```
            int nextc = c-1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
```

如上代码首先获取当前状态值（计数器值）。代码（1）判断如果当前状态值为 0 则直接返回 false，从而 countDown（）方法直接返回；否则执行代码（2）使用 CAS 将计数器值减 1，CAS 失败则循环重试，否则如果当前计数器值为 0 则返回 true，返回 true 说明是最后一个线程调用的 countdown 方法，那么该线程除了让计数器值减 1 外，还需要唤醒因调用 CountDownLatch 的 await 方法而被阻塞的线程，具体是调用 AQS 的 doReleaseShared 方法来激活阻塞的线程。这里代码（1）貌似是多余的，其实不然，之所以添加代码（1）是为了防止当计数器值为 0 后，其他线程又调用了 countDown 方法，如果没有代码（1），状态值就可能会变成负数。

4. long getCount() 方法

获取当前计数器的值，也就是 AQS 的 state 的值，一般在测试时使用该方法。下面看下代码。

```
public long getCount() {
    return sync.getCount();
}

int getCount() {
 return getState();
}
```

由如上代码可知，在其内部还是调用了 AQS 的 getState 方法来获取 state 的值（计数器当前值）。

10.1.3 小结

本节首先介绍了 CountDownLatch 的使用，相比使用 join 方法来实现线程间同步，前者更具有灵活性和方便性。另外还介绍了 CountDownLatch 的原理，CountDownLatch 是使用 AQS 实现的。使用 AQS 的状态变量来存放计数器的值。首先在初始化 CountDownLatch 时设置状态值（计数器值），当多个线程调用 countdown 方法时实际是原子性递减 AQS 的状态值。当线程调用 await 方法后当前线程会被放入 AQS 的阻塞队列等

待计数器为 0 再返回。其他线程调用 countdown 方法让计数器值递减 1，当计数器值变为 0 时，当前线程还要调用 AQS 的 doReleaseShared 方法来激活由于调用 await() 方法而被阻塞的线程。

10.2 回环屏障 CyclicBarrier 原理探究

上节介绍的 CountDownLatch 在解决多个线程同步方面相对于调用线程的 join 方法已经有了不少优化，但是 CountDownLatch 的计数器是一次性的，也就是等到计数器值变为 0 后，再调用 CountDownLatch 的 await 和 countdown 方法都会立刻返回，这就起不到线程同步的效果了。所以为了满足计数器可以重置的需要，JDK 开发组提供了 CyclicBarrier 类，并且 CyclicBarrier 类的功能并不限于 CountDownLatch 的功能。从字面意思理解，CyclicBarrier 是回环屏障的意思，它可以让一组线程全部达到一个状态后再全部同时执行。这里之所以叫作回环是因为当所有等待线程执行完毕，并重置 CyclicBarrier 的状态后它可以被重用。之所以叫作屏障是因为线程调用 await 方法后就会被阻塞，这个阻塞点就称为屏障点，等所有线程都调用了 await 方法后，线程们就会冲破屏障，继续向下运行。

10.2.1 案例介绍

在介绍原理前先介绍几个实例以便加深理解。在下面的例子中，我们要实现的是，使用两个线程去执行一个被分解的任务 A，当两个线程把自己的任务都执行完毕后再对它们的结果进行汇总处理。

```
public class CycleBarrierTest1 {

    // 创建一个CyclicBarrier实例,添加一个所有子线程全部到达屏障后执行的任务
    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(2, new Runnable()
{
        public void run() {
            System.out.println(Thread.currentThread() + " task1 merge result");
        }
    });

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

        //创建一个线程个数固定为2的线程池
        ExecutorService executorService = Executors.newFixedThreadPool(2);
```

```
        // 将线程A添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() + " task1-1");

                    System.out.println(Thread.currentThread() + " enter in
                       barrier");
                    cyclicBarrier.await();
                    System.out.println(Thread.currentThread() + " enter out
                      barrier");

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

        // 将线程B添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {
                    System.out.println(Thread.currentThread() + " task1-2");

                    System.out.println(Thread.currentThread() + " enter in
                       barrier");
                    cyclicBarrier.await();
                    System.out.println(Thread.currentThread() + " enter out
                     barrier");

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

        // 关闭线程池
        executorService.shutdown();
    }
}
```

