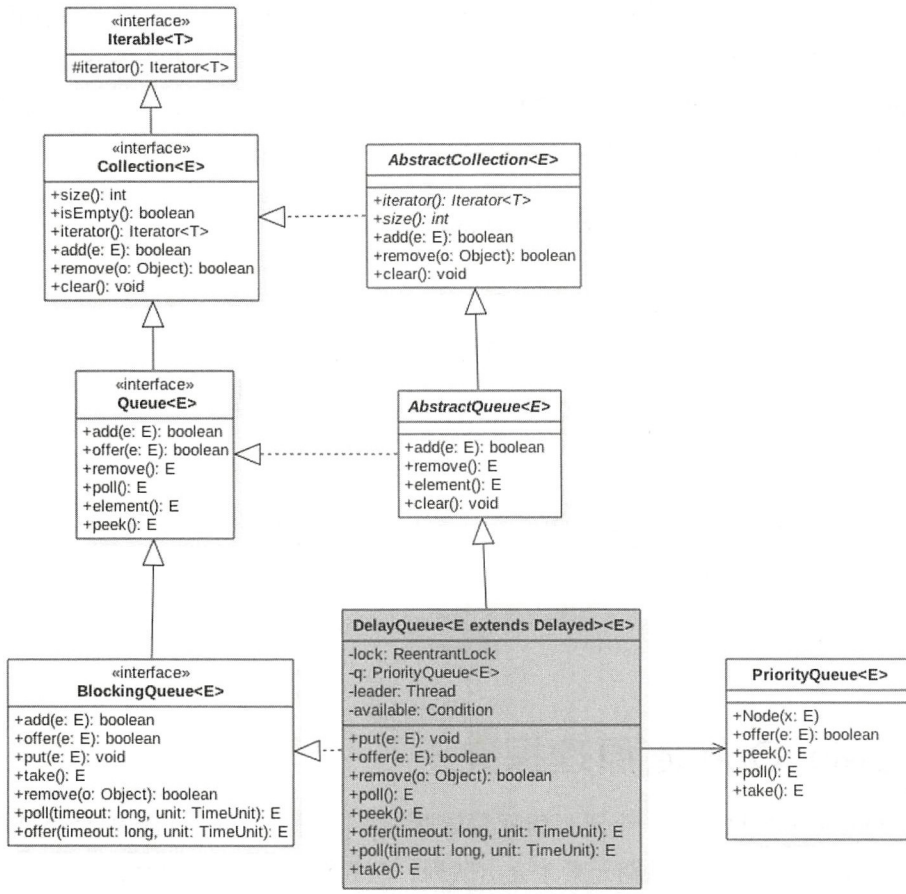

图 7-34

在如下代码中，条件变量 available 与 lock 锁是对应的，其目的是为了实现线程间同步。

```
private final Condition available = lock.newCondition();
```

其中 leader 变量的使用基于 Leader-Follower 模式的变体，用于尽量减少不必要的线程等待。当一个线程调用队列的 take 方法变为 leader 线程后，它会调用条件变量 available.awaitNanos(delay) 等待 delay 时间，但是其他线程（follwer 线程）则会调用 available.await() 进行无限等待。leader 线程延迟时间过期后，会退出 take 方法，并通过调用 available.signal() 方法唤醒一个 follwer 线程，被唤醒的 follwer 线程被选举为新的 leader 线程。

7.5.2　主要函数原理讲解

1. offer 操作

插入元素到队列，如果插入元素为 null 则抛出 NullPointerException 异常，否则由于是无界队列，所以一直返回 true。插入元素要实现 Delayed 接口。

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        q.offer(e);
        if (q.peek() == e) {//(2)
            leader = null;
            available.signal();
        }
        return true;
    } finally {
        lock.unlock();
    }
}
```

如上代码首先获取独占锁，然后添加元素到优先级队列，由于 q 是优先级队列，所以添加元素后，调用 q.peek() 方法返回的并不一定是当前添加的元素。如果代码（2）判断结果为 true，则说明当前元素 e 是最先将过期的，那么重置 leader 线程为 null，这时候激活 avaliable 变量条件队列里面的一个线程，告诉它队列里面有元素了。

2. take 操作

获取并移除队列里面延迟时间过期的元素，如果队列里面没有过期元素则等待。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            //获取但不移除队首元素(1)
            E first = q.peek();
            if (first == null)
                available.await();//(2)
```

```
                else {
                    long delay = first.getDelay(TimeUnit.NANOSECONDS);
                    if (delay <= 0)//(3)
                        return q.poll();
                    else if (leader != null)//(4)
                        available.await();
                    else {
                        Thread thisThread = Thread.currentThread();
                        leader = thisThread;//(5)
                        try {
                            available.awaitNanos(delay);//(6)
                        } finally {
                            if (leader == thisThread)
                                leader = null;
                        }
                    }
                }
            }
        } finally {
            if (leader == null && q.peek() != null)//(7)
                available.signal();
            lock.unlock();//(8)
        }
    }
```

如上代码首先获取独占锁 lock。假设线程 A 第一次调用队列的 take（）方法时队列为空，则执行代码（1）后 first==null，所以会执行代码（2）把当前线程放入 available 的条件队列里阻塞等待。

当有另外一个线程 B 执行 offer（item）方法并且添加元素到队列时，假设此时没有其他线程执行入队操作，则线程 B 添加的元素是队首元素，那么执行 q.peek()。

e 这时候就会重置 leader 线程为 null，并且激活条件变量的条件队列里面的一个线程。此时线程 A 就会被激活。

线程 A 被激活并循环后重新获取队首元素，这时候 first 就是线程 B 新增的元素，可知这时候 first 不为 null，则调用 first.getDelay(TimeUnit.NANOSECONDS) 方法查看该元素还剩余多少时间就要过期，如果 delay<=0 则说明已经过期，那么直接出队返回。否则查看 leader 是否为 null，不为 null 则说明其他线程也在执行 take，则把该线程放入条件队列。如果这时候 leader 为 null，则选取当前线程 A 为 leader 线程，然后执行代码 (5) 等待 delay 时间（这期间该线程会释放锁，所以其他线程可以 offer 添加元素，也可以 take 阻塞自己），

剩余过期时间到后，线程 A 会重新竞争得到锁，然后重置 leader 线程为 null，重新进入循环，这时候就会发现队头的元素已经过期了，则会直接返回队头元素。

在返回前会执行 finally 块里面的代码（7），代码（7）执行结果为 true 则说明当前线程从队列移除过期元素后，又有其他线程执行了入队操作，那么这时候调用条件变量的 singal 方法，激活条件队列里面的等待线程。

3. poll 操作

获取并移除队头过期元素，如果没有过期元素则返回 null。

```
public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E first = q.peek();
       //如果队列为空，或者不为空但是队头元素没有过期则返回null
        if (first == null || first.getDelay(TimeUnit.NANOSECONDS) > 0)
            return null;
        else
            return q.poll();
    } finally {
        lock.unlock();
    }
}
```

这段代码比较简单，首先获取独占锁，然后获取队头元素，如果队头元素为 null 或者还没过期则返回 null，否则返回队头元素。

4. size 操作

计算队列元素个数，包含过期的和没有过期的。

```
public int size() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return q.size();
    } finally {
        lock.unlock();
    }
}
```

这段代码比较简单，首先获取独占锁，然后调用优先级队列的 size 方法。

7.5.3 案例介绍

下面我们通过一个简单的案例来加深对 DelayQueue 的理解，代码如下。

```
public class TestDelay {

    static class DelayedEle implements Delayed {

        private final long delayTime; // 延迟时间
        private final long expire; // 到期时间
        private String taskName; // 任务名称

        public DelayedEle(long delay, String taskName) {
            delayTime = delay;
            this.taskName = taskName;
            expire = System.currentTimeMillis() + delay;
        }

        /**
         * 剩余时间=到期时间-当前时间
         */
        @Override
        public long getDelay(TimeUnit unit) {
            return unit.convert(this.expire - System.currentTimeMillis(), TimeUnit.
             MILLISECONDS);
        }

        /**
         * 优先级队列里面的优先级规则
         */
        @Override
        public int compareTo(Delayed o) {
            return (int) (this.getDelay(TimeUnit.MILLISECONDS) -
o.getDelay(TimeUnit.MILLISECONDS));
        }

        @Override
        public String toString() {
            final StringBuilder sb = new StringBuilder("DelayedEle{");
            sb.append("delay=").append(delayTime);
            sb.append(", expire=").append(expire);
```

```
            sb.append(", taskName='").append(taskName).append('\'');
            sb.append('}');
            return sb.toString();
        }
    }

    public static void main(String[] args) {

        //(1)创建delay队列
        DelayQueue<DelayedEle> delayQueue = new DelayQueue<DelayedEle>();

        //(2)创建延迟任务
        Random random = new Random();
        for (int i = 0; i < 10; ++i) {
            DelayedEle element = new DelayedEle(random.nextInt(500), "task:" + i);
            delayQueue.offer(element);

        }

        //(3)依次取出任务并打印
        DelayedEle ele = null;
        try {

            //(3.1)循环，如果想避免虚假唤醒，则不能把全部元素都打印出来
            for(;;){

                //(3.2)获取过期任务并打印
                while ((ele = delayQueue.take()) != null) {
                    System.out.println(ele.toString());
                }

            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}
```