输出结果如下。

```
<terminated> CycleBarrierTest1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] task1-1
Thread[pool-1-thread-1,5,main] enter in barrier
Thread[pool-1-thread-2,5,main] task1-2
Thread[pool-1-thread-2,5,main] enter in barrier
Thread[pool-1-thread-2,5,main] task1 merge result
Thread[pool-1-thread-2,5,main] enter out barrier
Thread[pool-1-thread-1,5,main] enter out barrier
```

如上代码创建了一个 CyclicBarrier 对象，其第一个参数为计数器初始值，第二个参数 Runable 是当计数器值为 0 时需要执行的任务。在 main 函数里面首先创建了一个大小为 2 的线程池，然后添加两个子任务到线程池，每个子任务在执行完自己的逻辑后会调用 await 方法。一开始计数器值为 2，当第一个线程调用 await 方法时，计数器值会递减为 1。由于此时计数器值不为 0，所以当前线程就到了屏障点而被阻塞。然后第二个线程调用 await 时，会进入屏障，计数器值也会递减，现在计数器值为 0，这时就会去执行 CyclicBarrier 构造函数中的任务，执行完毕后退出屏障点，并且唤醒被阻塞的第二个线程，这时候第一个线程也会退出屏障点继续向下运行。

上面的例子说明了多个线程之间是相互等待的，假如计数器值为 N，那么随后调用 await 方法的 N–1 个线程都会因为到达屏障点而被阻塞，当第 N 个线程调用 await 后，计数器值为 0 了，这时候第 N 个线程才会发出通知唤醒前面的 N–1 个线程。也就是当全部线程都到达屏障点时才能一块继续向下执行。对于这个例子来说，使用 CountDownLatch 也可以得到类似的输出结果。下面再举个例子来说明 CyclicBarrier 的可复用性。

假设一个任务由阶段 1、阶段 2 和阶段 3 组成，每个线程要串行地执行阶段 1、阶段 2 和阶段 3，当多个线程执行该任务时，必须要保证所有线程的阶段 1 全部完成后才能进入阶段 2 执行，当所有线程的阶段 2 全部完成后才能进入阶段 3 执行。下面使用 CyclicBarrier 来完成这个需求。

```
public class CycleBarrierTest2 {

    // 创建一个CyclicBarrier实例
    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(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 {

                    System.out.println(Thread.currentThread() +  " step1");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +  " step2");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +  " step3");

                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });

        // 将线程B添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {
                    System.out.println(Thread.currentThread() +  " step1");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +  " step2");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +  " step3");


                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });

        //关闭线程池
        executorService.shutdown();
    }
}
```

输出结果如下。

```
<terminated> CycleBarrierTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] step1
Thread[pool-1-thread-2,5,main] step1
Thread[pool-1-thread-2,5,main] step2
Thread[pool-1-thread-1,5,main] step2
Thread[pool-1-thread-1,5,main] step3
Thread[pool-1-thread-2,5,main] step3
```

在如上代码中，每个子线程在执行完阶段 1 后都调用了 await 方法，等到所有线程都到达屏障点后才会一块往下执行，这就保证了所有线程都完成了阶段 1 后才会开始执行阶段 2。然后在阶段 2 后面调用了 await 方法，这保证了所有线程都完成了阶段 2 后，才能开始阶段 3 的执行。这个功能使用单个 CountDownLatch 是无法完成的。

10.2.2 实现原理探究

为了能够一览 CyclicBarrier 的架构设计，下面先看下 CyclicBarrier 的类图结构，如图 10-2 所示。

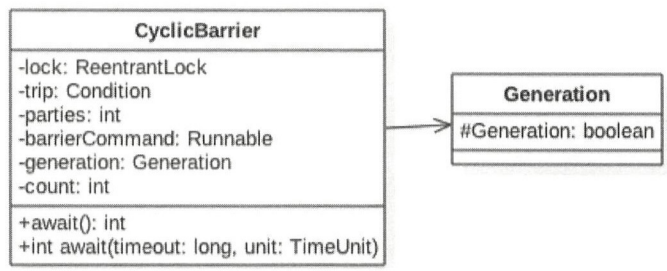

图 10-2

由以上类图可知，CyclicBarrier 基于独占锁实现，本质底层还是基于 AQS 的。parties 用来记录线程个数，这里表示多少线程调用 await 后，所有线程才会冲破屏障继续往下运行。而 count 一开始等于 parties，每当有线程调用 await 方法就递减 1，当 count 为 0 时就表示所有线程都到了屏障点。你可能会疑惑，为何维护 parties 和 count 两个变量，只使用 count 不就可以了？别忘了 CycleBarier 是可以被复用的，使用两个变量的原因是，parties 始终用来记录总的线程个数，当 count 计数器值变为 0 后，会将 parties 的值赋给 count，从而进行复用。这两个变量是在构造 CyclicBarrier 对象时传递的，如下所示。

```
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
```

```
        this.parties = parties;
        this.count = parties;
        this.barrierCommand = barrierAction;
    }
```