如上代码首先创建延迟任务 DelayedEle 类，其中 delayTime 表示当前任务需要延迟多少 ms 时间过期，expire 则是当前时间的 ms 值加上 delayTime 的值。另外，实现了 Delayed 接口，实现了 long getDelay(TimeUnit unit) 方法用来获取当前元素还剩下多少时间

过期，实现了 int compareTo(Delayed o) 方法用来决定优先级队列元素的比较规则。

在 main 函数内首先创建了一个延迟队列，然后使用随机数生成器生成了 10 个延迟任务，最后通过循环依次获取延迟任务，并打印。运行上面代码，一个可能的输出如下所示。

```
DelayedEle{delay=73, expire=1523428917194, taskName='task:4'}
DelayedEle{delay=97, expire=1523428917218, taskName='task:5'}
DelayedEle{delay=150, expire=1523428917272, taskName='task:9'}
DelayedEle{delay=205, expire=1523428917326, taskName='task:3'}
DelayedEle{delay=236, expire=1523428917354, taskName='task:1'}
DelayedEle{delay=324, expire=1523428917446, taskName='task:7'}
DelayedEle{delay=340, expire=1523428917461, taskName='task:2'}
DelayedEle{delay=392, expire=1523428917510, taskName='task:0'}
DelayedEle{delay=403, expire=1523428917525, taskName='task:8'}
DelayedEle{delay=416, expire=1523428917538, taskName='task:6'}
```

可见，出队的顺序和 delay 时间有关，而与创建任务的顺序无关。

7.5.4 小结

本节讲解了 DelayQueue 队列（见图 7-34），其内部使用 PriorityQueue 存放数据，使用 ReentrantLock 实现线程同步。另外队列里面的元素要实现 Delayed 接口，其中一个是获取当前元素到过期时间剩余时间的接口，在出队时判断元素是否过期了，一个是元素之间比较的接口，因为这是一个有优先级的队列。

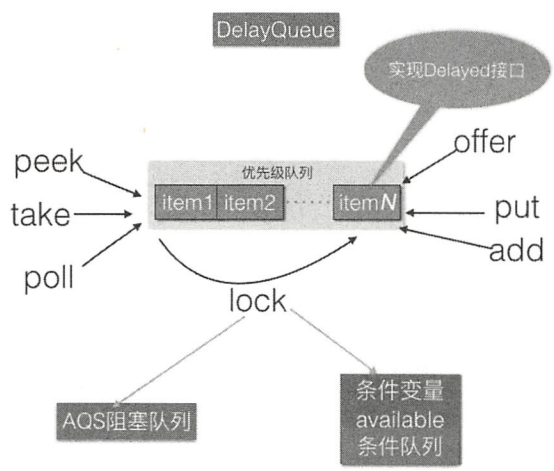

图 7-35

第8章

Java并发包中线程池ThreadPoolExecutor原理探究

8.1 介绍

线程池主要解决两个问题：一是当执行大量异步任务时线程池能够提供较好的性能。在不使用线程池时，每当需要执行异步任务时直接 new 一个线程来运行，而线程的创建和销毁是需要开销的。线程池里面的线程是可复用的，不需要每次执行异步任务时都重新创建和销毁线程。二是线程池提供了一种资源限制和管理的手段，比如可以限制线程的个数，动态新增线程等。每个 ThreadPoolExecutor 也保留了一些基本的统计数据，比如当前线程池完成的任务数目等。

另外，线程池也提供了许多可调参数和可扩展性接口，以满足不同情境的需要，程序员可以使用更方便的 Executors 的工厂方法，比如 newCachedThreadPool（线程池线程个数最多可达 Integer.MAX_VALUE，线程自动回收）、newFixedThreadPool（固定大小的线程池）和 newSingleThreadExecutor（单个线程）等来创建线程池，当然用户还可以自定义。

8.2 类图介绍

在如图 8-1 所示的类图中，Executors 其实是个工具类，里面提供了好多静态方法，这些方法根据用户选择返回不同的线程池实例。ThreadPoolExecutor 继承了 AbstractExecutorService，成员变量 ctl 是一个 Integer 的原子变量，用来记录线程池状态和线程池中线程个数，类似于 ReentrantReadWriteLock 使用一个变量来保存两种信息。

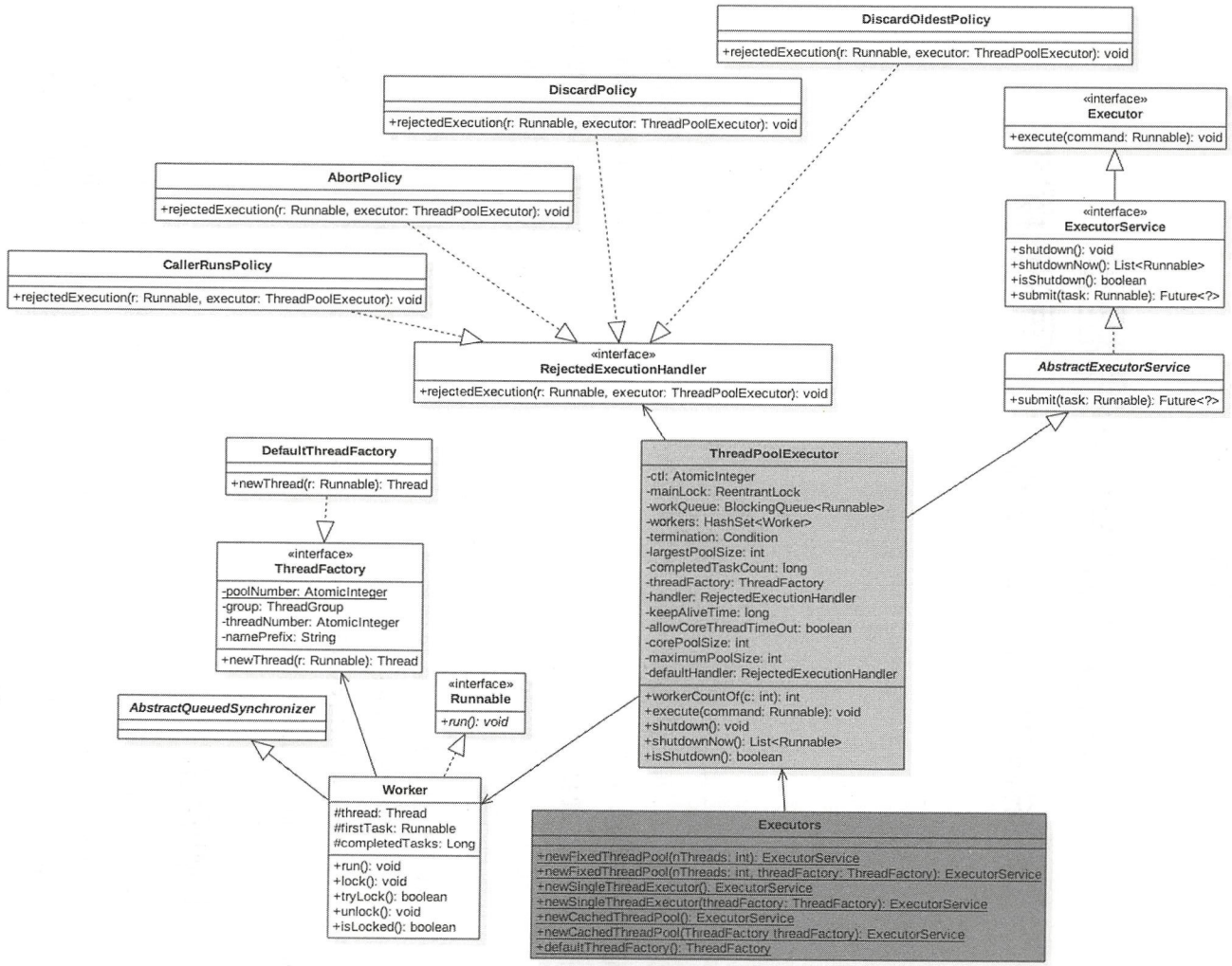
DiscardOldestPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
DiscardPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
«interface»
Executor
+execute(command: Runnable): void
AbortPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
«interface»
ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+submit(task: Runnable): Future<?>
CallerRunsPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
«interface»
RejectedExecutionHandler
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
AbstractExecutorService
+submit(task: Runnable): Future<?>
DefaultThreadFactory
+newThread(r: Runnable): Thread
ThreadPoolExecutor
-ctl: AtomicInteger
-mainLock: ReentrantLock
-workQueue: BlockingQueue<Runnable>
-workers: HashSet<Worker>
-termination: Condition
-largestPoolSize: int
-completedTaskCount: long
-threadFactory: ThreadFactory
-handler: RejectedExecutionHandler
-keepAliveTime: long
-allowCoreThreadTimeOut: boolean
-corePoolSize: int
-maximumPoolSize: int
-defaultHandler: RejectedExecutionHandler
+workerCountOf(c: int): int
+execute(command: Runnable): void
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
«interface»
ThreadFactory
-poolNumber: AtomicInteger
-group: ThreadGroup
-threadNumber: AtomicInteger
-namePrefix: String
+newThread(r: Runnable): Thread
«interface»
Runnable
+run(): void
AbstractQueuedSynchronizer
Worker
#thread: Thread
#firstTask: Runnable
#completedTasks: Long
+run(): void
+lock(): void
+tryLock(): boolean
+unlock(): void
+isLocked(): boolean
Executors
+newFixedThreadPool(nThreads: int): ExecutorService
+newFixedThreadPool(nThreads: int, threadFactory: ThreadFactory): ExecutorService
+newSingleThreadExecutor(): ExecutorService
+newSingleThreadExecutor(threadFactory: ThreadFactory): ExecutorService
+newCachedThreadPool(): ExecutorService
+newCachedThreadPool(ThreadFactory threadFactory): ExecutorService
+defaultThreadFactory(): ThreadFactory

图 8-1

这里假设 Integer 类型是 32 位二进制表示，则其中高 3 位用来表示线程池状态，后面 29 位用来记录线程池线程个数。

```
//（高3位）用来表示线程池状态，（低29位）用来表示线程个数
//默认是RUNNING状态，线程个数为0
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

//线程个数掩码位数，并不是所有平台的int类型都是32位的，所以准确地说，是具体平台下Integer的二进制
位数-3后的剩余位数所表示的数才是线程的个数
private static final int COUNT_BITS = Integer.SIZE - 3;

//线程最大个数(低29位)00011111111111111111111111111111
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
```

线程池状态：

```
//（高3位）：11100000000000000000000000000000
private static final int RUNNING    = -1 << COUNT_BITS;

//（高3位）：00000000000000000000000000000000
private static final int SHUTDOWN   =  0 << COUNT_BITS;

//（高3位）：00100000000000000000000000000000
private static final int STOP       =  1 << COUNT_BITS;

//（高3位）：01000000000000000000000000000000
private static final int TIDYING    =  2 << COUNT_BITS;

//（高3位）：01100000000000000000000000000000
private static final int TERMINATED =  3 << COUNT_BITS;


// 获取高3位（运行状态）
private static int runStateOf(int c)     { return c & ~CAPACITY; }

//获取低29位（线程个数）
private static int workerCountOf(int c)  { return c & CAPACITY; }

//计算ctl新值（线程状态与线程个数）
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

线程池状态含义如下。

- RUNNING：接受新任务并且处理阻塞队列里的任务。

- SHUTDOWN：拒绝新任务但是处理阻塞队列里的任务。
- STOP：拒绝新任务并且抛弃阻塞队列里的任务，同时会中断正在处理的任务。
- TIDYING：所有任务都执行完（包含阻塞队列里面的任务）后当前线程池活动线程数为 0，将要调用 terminated 方法。
- TERMINATED：终止状态。terminated 方法调用完成以后的状态。

线程池状态转换列举如下。

- RUNNING -> SHUTDOWN：显式调用 shutdown() 方法，或者隐式调用了 finalize() 方法里面的 shutdown() 方法。
- RUNNING 或 SHUTDOWN)-> STOP：显式调用 shutdownNow() 方法时。
- SHUTDOWN -> TIDYING：当线程池和任务队列都为空时。
- STOP -> TIDYING：当线程池为空时。
- TIDYING -> TERMINATED：当 terminated() hook 方法执行完成时。

线程池参数如下。

- corePoolSize：线程池核心线程个数。
- workQueue：用于保存等待执行的任务的阻塞队列，比如基于数组的有界 ArrayBlockingQueue、基于链表的无界 LinkedBlockingQueue、最多只有一个元素的同步队列 SynchronousQueue 及优先级队列 PriorityBlockingQueue 等。
- maximunPoolSize：线程池最大线程数量。
- ThreadFactory：创建线程的工厂。
- RejectedExecutionHandler：饱和策略，当队列满并且线程个数达到 maximunPoolSize 后采取的策略，比如 AbortPolicy（抛出异常）、CallerRunsPolicy（使用调用者所在线程来运行任务）、DiscardOldestPolicy（调用 poll 丢弃一个任务，执行当前任务）及 DiscardPolicy（默默丢弃，不抛出异常）
- keeyAliveTime：存活时间。如果当前线程池中的线程数量比核心线程数量多，并且是闲置状态，则这些闲置的线程能存活的最大时间。
- TimeUnit：存活时间的时间单位。

线程池类型如下。

- newFixedThreadPool：创建一个核心线程个数和最大线程个数都为 nThreads 的线程池，并且阻塞队列长度为 Integer.MAX_VALUE。keeyAliveTime=0 说明只要线程个

数比核心线程个数多并且当前空闲则回收。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
//使用自定义线程创建工厂
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory);
}
```

- newSingleThreadExecutor：创建一个核心线程个数和最大线程个数都为 1 的线程池，并且阻塞队列长度为 Integer.MAX_VALUE。keeyAliveTime=0 说明只要线程个数比核心线程个数多并且当前空闲则回收。

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

//使用自己的线程工厂
public static ExecutorService newSingleThreadExecutor(ThreadFactory
threadFactory) {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>(),
                                threadFactory));
}
```

- newCachedThreadPool：创建一个按需创建线程的线程池，初始线程个数为 0，最多线程个数为 Integer.MAX_VALUE，并且阻塞队列为同步队列。keeyAliveTime=60 说明只要当前线程在 60s 内空闲则回收。这个类型的特殊之处在于，加入同步队列的任务会被马上执行，同步队列里面最多只有一个任务。

```
public static ExecutorService newCachedThreadPool() {
```

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

    //使用自定义的线程工厂
    public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>(),
                                      threadFactory);
    }
```

如上 ThreadPoolExecutor 类图所示，其中 mainLock 是独占锁，用来控制新增 Worker 线程操作的原子性。termination 是该锁对应的条件队列，在线程调用 awaitTermination 时用来存放阻塞的线程。

Worker 继承 AQS 和 Runnable 接口，是具体承载任务的对象。Worker 继承了 AQS，自己实现了简单不可重入独占锁，其中 state=0 表示锁未被获取状态，state=1 表示锁已经被获取的状态，state=—1 是创建 Worker 时默认的状态，创建时状态设置为 –1 是为了避免该线程在运行 runWorker() 方法前被中断，下面会具体讲解。其中变量 firstTask 记录该工作线程执行的第一个任务，thread 是具体执行任务的线程。

DefaultThreadFactory 是线程工厂，newThread 方法是对线程的一个修饰。其中 poolNumber 是个静态的原子变量，用来统计线程工厂的个数，threadNumber 用来记录每个线程工厂创建了多少线程，这两个值也作为线程池和线程的名称的一部分。

8.3 源码分析

8.3.1 public void execute(Runnable command)

execute 方法的作用是提交任务 command 到线程池进行执行。用户线程提交任务到线程池的模型图如图 8-2 所示。

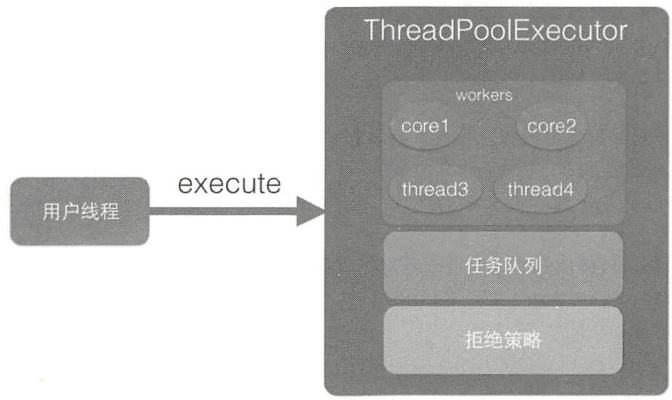

图 8-2

从该图可以看出，ThreadPoolExecutor 的实现实际是一个生产消费模型，当用户添加任务到线程池时相当于生产者生产元素，workers 线程工作集中的线程直接执行任务或者从任务队列里面获取任务时则相当于消费者消费元素。

用户线程提交任务的 execute 方法的具体代码如下。

```
public void execute(Runnable command) {

    //(1) 如果任务为null，则抛出NPE异常
    if (command == null)
        throw new NullPointerException();

    //(2)获取当前线程池的状态+线程个数变量的组合值
    int c = ctl.get();

    //(3)当前线程池中线程个数是否小于corePoolSize,小于则开启新线程运行
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }

    //(4)如果线程池处于RUNNING状态，则添加任务到阻塞队列
    if (isRunning(c) && workQueue.offer(command)) {

        //(4.1)二次检查
        int recheck = ctl.get();
```

```
        //(4.2)如果当前线程池状态不是RUNNING则从队列中删除任务，并执行拒绝策略
        if (! isRunning(recheck) && remove(command))
            reject(command);

        //(4.3)否则如果当前线程池为空，则添加一个线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //(5)如果队列满，则新增线程，新增失败则执行拒绝策略
    else if (!addWorker(command, false))
        reject(command);
}
```

代码（3）判断如果当前线程池中线程个数小于 corePoolSize，会向 workers 里面新增一个核心线程（core 线程）执行该任务。

如果当前线程池中线程个数大于等于 corePoolSize 则执行代码（4）。如果当前线程池处于 RUNNING 状态则添加当前任务到任务队列。这里需要判断线程池状态是因为有可能线程池已经处于非 RUNNING 状态，而在非 RUNNING 状态下是要抛弃新任务的。