还有一个变量 barrierCommand 也通过构造函数传递，这是一个任务，这个任务的执行时机是当所有线程都到达屏障点后。使用 lock 首先保证了更新计数器 count 的原子性。另外使用 lock 的条件变量 trip 支持线程间使用 await 和 signal 操作进行同步。

最后，在变量 generation 内部有一个变量 broken，其用来记录当前屏障是否被打破。注意，这里的 broken 并没有被声明为 volatile 的，因为是在锁内使用变量，所以不需要声明。

```
    private static class Generation {
        boolean broken = false;
    }
```

下面来看 CyclicBarrier 中的几个重要的方法。

1. int await() 方法

当前线程调用 CyclicBarrier 的该方法时会被阻塞，直到满足下面条件之一才会返回：parties 个线程都调用了 await() 方法，也就是线程都到了屏障点；其他线程调用了当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常而返回；与当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时，会抛出 BrokenBarrierException 异常，然后返回。

由如下代码可知，在内部调用了 dowait 方法。第一个参数为 false 则说明不设置超时时间，这时候第二个参数没有意义。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

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

当前线程调用 CyclicBarrier 的该方法时会被阻塞，直到满足下面条件之一才会返回：parties 个线程都调用了 await() 方法，也就是线程都到了屏障点，这时候返回 true；设置的

超时时间到了后返回 false；其他线程调用当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常然后返回；与当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时，会抛出 BrokenBarrierException 异常，然后返回。

由如下代码可知，在内部调用了 dowait 方法。第一个参数为 true 则说明设置了超时时间，这时候第二个参数是超时时间。

```
public int await(long timeout, TimeUnit unit)
    throws InterruptedException,
           BrokenBarrierException,
           TimeoutException {
    return dowait(true, unit.toNanos(timeout));
}
```

3. int dowait(boolean timed, long nanos) 方法

该方法实现了 CyclicBarrier 的核心功能，其代码如下。

```
private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
             ...

            //(1)如果index==0则说明所有线程都到了屏障点，此时执行初始化时传递的任务
            int index = --count;
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    //(2)执行任务
                    if (command != null)
                        command.run();
                    ranAction = true;
                    //(3)激活其他因调用await方法而被阻塞的线程，并重置CyclicBarrier
                    nextGeneration();
                    //返回
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
```

```
                }
            }

            // (4)如果index!=0
            for (;;) {
                try {
                     //(5)没有设置超时时间,
                     if (!timed)
                        trip.await();
                    //(6)设置了超时时间
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    ...
                }
                    ...
            }
        } finally {
            lock.unlock();
        }
    }

    private void nextGeneration() {
        //(7)唤醒条件队列里面阻塞线程
        trip.signalAll();
        //(8)重置CyclicBarrier
        count = parties;
        generation = new Generation();
    }
```

以上是 dowait 方法的主干代码。当一个线程调用了 dowait 方法后，首先会获取独占锁 lock，如果创建 CycleBarrier 时传递的参数为 10，那么后面 9 个调用线程会被阻塞。然后当前获取到锁的线程会对计数器 count 进行递减操作，递减后 count=index=9，因为 index!=0 所以当前线程会执行代码（4）。如果当前线程调用的是无参数的 await() 方法，则这里 timed=false，所以当前线程会被放入条件变量 trip 的条件阻塞队列，当前线程会被挂起并释放获取的 lock 锁。如果调用的是有参数的 await 方法则 timed=true，然后当前线程也会被放入条件变量的条件队列并释放锁资源，不同的是当前线程会在指定时间超时后自动被激活。