如果向任务队列添加任务成功，则代码（4.2）对线程池状态进行二次校验，这是因为添加任务到任务队列后，执行代码（4.2）前有可能线程池的状态已经变化了。这里进行二次校验，如果当前线程池状态不是 RUNNING 了则把任务从任务队列移除，移除后执行拒绝策略；如果二次校验通过，则执行代码（4.3）重新判断当前线程池里面是否还有线程，如果没有则新增一个线程。

如果代码（4）添加任务失败，则说明任务队列已满，那么执行代码（5）尝试新开启线程（如图 8-1 中的 thread3 和 thread4）来执行该任务，如果当前线程池中线程个数 >maximumPoolSize 则执行拒绝策略。

下面分析下新增线程的 addWorkder 方法，代码如下。

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

        //(6) 检查队列是否只在必要时为空
        if (rs >= SHUTDOWN &&
```

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

        //(7)循环CAS增加线程个数
        for (;;) {
            int wc = workerCountOf(c);

            //(7.1)如果线程个数超限则返回false
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            //(7.2)CAS增加线程个数，同时只有一个线程成功
            if (compareAndIncrementWorkerCount(c))
                break retry;
            //(7.3)CAS失败了，则看线程池状态是否变化了，变化则跳到外层循环重新尝试获取线程池
                状态，否则内层循环重新CAS。
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
        }
    }

    //(8)到这里说明CAS成功了
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        //(8.1)创建worker
        final ReentrantLock mainLock = this.mainLock;
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {

            //(8.2)加独占锁，为了实现workers同步，因为可能多个线程调用了线程池的execute方法
            mainLock.lock();
            try {

                //(8.3)重新检查线程池状态，以避免在获取锁前调用了shutdown接口
                int c = ctl.get();
                int rs = runStateOf(c);
```

```
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    //(8.4)添加任务
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            //(8.5)添加成功后则启动任务
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

代码比较长，主要分两个部分：第一部分双重循环的目的是通过 CAS 操作增加线程数；第二部分主要是把并发安全的任务添加到 workers 里面，并且启动任务执行。

首先来分析第一部分的代码（6）。

```
rs >= SHUTDOWN &&
               ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty())
```

展开！运算后等价于

```
s >= SHUTDOWN &&
                (rs != SHUTDOWN ||//(I)
               firstTask != null ||//(II)
               workQueue.isEmpty())//(III)
```

也就是说代码（6）在下面几种情况下会返回 false：

- （I）当前线程池状态为 STOP、TIDYING 或 TERMINATED。
- （II）当前线程池状态为 SHUTDOWN 并且已经有了第一个任务。
- （III）当前线程池状态为 SHUTDOWN 并且任务队列为空。

内层循环的作用是使用 CAS 操作增加线程数，代码（7.1）判断如果线程个数超限则返回 false，否则执行代码（7.2）CAS 操作设置线程个数，CAS 成功则退出双循环，CAS 失败则执行代码（7.3）看当前线程池的状态是否变化了，如果变了，则再次进入外层循环重新获取线程池状态，否则进入内层循环继续进行 CAS 尝试。

执行到第二部分的代码（8）时说明使用 CAS 成功地增加了线程个数，但是现在任务还没开始执行。这里使用全局的独占锁来控制把新增的 Worker 添加到工作集 workers 中。代码（8.1）创建了一个工作线程 Worker。

代码（8.2) 获取了独占锁，代码（8.3）重新检查线程池状态，这是为了避免在获取锁前其他线程调用了 shutdown 关闭了线程池。如果线程池已经被关闭，则释放锁，新增线程失败，否则执行代码（8.4）添加工作线程到线程工作集，然后释放锁。代码（8.5）判断如果新增工作线程成功，则启动工作线程。

8.3.2　工作线程 Worker 的执行

用户线程提交任务到线程池后，由 Worker 来执行。先看下 Worker 的构造函数。

```
Worker(Runnable firstTask) {
    setState(-1); // 在调用runWorker前禁止中断
    this.firstTask = firstTask;
    this.thread = getThreadFactory().newThread(this);//创建一个线程
}
```

在构造函数内首先设置 Worker 的状态为 –1，这是为了避免当前 Worker 在调用 runWorker 方法前被中断（当其他线程调用了线程池的 shutdownNow 时，如果 Worker 状态 >=0 则会中断该线程）。这里设置了线程的状态为 –1，所以该线程就不会被中断了。在如下 runWorker 代码中，运行代码（9）时会调用 unlock 方法，该方法把 status 设置为了 0，所以这时候调用 shutdownNow 会中断 Worker 线程。

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

```
        w.unlock(); //(9)将state设置为0，允许中断
        boolean completedAbruptly = true;
        try {
            //(10)
            while (task != null || (task = getTask()) != null) {

                //(10.1)
                w.lock();
                ...
                try {
                    //(10.2)执行任务前干一些事情
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        task.run();//(10.3)执行任务
                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {
                        //(10.4)执行任务完毕后干一些事情
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    //(10.5)统计当前Worker完成了多少个任务
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {

            //(11)执行清理工作
            processWorkerExit(w, completedAbruptly);
        }
    }
```

在如上代码（10）中，如果当前 task==null 或者调用 getTask 从任务队列获取的任务返回 null，则跳转到代码（11）执行。如果 task 不为 null 则执行代码（10.1）获取工作线程内部持有的独占锁，然后执行扩展接口代码（10.2）在具体任务执行前做一些事情。代

码（10.3）具体执行任务，代码（10.4）在任务执行完毕后做一些事情，代码（10.5）统计当前 Worker 完成了多少个任务，并释放锁。

这里在执行具体任务期间加锁，是为了避免在任务运行期间，其他线程调用了 shutdown 后正在执行的任务被中断（shutdown 只会中断当前被阻塞挂起的线程）

代码（11）执行清理任务，其代码如下。

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
     ...

    //(11.1)统计整个线程池完成的任务个数,并从工作集里面删除当前Woker
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        completedTaskCount += w.completedTasks;
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }

    //(11.2)尝试设置线程池状态为TERMINATED, 如果当前是SHUTDONW状态并且工作队列为空
    //或者当前是STOP状态,当前线程池里面没有活动线程
    tryTerminate();

    //(11.3)如果当前线程个数小于核心个数, 则增加
    int c = ctl.get();
    if (runStateLessThan(c, STOP)) {
        if (!completedAbruptly) {
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        addWorker(null, false);
    }
}
```

在如上代码中，代码（11.1）统计线程池完成任务个数，并且在统计前加了全局锁。把在当前工作线程中完成的任务累加到全局计数器，然后从工作集中删除当前 Worker。

代码（11.2）判断如果当前线程池状态是 SHUTDOWN 并且工作队列为空，或

者当前线程池状态是 STOP 并且当前线程池里面没有活动线程，则设置线程池状态为 TERMINATED。如果设置为了 TERMINATED 状态，则还需要调用条件变量 termination 的 signalAll（）方法激活所有因为调用线程池的 awaitTermination 方法而被阻塞的线程。

代码（11.3）则判断当前线程池里面线程个数是否小于核心线程个数，如果是则新增一个线程。

8.3.3 shutdown 操作

调用 shutdown 方法后，线程池就不会再接受新的任务了，但是工作队列里面的任务还是要执行的。该方法会立刻返回，并不等待队列任务完成再返回。

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //(12)权限检查
        checkShutdownAccess();

        //(13)设置当前线程池状态为SHUTDOWN，如果已经是SHUTDOWN则直接返回
        advanceRunState(SHUTDOWN);

        //(14)设置中断标志
        interruptIdleWorkers();
        onShutdown();
    } finally {
        mainLock.unlock();
    }
    //(15)尝试将状态变为TERMINATED
    tryTerminate();
}
```

在如上代码中，代码（12）检查看是否设置了安全管理器，是则看当前调用 shutdown 命令的线程是否有关闭线程的权限，如果有权限则还要看调用线程是否有中断工作线程的权限，如果没有权限则抛出 SecurityException 或者 NullPointerException 异常。

其中代码（13）的内容如下，如果当前线程池状态 >=SHUTDOWN 则直接返回，否则设置为 SHUTDOWN 状态。

```
private void advanceRunState(int targetState) {
```

```
    for (;;) {
        int c = ctl.get();
        if (runStateAtLeast(c, targetState) ||
            ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
            break;
    }
}
```

代码（14）的内容如下，其设置所有空闲线程的中断标志。这里首先加了全局锁，同时只有一个线程可以调用 shutdown 方法设置中断标志。然后尝试获取 Worker 自己的锁，获取成功则设置中断标志。由于正在执行的任务已经获取了锁，所以正在执行的任务没有被中断。这里中断的是阻塞到 getTask() 方法并企图从队列里面获取任务的线程，也就是空闲线程。

```
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            //如果工作线程没有被中断，并且没有正在运行则设置中断标志
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}

final void tryTerminate() {
        for (;;) {
            ...
            int c = ctl.get();
            ...
```