当第一个获取锁的线程由于被阻塞释放锁后，被阻塞的 9 个线程中有一个会竞争到

lock 锁，然后执行与第一个线程同样的操作，直到最后一个线程获取到 lock 锁，此时已经有 9 个线程被放入了条件变量 trip 的条件队列里面。最后 count=index 等于 0，所以执行代码（2），如果创建 CyclicBarrier 时传递了任务，则在其他线程被唤醒前先执行任务，任务执行完毕后再执行代码（3），唤醒其他 9 个线程，并重置 CyclicBarrier，然后这 10 个线程就可以继续向下运行了。

10.2.3 小结

本节首先通过案例说明了 CycleBarrier 与 CountDownLatch 的不同在于，前者是可以复用的，并且前者特别适合分段任务有序执行的场景。然后分析了 CycleBarrier，其通过独占锁 ReentrantLock 实现计数器原子性更新，并使用条件变量队列来实现线程同步。

10.3 信号量 Semaphore 原理探究

Semaphore 信号量也是 Java 中的一个同步器，与 CountDownLatch 和 CycleBarrier 不同的是，它内部的计数器是递增的，并且在一开始初始化 Semaphore 时可以指定一个初始值，但是并不需要知道需要同步的线程个数，而是在需要同步的地方调用 acquire 方法时指定需要同步的线程个数。

10.3.1 案例介绍

同样下面的例子也是在主线程中开启两个子线程让它们执行，等所有子线程执行完毕后主线程再继续向下运行。

```
public class SemaphoreTest {

    // 创建一个Semaphore实例
    private static Semaphore semaphore = new Semaphore(0);

    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 {
```

```
                System.out.println(Thread.currentThread() +  " over");
                semaphore.release();

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

    // 将线程B添加到线程池
    executorService.submit(new Runnable() {
        public void run() {
            try {

                System.out.println(Thread.currentThread() +  " over");
                semaphore.release();

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

    // 等待子线程执行完毕，返回
    semaphore.acquire(2);
    System.out.println("all child thread over!");

    //关闭线程池
    executorService.shutdown();
  }
}
```

输出结果如下。

```
<terminated> SemaphoreTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] over
Thread[pool-1-thread-2,5,main] over
all child thread over!
```

如上代码首先创建了一个信号量实例，构造函数的入参为0，说明当前信号量计数器的值为0。然后main函数向线程池添加两个线程任务，在每个线程内部调用信号量的release方法，这相当于让计数器值递增1。最后在main线程里面调用信号量的acquire方

法，传参为 2 说明调用 acquire 方法的线程会一直阻塞，直到信号量的计数变为 2 才会返回。看到这里也就明白了，如果构造 Semaphore 时传递的参数为 N，并在 M 个线程中调用了该信号量的 release 方法，那么在调用 acquire 使 M 个线程同步时传递的参数应该是 $M+N$。

下面举个例子来模拟 CyclicBarrier 复用的功能，代码如下。

```
public class SemaphoreTest2 {

    // 创建一个Semaphore实例
    private static volatile Semaphore semaphore = new Semaphore(0);

    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 {

                    System.out.println(Thread.currentThread() + " A  task over");
                    semaphore.release();

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

        // 将线程B添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() + " A  task over");
                    semaphore.release();

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
```

```
        // (1)等待子线程执行任务A完毕，返回
        semaphore.acquire(2);

        // 将线程c添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() + " B  task over");
                    semaphore.release();

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

        // 将线程d添加到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() + " B  task over");
                    semaphore.release();

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

        // (2)等待子线程执行B完毕，返回
        semaphore.acquire(2);

        System.out.println("task is over");

        // 关闭线程池
        executorService.shutdown();
    }
}
```

输出结果为

```
<terminated> SemaphoreTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main]sub_A  task over
Thread[pool-1-thread-2,5,main] sub_A  task over
task A is over
Thread[pool-1-thread-1,5,main] sub_B  task over
Thread[pool-1-thread-2,5,main]sub_B  task over
task B is over
```

如上代码首先将线程 A 和线程 B 加入到线程池。主线程执行代码（1）后被阻塞。线程 A 和线程 B 调用 release 方法后信号量的值变为了 2，这时候主线程的 aquire 方法会在获取到 2 个信号量后返回（返回后当前信号量值为 0）。然后主线程添加线程 C 和线程 D 到线程池，之后主线程执行代码（2）后被阻塞（因为主线程要获取 2 个信号量，而当前信号量个数为 0）。当线程 C 和线程 D 执行完 release 方法后，主线程才返回。从本例子可以看出，Semaphore 在某种程度上实现了 CyclicBarrier 的复用功能。