```
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {//设置当前线程池状态为TIDYING
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        terminated();
                    } finally {
                        //设置当前线程池状态为TERMINATED
                        ctl.set(ctlOf(TERMINATED, 0));
                        //激活因调用条件变量termination的await系列方法而被阻塞的所有线程
                        termination.signalAll();
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
        }
    }
```

在如上代码中，首先使用 CAS 设置当前线程池状态为 TIDYING，如果设置成功则执行扩展接口 terminated 在线程池状态变为 TERMINATED 前做一些事情，然后设置当前线程池状态为 TERMINATED。最后调用 termination.signalAll() 激活因调用条件变量 termination 的 await 系列方法而被阻塞的所有线程，关于这一点随后讲到 awaitTermination 方法时具体讲解。

8.3.4 shutdownNow 操作

调用 shutdownNow 方法后，线程池就不会再接受新的任务了，并且会丢弃工作队列里面的任务，正在执行的任务会被中断，该方法会立刻返回，并不等待激活的任务执行完成。返回值为这时候队列里面被丢弃的任务列表。

```
public List<Runnable> shutdownNow() {


    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();//(16)权限检查
        advanceRunState(STOP);//(17) 设置线程池状态为STOP
```

```
            interruptWorkers();//(18)中断所有线程
            tasks = drainQueue();//(19)将队列任务移动到tasks中
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }
```

在如上代码中，首先调用代码（16）检查权限，然后调用代码（17）设置当前线程池状态为 STOP，随后执行代码（18）中断所有的工作线程。这里需要注意的是，中断的所有线程包含空闲线程和正在执行任务的线程。

```
    private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                w.interruptIfStarted();
        } finally {
            mainLock.unlock();
        }
    }
```

然后代码（19）将当前任务队列里面的任务移动到 tasks 列表。

8.3.5 awaitTermination 操作

当线程调用 awaitTermination 方法后，当前线程会被阻塞，直到线程池状态变为 TERMINATED 才返回，或者等待时间超时才返回。整个过程中独占锁的代码如下。

```
    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (;;) {
                if (runStateAtLeast(ctl.get(), TERMINATED))
                    return true;
                if (nanos <= 0)
                    return false;
                nanos = termination.awaitNanos(nanos);
```

```
            }
        } finally {
            mainLock.unlock();
        }
    }
```

如上代码首先获取独占锁，然后在无限循环内部判断当前线程池状态是否至少是TERMINATED状态，如果是则直接返回，否则说明当前线程池里面还有线程在执行，则看设置的超时时间nanos是否小于0，小于0则说明不需要等待，那就直接返回，如果大于0则调用条件变量termination的awaitNanos方法等待nanos时间，期望在这段时间内线程池状态变为TERMINATED。

在讲解shutdown方法时提到过，当线程池状态变为TERMINATED时，会调用termination.signalAll()用来激活调用条件变量termination的await系列方法被阻塞的所有线程，所以如果在调用awaitTermination之后又调用了shutdown方法，并且在shutdown内部将线程池状态设置为TERMINATED，则termination.awaitNanos方法会返回。

另外在工作线程Worker的runWorker方法内，当工作线程运行结束后，会调用processWorkerExit方法，在processWorkerExit方法内部也会调用tryTerminate方法测试当前是否应该把线程池状态设置为TERMINATED，如果是，则也会调用termination.signalAll()用来激活调用线程池的awaitTermination方法而被阻塞的线程。

而且当等待时间超时后，termination.awaitNanos也会返回，这时候会重新检查当前线程池状态是否为TERMINATED，如果是则直接返回，否则继续阻塞挂起自己。

8.4 总结

线程池巧妙地使用一个Integer类型的原子变量来记录线程池状态和线程池中的线程个数。通过线程池状态来控制任务的执行，每个Worker线程可以处理多个任务。线程池通过线程的复用减少了线程创建和销毁的开销。

第9章

Java并发包中ScheduledThreadPoolExecutor原理探究

9.1 介绍

前面讲解了Java中线程池ThreadPoolExecutor的原理，ThreadPoolExecutor只是Executors工具类的一部分功能。下面来介绍另外一部分功能，也就是ScheduledThreadPoolExecutor的实现，这是一个可以在指定一定延迟时间后或者定时进行任务调度执行的线程池。

9.2 类图介绍

类图结构如图9-1所示。

Executors其实是个工具类，它提供了好多静态方法，可根据用户的选择返回不同的线程池实例。ScheduledThreadPoolExecutor继承了ThreadPoolExecutor并实现了ScheduledExecutorService接口。线程池队列是DelayedWorkQueue，其和DelayedQueue类似，是一个延迟队列。

ScheduledFutureTask是具有返回值的任务，继承自FutureTask。FutureTask的内部有一个变量state用来表示任务的状态，一开始状态为NEW，所有状态为

```
private static final int NEW          = 0;//初始状态
private static final int COMPLETING   = 1;//执行中状态
private static final int NORMAL       = 2;//正常运行结束状态
private static final int EXCEPTIONAL  = 3;//运行中异常
private static final int CANCELLED    = 4;//任务被取消
private static final int INTERRUPTING = 5;//任务正在被中断
private static final int INTERRUPTED  = 6;//任务已经被中断
```

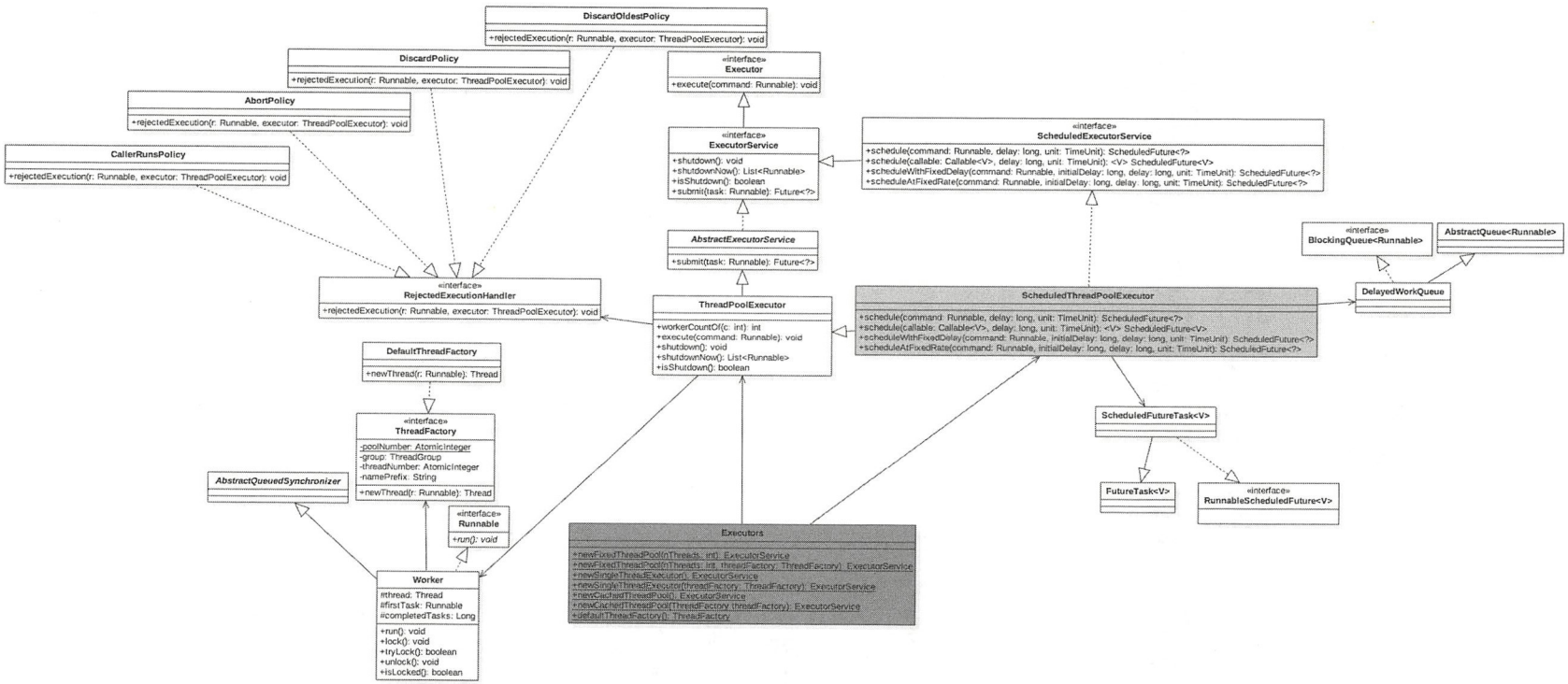
DiscardOldestPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
DiscardPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
«interface»
Executor
+execute(command: Runnable): void
AbortPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
CallerRunsPolicy
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
«interface»
ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+submit(task: Runnable): Future<?>
«interface»
ScheduledExecutorService
+schedule(command: Runnable, delay: long, unit: TimeUnit): ScheduledFuture<?>
+schedule(callable: Callable<V>, delay: long, unit: TimeUnit): <V> ScheduledFuture<V>
+scheduleWithFixedDelay(command: Runnable, initialDelay: long, delay: long, unit: TimeUnit): ScheduledFuture<?>
+scheduleAtFixedRate(command: Runnable, initialDelay: long, delay: long, unit: TimeUnit): ScheduledFuture<?>
AbstractExecutorService
+submit(task: Runnable): Future<?>
«interface»
BlockingQueue<Runnable>
AbstractQueue<Runnable>
DelayedWorkQueue
«interface»
RejectedExecutionHandler
+rejectedExecution(r: Runnable, executor: ThreadPoolExecutor): void
ThreadPoolExecutor
+workerCountOf(c: int): int
+execute(command: Runnable): void
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
ScheduledThreadPoolExecutor
+schedule(command: Runnable, delay: long, unit: TimeUnit): ScheduledFuture<?>
+schedule(callable: Callable<V>, delay: long, unit: TimeUnit): <V> ScheduledFuture<V>
+scheduleWithFixedDelay(command: Runnable, initialDelay: long, delay: long, unit: TimeUnit): ScheduledFuture<?>
+scheduleAtFixedRate(command: Runnable, initialDelay: long, delay: long, unit: TimeUnit): ScheduledFuture<?>
DefaultThreadFactory
+newThread(r: Runnable): Thread
«interface»
ThreadFactory
-poolNumber: AtomicInteger
-group: ThreadGroup
-threadNumber: AtomicInteger
-namePrefix: String
+newThread(r: Runnable): Thread
ScheduledFutureTask<V>
FutureTask<V>
«interface»
RunnableScheduledFuture<V>
AbstractQueuedSynchronizer
«interface»
Runnable
+run(): void
Executors
+newFixedThreadPool(nThreads: int): ExecutorService
+newFixedThreadPool(nThreads: int, threadFactory: ThreadFactory): ExecutorService
+newSingleThreadExecutor(): ExecutorService
+newSingleThreadExecutor(threadFactory: ThreadFactory): ExecutorService
+newCachedThreadPool(): ExecutorService
+newCachedThreadPool(threadFactory: threadFactory): ExecutorService
+defaultThreadFactory(): ThreadFactory
Worker
#thread: Thread
#firstTask: Runnable
#completedTasks: Long
+run(): void
+lock(): void
+tryLock(): boolean
+unlock(): void
+isLocked(): boolean

图 9-1

可能的任务状态转换路径为