10.3.2 实现原理探究

为了能够一览 Semaphore 的内部结构，首先看下 Semaphore 的类图，如图 10-3 所示。

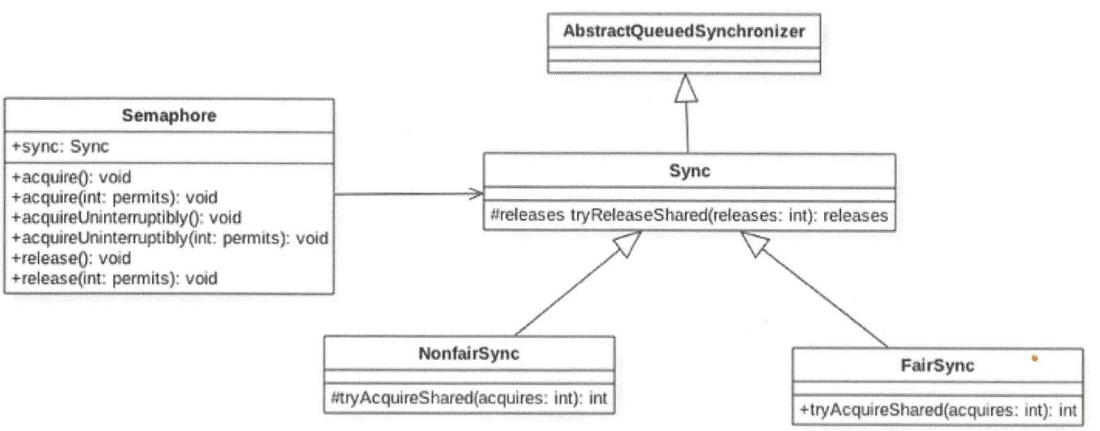

图 10-3

由该类图可知，Semaphore 还是使用 AQS 实现的。Sync 只是对 AQS 的一个修饰，并且 Sync 有两个实现类，用来指定获取信号量时是否采用公平策略。例如，下面的代码在创建 Semaphore 时会使用一个变量指定是否使用公平策略。

```
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new
    NonfairSync(permits);
}

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

在如上代码中，Semaphore 默认采用非公平策略，如果需要使用公平策略则可以使用带两个参数的构造函数来构造 Semaphore 对象。另外，如 CountDownLatch 构造函数传递的初始化信号量个数 permits 被赋给了 AQS 的 state 状态变量一样，这里 AQS 的 state 值也表示当前持有的信号量个数。

下面来看 Semaphore 实现的主要方法。

1. void acquire() 方法

当前线程调用该方法的目的是希望获取一个信号量资源。如果当前信号量个数大于 0，则当前信号量的计数会减 1，然后该方法直接返回。否则如果当前信号量个数等于 0，则当前线程会被放入 AQS 的阻塞队列。当其他线程调用了当前线程的 interrupt（）方法中断了当前线程时，则当前线程会抛出 InterruptedException 异常返回。下面看下代码实现。

```
public void acquire() throws InterruptedException {
    //传递参数为1，说明要获取1个信号量资源
    sync.acquireSharedInterruptibly(1);
}


public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {

    //（1）如果线程被中断，则抛出中断异常
    if (Thread.interrupted())
        throw new InterruptedException();

    //（2）否则调用Sync子类方法尝试获取，这里根据构造函数确定使用公平策略
    if (tryAcquireShared(arg) < 0)
```

```
            //如果获取失败则放入阻塞队列。然后再次尝试，如果失败则调用park方法挂起当前线程
        doAcquireSharedInterruptibly(arg);
    }
```

由如上代码可知，acquire() 在内部调用了 Sync 的 acquireSharedInterruptibly 方法，后者会对中断进行响应（如果当前线程被中断，则抛出中断异常）。尝试获取信号量资源的 AQS 的方法 tryAcquireShared 是由 Sync 的子类实现的，所以这里分别从两方面来讨论。先讨论非公平策略 NonfairSync 类的 tryAcquireShared 方法，代码如下。