```
NEW -> COMPLETING -> NORMAL //初始状态->执行中->正常结束
NEW -> COMPLETING -> EXCEPTIONAL//初始状态->执行中->执行异常
NEW -> CANCELLED//初始状态->任务取消
NEW -> INTERRUPTING -> INTERRUPTED//初始状态->被中断中->被中断
```

ScheduledFutureTask 内部还有一个变量 period 用来表示任务的类型，任务类型如下：

- period=0，说明当前任务是一次性的，执行完毕后就退出了。
- period 为负数，说明当前任务为 fixed-delay 任务，是固定延迟的定时可重复执行任务。
- period 为正数，说明当前任务为 fixed-rate 任务，是固定频率的定时可重复执行任务。

ScheduledThreadPoolExecutor 的一个构造函数如下，由该构造函数可知线程池队列是 DelayedWorkQueue。

```
//使用改造后的Delayqueue
public ScheduledThreadPoolExecutor(int corePoolSize) {
    //调用父类ThreadPoolExecutor的构造函数
    super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
          new DelayedWorkQueue());
}

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

9.3 原理剖析

本节讲解三个重要函数。

- schedule(Runnable command, long delay,TimeUnit unit)
- scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit)
- scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit)

9.3.1 schedule(Runnable command, long delay,TimeUnit unit) 方法

该方法的作用是提交一个延迟执行的任务，任务从提交时间算起延迟单位为 unit 的 delay 时间后开始执行。提交的任务不是周期性任务，任务只会执行一次，代码如下。

```
public ScheduledFuture<?> schedule(Runnable command,
                                   long delay,
                                   TimeUnit unit) {
    //(1)参数校验
    if (command == null || unit == null)
        throw new NullPointerException();

    //(2)任务转换
    RunnableScheduledFuture<?> t = decorateTask(command,
        new ScheduledFutureTask<Void>(command, null,
                                      triggerTime(delay, unit)));
    //(3)添加任务到延迟队列
    delayedExecute(t);
    return t;
}
```

I. 如上代码（1）进行参数校验，如果 command 或者 unit 为 null，则抛出 NPE 异常。

II. 代码（2）装饰任务，把提交的 command 任务转换为 ScheduledFutureTask。ScheduledFutureTask 是具体放入延迟队列里面的东西。由于是延迟任务，所以 ScheduledFutureTask 实现了 long getDelay(TimeUnit unit) 和 int compareTo(Delayed other) 方法。triggerTime 方法将延迟时间转换为绝对时间，也就是把当前时间的纳秒数加上延迟的纳秒数后的 long 型值。ScheduledFutureTask 的构造函数如下。

```
ScheduledFutureTask(Runnable r, V result, long ns) {
  //调用父类FutureTask的构造函数
  super(r, result);
  this.time = ns;
  this.period = 0;//period为0，说明为一次性任务
  this.sequenceNumber = sequencer.getAndIncrement();
}
```

在构造函数内部首先调用了父类 FutureTask 的构造函数，父类 FutureTask 的构造函数代码如下。

```
//通过适配器把runnable转换为callable
```

```
public FutureTask(Runnable runnable, V result) {
   this.callable = Executors.callable(runnable, result);
   this.state = NEW;        //设置当前任务状态为NEW
}
```

FutureTask 中的任务被转换为 Callable 类型后，被保存到了变量 this.callable 里面，并设置 FutureTask 的任务状态为 NEW。

然后在 ScheduledFutureTask 构造函数内部设置 time 为上面说的绝对时间。需要注意，这里 period 的值为 0，这说明当前任务为一次性的任务，不是定时反复执行任务。其中 long getDelay(TimeUnit unit) 方法的代码如下（该方法用来计算当前任务还有多少时间就过期了）。

```
//元素过期算法，装饰后时间-当前时间，就是即将过期剩余时间
public long getDelay(TimeUnit unit) {
  return unit.convert(time - now(), NANOSECONDS);
}
```

compareTo(Delayed other) 方法的代码如下：

```
public int compareTo(Delayed other) {
    if (other == this) // compare zero ONLY if same object
        return 0;
    if (other instanceof ScheduledFutureTask) {
        ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
        long diff = time - x.time;
        if (diff < 0)
            return -1;
        else if (diff > 0)
            return 1;
        else if (sequenceNumber < x.sequenceNumber)
            return -1;
        else
            return 1;
    }
    long d = (getDelay(TimeUnit.NANOSECONDS) -
              other.getDelay(TimeUnit.NANOSECONDS));
    return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
}
```

compareTo 的作用是加入元素到延迟队列后，在内部建立或者调整堆时会使用该元素的 compareTo 方法与队列里面其他元素进行比较，让最快要过期的元素放到队首。所以无

论什么时候向队列里面添加元素，队首的元素都是最快要过期的元素。

III. 代码（3）将任务添加到延迟队列，delayedExecute 的代码如下。

```
private void delayedExecute(RunnableScheduledFuture<?> task) {

    //(4)如果线程池关闭了，则执行线程池拒绝策略
    if (isShutdown())
        reject(task);
    else {
        //(5)添加任务到延迟队列
        super.getQueue().add(task);

        //(6)再次检查线程池状态
        if (isShutdown() &&
            !canRunInCurrentRunState(task.isPeriodic()) &&
            remove(task))
            task.cancel(false);
        else
            //(7)确保至少一个线程在处理任务
            ensurePrestart();
    }
}
```

IV. 代码（4）首先判断当前线程池是否已经关闭了，如果已经关闭则执行线程池的拒绝策略，否则执行代码（5）将任务添加到延迟队列。添加完毕后还要重新检查线程池是否被关闭了，如果已经关闭则从延迟队列里面删除刚才添加的任务，但是此时有可能线程池中的线程已经从任务队列里面移除了该任务，也就是该任务已经在执行了，所以还需要调用任务的 cancle 方法取消任务。

V. 如果代码(6)判断结果为 false，则会执行代码(7)确保至少有一个线程在处理任务，即使核心线程数 corePoolSize 被设置为 0。ensurePrestart 的代码如下。

```
void ensurePrestart() {
    int wc = workerCountOf(ctl.get());
    //增加核心线程数
    if (wc < corePoolSize)
        addWorker(null, true);
    //如果初始化corePoolSize==0，则也添加一个线程。
    else if (wc == 0)
        addWorker(null, false);
    }
```

如上代码首先获取线程池中的线程个数，如果线程个数小于核心线程数则新增一个线程，否则如果当前线程数为 0 则新增一个线程。

上面我们分析了如何向延迟队列添加任务，下面我们来看线程池里面的线程如何获取并执行任务。在前面讲解 ThreadPoolExecutor 时我们说过，具体执行任务的线程是 Worker 线程，Worker 线程调用具体任务的 run 方法来执行。由于这里的任务是 ScheduledFutureTask，所以我们下面看看 ScheduledFutureTask 的 run 方法。

```
public void run() {

    //(8)是否只执行一次
    boolean periodic = isPeriodic();

    //(9)取消任务
    if (!canRunInCurrentRunState(periodic))
        cancel(false);
    //(10)只执行一次，调用schedule方法时候
    else if (!periodic)
        ScheduledFutureTask.super.run();

    //(11)定时执行
    else if (ScheduledFutureTask.super.runAndReset()) {
        //(11.1)设置time=time+period
        setNextRunTime();

        //(11.2)重新加入该任务到delay队列
        reExecutePeriodic(outerTask);
    }
}
```