```
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}

final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
     //获取当前信号量值
     int available = getState();
     //计算当前剩余值
     int remaining = available - acquires;
     //如果当前剩余值小于0或者CAS设置成功则返回
     if (remaining < 0 ||
         compareAndSetState(available, remaining))
         return remaining;
    }
}
```

如上代码先获取当前信号量值（available），然后减去需要获取的值（acquires），得到剩余的信号量个数（remaining），如果剩余值小于 0 则说明当前信号量个数满足不了需求，那么直接返回负数，这时当前线程会被放入 AQS 的阻塞队列而被挂起。如果剩余值大于 0，则使用 CAS 操作设置当前信号量值为剩余值，然后返回剩余值。

另外，由于 NonFairSync 是非公平获取的，也就是说先调用 aquire 方法获取信号量的线程不一定比后来者先获取到信号量。考虑下面场景，如果线程 A 先调用了 aquire（）方法获取信号量，但是当前信号量个数为 0，那么线程 A 会被放入 AQS 的阻塞队列。过一段时间后线程 C 调用了 release（）方法释放了一个信号量，如果当前没有其他线程获取信号量，那么线程 A 就会被激活，然后获取该信号量，但是假如线程 C 释放信号量后，线程 C 调用了 aquire 方法，那么线程 C 就会和线程 A 去竞争这个信号量资源。如果采用非

公平策略，由 nonfairTryAcquireShared 的代码可知，线程 C 完全可以在线程 A 被激活前，或者激活后先于线程 A 获取到该信号量，也就是在这种模式下阻塞线程和当前请求的线程是竞争关系，而不遵循先来先得的策略。下面看公平性的 FairSync 类是如何保证公平性的。

```
protected int tryAcquireShared(int acquires) {
    for (;;) {
        if (hasQueuedPredecessors())
            return -1;
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

可见公平性还是靠 hasQueuedPredecessors 这个函数来保证的。前面章节讲过，公平策略是看当前线程节点的前驱节点是否也在等待获取该资源，如果是则自己放弃获取的权限，然后当前线程会被放入 AQS 阻塞队列，否则就去获取。

2. void acquire(int permits) 方法

该方法与 acquire() 方法不同，后者只需要获取一个信号量值，而前者则获取 permits 个。

```
public void acquire(int permits) throws InterruptedException {
    if (permits < 0) throw new IllegalArgumentException();
    sync.acquireSharedInterruptibly(permits);
}
```

3. void acquireUninterruptibly() 方法

该方法与 acquire() 类似，不同之处在于该方法对中断不响应，也就是当当前线程调用了 acquireUninterruptibly 获取资源时（包含被阻塞后），其他线程调用了当前线程的 interrupt（）方法设置了当前线程的中断标志，此时当前线程并不会抛出 InterruptedException 异常而返回。

```
public void acquireUninterruptibly() {
    sync.acquireShared(1);
}
```

4. void acquireUninterruptibly(int permits) 方法

该方法与 acquire(int permits) 方法的不同之处在于，该方法对中断不响应。

```
public void acquireUninterruptibly(int permits) {
    if (permits < 0) throw new IllegalArgumentException();
    sync.acquireShared(permits);
}
```

5. void release() 方法

该方法的作用是把当前 Semaphore 对象的信号量值增加 1，如果当前有线程因为调用 aquire 方法被阻塞而被放入了 AQS 的阻塞队列，则会根据公平策略选择一个信号量个数能被满足的线程进行激活，激活的线程会尝试获取刚增加的信号量，下面看代码实现。

```
public void release() {
    //(1)arg=1
    sync.releaseShared(1);
}

public final boolean releaseShared(int arg) {

    //(2)尝试释放资源
    if (tryReleaseShared(arg)) {

        //(3)资源释放成功则调用park方法唤醒AQS队列里面最先挂起的线程
        doReleaseShared();
        return true;
    }
    return false;
}

protected final boolean tryReleaseShared(int releases) {
    for (;;) {

        //(4)获取当前信号量值
        int current = getState();

        //(5)将当前信号量值增加releases，这里为增加1
        int next = current + releases;
        if (next < current) // 移除处理
            throw new Error("Maximum permit count exceeded");
```

```
            //(6)使用CAS保证更新信号量值的原子性
            if (compareAndSetState(current, next))
                return true;
        }
    }
```