VI. 代码（8）中的 isPeriodic 的作用是判断当前任务是一次性任务还是可重复执行的任务，isPeriodic 的代码如下。

```
    public boolean isPeriodic() {
        return period != 0;
    }
```

可以看到，其内部是通过 period 的值来判断的，由于转换任务在创建 ScheduledFutureTask 时传递的 period 的值为 0，所以这里 isPeriodic 返回 false。

VII. 代码（9）判断当前任务是否应该被取消，canRunInCurrentRunState 的代码如下。

```
boolean canRunInCurrentRunState(boolean periodic) {
```

```
        return isRunningOrShutdown(periodic ?
                                   continueExistingPeriodicTasksAfterShutdown :
                                   executeExistingDelayedTasksAfterShutdown);
    }
```

这里传递的 periodic 的值为 false，所以 isRunningOrShutdown 的参数为 executeExistingDelayedTasksAfterShutdown。executeExistingDelayedTasksAfterShutdown 默认为 true，表示当其他线程调用了 shutdown 命令关闭了线程池后，当前任务还是要执行，否则如果为 false，则当前任务要被取消。

VIII. 由于 periodic 的值为 false，所以执行代码（10）调用父类 FutureTask 的 run 方法具体执行任务。FutureTask 的 run 方法的代码如下。

```
public void run() {
        //(12)
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return;

        //(13)
        try {

            Callable<V> c = callable;
            if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                    result = c.call();
                    ran = true;
                } catch (Throwable ex) {
                    result = null;
                    ran = false;
                    //(13.1)
                    setException(ex);
                }
                //(13.2)
                if (ran)
                    set(result);
            }
        } finally {
            ...
        }
```

```
    }
```

代码（12）判断如果任务状态不是NEW则直接返回，或者如果当前任务状态为NEW但是使用CAS设置当前任务的持有者为当前线程失败则直接返回。代码（13）具体调用callable的call方法执行任务。这里在调用前又判断了任务的状态是否为NEW，是为了避免在执行代码（12）后其他线程修改了任务的状态（比如取消了该任务）。

如果任务执行成功则执行代码（13.2）修改任务状态，set方法的代码如下。

```
protected void set(V v) {
    //如果当前任务的状态为NEW，则设置为COMPLETING
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = v;
        //设置当前任务的状态为NORMAL，也就是任务正常结束
        UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
        finishCompletion();
    }
}
```

如上代码首先使用CAS将当前任务的状态从NEW转换到COMPLETING。这里当有多个线程调用时只有一个线程会成功。成功的线程再通过UNSAFE.putOrderedInt设置任务的状态为正常结束状态，这里没有使用CAS是因为对于同一个任务只可能有一个线程运行到这里。在这里使用putOrderedInt比使用CAS或者putLongvolatile效率要高，并且这里的场景不要求其他线程马上对设置的状态值可见。

请思考个问题，在什么时候多个线程会同时执行CAS将当前任务的状态从NEW转换到COMPLETING？其实当同一个command被多次提交到线程池时就会存在这样的情况，因为同一个任务共享一个状态值state。

如果任务执行失败，则执行代码(13.1)。setException的代码如下，可见与set函数类似。

```
protected void setException(Throwable t) {
    //如果当前任务的状态为NEW，则设置为COMPLETING
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = t;

        //设置当前任务的状态为EXCEPTIONAL，也就是任务非正常结束
        UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL);
finishCompletion();
    }
}
```

到这里代码（10）的逻辑执行完毕，一次性任务也就执行完毕了，

下面会讲到，如果任务是可重复执行的，则不会执行代码（10）而是执行代码（11）。

9.3.2 scheduleWithFixedDelay(Runnable command,long initialDelay, long delay,TimeUnit unit) 方法

该方法的作用是，当任务执行完毕后，让其延迟固定时间后再次运行（fixed-delay 任务）。其中 initialDelay 表示提交任务后延迟多少时间开始执行任务 command，delay 表示当任务执行完毕后延长多少时间后再次运行 command 任务，unit 是 initialDelay 和 delay 的时间单位。任务会一直重复运行直到任务运行中抛出了异常，被取消了，或者关闭了线程池。scheduleWithFixedDelay 的代码如下。

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                 long initialDelay,
                                                 long delay,
                                                 TimeUnit unit) {
    //(14)参数校验
    if (command == null || unit == null)
        throw new NullPointerException();
    if (delay <= 0)
        throw new IllegalArgumentException();

    //(15)任务转换,注意这里是period=-delay<0
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
                                      null,
                                      triggerTime(initialDelay, unit),
                                      unit.toNanos(-delay));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    sft.outerTask = t;
    //(16)添加任务到队列
    delayedExecute(t);
    return t;
}
```

代码（14）进行参数校验，校验失败则抛出异常，代码（15）将 command 任务转换为 ScheduledFutureTask。这里需要注意的是，传递给 ScheduledFutureTask 的 period 变量的值为 -delay，period<0 说明该任务为可重复执行的任务。然后代码（16）添加任务到延迟队列后返回。

将任务添加到延迟队列后线程池线程会从队列里面获取任务，然后调用ScheduledFutureTask的run方法执行。由于这里period<0，所以isPeriodic返回true，所以执行代码（11）。runAndReset的代码如下。

```
protected boolean runAndReset() {
      //(17)
      if (state != NEW ||
          !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                       null, Thread.currentThread()))
          return false;

      //(18)
      boolean ran = false;
      int s = state;
      try {
          Callable<V> c = callable;
          if (c != null && s == NEW) {
              try {
                  c.call(); // don't set result
                  ran = true;
              } catch (Throwable ex) {
                  setException(ex);
              }
          }
      } finally {

        ...
  }
      return ran && s == NEW;//(19)
  }
```

该代码和FutureTask的run方法类似，只是任务正常执行完毕后不会设置任务的状态，这样做是为了让任务成为可重复执行的任务。这里多了代码（19），这段代码判断如果当前任务正常执行完毕并且任务状态为NEW则返回true，否则返回false。如果返回了true则执行代码（11.1）的setNextRunTime方法设置该任务下一次的执行时间。setNextRunTime的代码如下。

```
    private void setNextRunTime() {
          long p = period;
          if (p > 0)//fixed-rate类型任务
              time += p;
```

```
            else//fixed-delay类型任务
                time = triggerTime(-p);
        }
```

这里 p<0 说明当前任务为 fixed-delay 类型任务。然后设置 time 为当前时间加上 -p 的时间，也就是延迟 -p 时间后再次执行。

总结：本节介绍的 fixed-delay 类型的任务的执行原理为，当添加一个任务到延迟队列后，等待 initialDelay 时间，任务就会过期，过期的任务就会被从队列移除，并执行。执行完毕后，会重新设置任务的延迟时间，然后再把任务放入延迟队列，循环往复。需要注意的是，如果一个任务在执行中抛出了异常，那么这个任务就结束了，但是不影响其他任务的执行。

9.3.3 scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit) 方法

该方法相对起始时间点以固定频率调用指定的任务（fixed-rate 任务）。当把任务提交到线程池并延迟 initialDelay 时间（时间单位为 unit）后开始执行任务 command 。然后从 initialDelay+period 时间点再次执行，而后在 initialDelay + 2 * period 时间点再次执行，循环往复，直到抛出异常或者调用了任务的 cancel 方法取消了任务，或者关闭了线程池。scheduleAtFixedRate 的原理与 scheduleWithFixedDelay 类似，下面我们讲下它们之间的不同点。首先调用 scheduleAtFixedRate 的代码如下。

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit) {
    ...
    //装饰任务类，注意period=period>0，不是负的
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
                                      null,
                                      triggerTime(initialDelay, unit),
                                      unit.toNanos(period));
    ...
     return t;
}
```

在如上代码中，在将 fixed-rate 类型的任务 command 转换为 ScheduledFutureTask 时设

置 period=period，不再是 -period。

所以当前任务执行完毕后，调用 setNextRunTime 设置任务下次执行的时间时执行的是 time += p 而不再是 time = triggerTime(-p)。

总结：相对于 fixed-delay 任务来说，fixed-rate 方式执行规则为，时间为 initdelday + n*period 时启动任务，但是如果当前任务还没有执行完，下一次要执行任务的时间到了，则不会并发执行，下次要执行的任务会延迟执行，要等到当前任务执行完毕后再执行。

9.4 总结

本章讲解了 ScheduledThreadPoolExecutor 的实现原理，如图 9-2 所示，其内部使用 DelayQueue 来存放具体任务。任务分为三种，其中一次性执行任务执行完毕就结束了，fixed-delay 任务保证同一个任务在多次执行之间间隔固定时间，fixed-rate 任务保证按照固定的频率执行。任务类型使用 period 的值来区分。

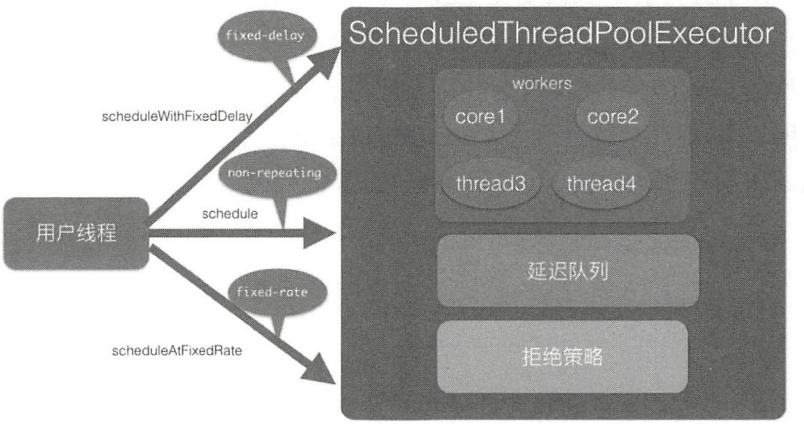

图 9-2

第10章

Java并发包中线程同步器原理剖析

10.1 CountDownLatch 原理剖析

10.1.1 案例介绍

在日常开发中经常会遇到需要在主线程中开启多个线程去并行执行任务，并且主线程需要等待所有子线程执行完毕后再进行汇总的场景。在 CountDownLatch 出现之前一般都使用线程的 join() 方法来实现这一点，但是 join 方法不够灵活，不能够满足不同场景的需要，所以 JDK 开发组提供了 CountDownLatch 这个类，我们前面介绍的例子使用 CountDownLatch 会更优雅。使用 CountDownLatch 的代码如下：

```
public class JoinCountDownLatch {

    // 创建一个CountDownLatch实例
    private static volatile CountDownLatch countDownLatch = new CountDownLatch(2);

    public static void main(String[] args) throws InterruptedException {

        Thread threadOne = new Thread(new Runnable() {

            @Override
            public void run() {

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }finally {
                countDownLatch.countDown();
            }

            System.out.println("child threadOne over!");

        }
    });

    Thread threadTwo = new Thread(new Runnable() {

        @Override
        public void run() {

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }finally {
                countDownLatch.countDown();
            }
            System.out.println("child threadTwo over!");

        }
    });

    // 启动子线程
    threadOne.start();
    threadTwo.start();

    System.out.println("wait all child thread over!");

    // 等待子线程执行完毕，返回
    countDownLatch.await();

    System.out.println("all child thread over!");

  }
}
```

输出结果如下。

```
<terminated> JoinCountDownLatch [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
wait all child thread over!
child threadOne over!
child threadTwo over!
all child thread over!
```

在如上代码中，创建了一个 CountDownLatch 实例，因为有两个子线程所以构造函数的传参为 2。主线程调用 countDownLatch.await（）方法后会被阻塞。子线程执行完毕后调用 countDownLatch.countDown() 方法让 countDownLatch 内部的计数器减 1，所有子线程执行完毕并调用 countDown（）方法后计数器会变为 0，这时候主线程的 await（）方法才会返回。

其实上面的代码还不够优雅，在项目实践中一般都避免直接操作线程，而是使用 ExecutorService 线程池来管理。使用 ExecutorService 时传递的参数是 Runable 或者 Callable 对象，这时候你没有办法直接调用这些线程的 join（）方法，这就需要选择使用 CountDownLatch 了。将上面代码修改为如下：

```
public class JoinCountDownLatch2 {
    // 创建一个CountDownLatch实例
    private static CountDownLatch countDownLatch = new CountDownLatch(2);

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        // 将线程A添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } finally {
                    countDownLatch.countDown();
                }
                System.out.println("child threadOne over!");
            }
        });
        // 将线程B添加到线程池
        executorService.submit(new Runnable() {
```

```
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
                System.out.println("child threadTwo over!");
            }
        });
        System.out.println("wait all child thread over!");
        // 等待子线程执行完毕，返回
        countDownLatch.await();
        System.out.println("all child thread over!");
        executorService.shutdown();
    }
}
```

输出结果如下。

```
<terminated> CountDownLacth2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
wait all child thread over!
child threadOne over!
child threadTwo over!
all child thread over!
```

这里总结下CountDownLatch与join方法的区别。一个区别是，调用一个子线程的join()方法后，该线程会一直被阻塞直到子线程运行完毕，而CountDownLatch则使用计数器来允许子线程运行完毕或者在运行中递减计数，也就是CountDownLatch可以在子线程运行的任何时候让await方法返回而不一定必须等到线程结束。另外，使用线程池来管理线程时一般都是直接添加Runable到线程池，这时候就没有办法再调用线程的join方法了，就是说countDownLatch相比join方法让我们对线程同步有更灵活的控制。

10.1.2　实现原理探究

从CountDownLatch的名字就可以猜测其内部应该有个计数器，并且这个计数器是递减的。下面就通过源码看看JDK开发组在何时初始化计数器，在何时递减计数器，当计数器变为0时做了什么操作，多个线程是如何通过计时器值实现同步的。为了一览CountDownLatch的内部结构，我们先看它的类图（如图10-1所示）。

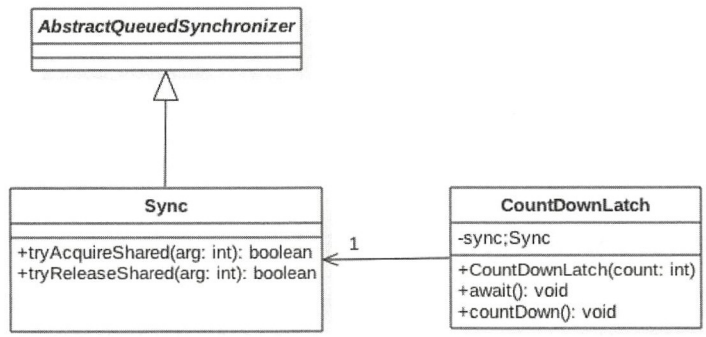

图 10-1

从类图可以看出，CountDownLatch 是使用 AQS 实现的。通过下面的构造函数，你会发现，实际上是把计数器的值赋给了 AQS 的状态变量 state，也就是这里使用 AQS 的状态值来表示计数器值。

```
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    this.sync = new Sync(count);
}

Sync(int count) {
    setState(count);
}
```

下面我们来研究 CountDownLatch 中的几个重要的方法，看它们是如何调用 AQS 来实现功能的。

1. void await() 方法

当线程调用 CountDownLatch 对象的 await 方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回：当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计数器的值为 0 时；其他线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程就会抛出 InterruptedException 异常，然后返回。

下面看下在 await() 方法内部是如何调用 AQS 的方法的。

```
//CountDownLatch的await ( ) 方法
public void await() throws InterruptedException {
   sync.acquireSharedInterruptibly(1);
```

```
}
```

从以上代码可以看到，await() 方法委托 sync 调用了 AQS 的 acquireSharedInterruptibly 方法，后者的代码如下：

```
    //AQS获取共享资源时可被中断的方法
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        //如果线程被中断则抛出异常
        if (Thread.interrupted())
            throw new InterruptedException();
        //查看当前计数器值是否为0，为0则直接返回，否则进入AQS的队列等待
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }

//sync类实现的AQS的接口
protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
```

由如上代码可知，该方法的特点是线程获取资源时可以被中断，并且获取的资源是共享资源。acquireSharedInterruptibly 首先判断当前线程是否已被中断，若是则抛出异常，否则调用 sync 实现的 tryAcquireShared 方法查看当前状态值（计数器值）是否为 0，是则当前线程的 await() 方法直接返回，否则调用 AQS 的 doAcquireSharedInterruptibly 方法让当前线程阻塞。另外可以看到，这里 tryAcquireShared 传递的 arg 参数没有被用到，调用 tryAcquireShared 的方法仅仅是为了检查当前状态值是不是为 0，并没有调用 CAS 让当前状态值减 1。

2. boolean await(long timeout, TimeUnit unit) 方法

当线程调用了 CountDownLatch 对象的该方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回：当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计数器值为 0 时，这时候会返回 true；设置的 timeout 时间到了，因为超时而返回 false；其他线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程会抛出 InterruptedException 异常，然后返回。

```
public boolean await(long timeout, TimeUnit unit)
        throws InterruptedException {
```

```
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }
```

3. void countDown() 方法

线程调用该方法后，计数器的值递减，递减后如果计数器值为 0 则唤醒所有因调用 await 方法而被阻塞的线程，否则什么都不做。下面看下 countDown() 方法是如何调用 AQS 的方法的。

```
    //CountDownLatch的countDown ( ) 方法
    public void countDown() {
       //委托sync调用AQS的方法
        sync.releaseShared(1);
    }
```

由如上代码可知，CountDownLatch 的 countDown（）方法委托 sync 调用了 AQS 的 releaseShared 方法，后者的代码如下。

```
   //AQS的方法
    public final boolean releaseShared(int arg) {
        //调用sync实现的tryReleaseShared
        if (tryReleaseShared(arg)) {
            //AQS的释放资源方法
            doReleaseShared();
            return true;
        }
        return false;
    }
```

在如上代码中，releaseShared 首先调用了 sync 实现的 AQS 的 tryReleaseShared 方法，其代码如下。

```
//sync的方法
protected boolean tryReleaseShared(int releases) {
  //循环进行CAS，直到当前线程成功完成CAS使计数器值（状态值state）减1并更新到state
  for (;;) {
      int c = getState();

      //如果当前状态值为0则直接返回（1）
      if (c == 0)
          return false;

      //使用CAS让计数器值减1（2）
```