由代码 release()->sync.releaseShared(1) 可知，release 方法每次只会对信号量值增加 1，tryReleaseShared 方法是无限循环，使用 CAS 保证了 release 方法对信号量递增 1 的原子性操作。tryReleaseShared 方法增加信号量值成功后会执行代码（3），即调用 AQS 的方法来激活因为调用 aquire 方法而被阻塞的线程。

6. void release(int permits) 方法

该方法与不带参数的 release 方法的不同之处在于，前者每次调用会在信号量值原来的基础上增加 permits，而后者每次增加 1。

```
public void release(int permits) {
    if (permits < 0) throw new IllegalArgumentException();
    sync.releaseShared(permits);
}
```

另外可以看到，这里的 sync.releaseShared 是共享方法，这说明该信号量是线程共享的，信号量没有和固定线程绑定，多个线程可以同时使用 CAS 去更新信号量的值而不会被阻塞。

10.3.3 小结

本节首先通过案例介绍了 Semaphore 的使用方法，Semaphore 完全可以达到 CountDownLatch 的效果，但是 Semaphore 的计数器是不可以自动重置的，不过通过变相地改变 aquire 方法的参数还是可以实现 CycleBarrier 的功能的。然后介绍了 Semaphore 的源码实现，Semaphore 也是使用 AQS 实现的，并且获取信号量时有公平策略和非公平策略之分。

10.4 总结

本章介绍了并发包中关于线程协作的一些重要类。首先 CountDownLatch 通过计数器提供了更灵活的控制，只要检测到计数器值为 0，就可以往下执行，这相比使用 join 必

须等待线程执行完毕后主线程才会继续向下运行更灵活。另外，CyclicBarrier 也可以达到 CountDownLatch 的效果，但是后者在计数器值变为 0 后，就不能再被复用，而前者则可以使用 reset 方法重置后复用，前者对同一个算法但是输入参数不同的类似场景比较适用。而 Semaphore 采用了信号量递增的策略，一开始并不需要关心同步的线程个数，等调用 aquire 方法时再指定需要同步的个数，并且提供了获取信号量的公平性策略。使用本章介绍的类会大大减少你在 Java 中使用 wait、notify 等来实现线程同步的代码量，在日常开发中当需要进行线程同步时使用这些同步类会节省很多代码并且可以保证正确性。

第三部分

Java并发编程实践篇

在高级篇讲解了 Java 中并发组件的原理实现，在这一篇我们要进行实践，只知道原理是不行的，还应该知道怎么在业务中使用。下面我们就来看看如何使用这些并发组件，以及进行并发编程时常会遇到哪些问题。

第11章

并发编程实践

11.1 ArrayBlockingQueue 的使用

这一节我们讲解 logback 异步日志打印中 ArrayBlockingQueue 的使用。

11.1.1 异步日志打印模型概述

在高并发、高流量并且响应时间要求比较小的系统中同步打印日志已经满足不了需求了，这是因为打印日志本身是需要写磁盘的，写磁盘的操作会暂时阻塞调用打印日志的业务线程，这会造成调用线程的 rt 增加。如图 11-1 所示为同步日志打印模型。

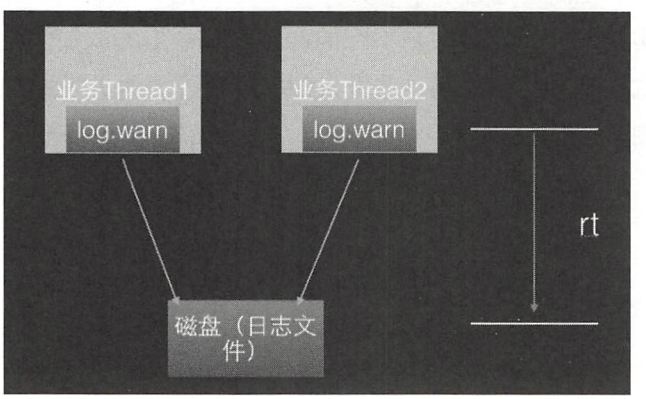

图 11-1

同步日志打印模型的缺点是将日志写入磁盘的操作是业务线程同步调用完成的，那么是否可以让业务线程把要打印的日志任务放入一个队列后直接返回，然后使用一个线程专

门负责从队列中获取日志任务并将其写入磁盘呢？这样的话，业务线程打印日志的耗时就仅仅是把日志任务放入队列的耗时了，其实这就是 logback 提供的异步日志打印模型要做的事，具体如图 11-2 所示。

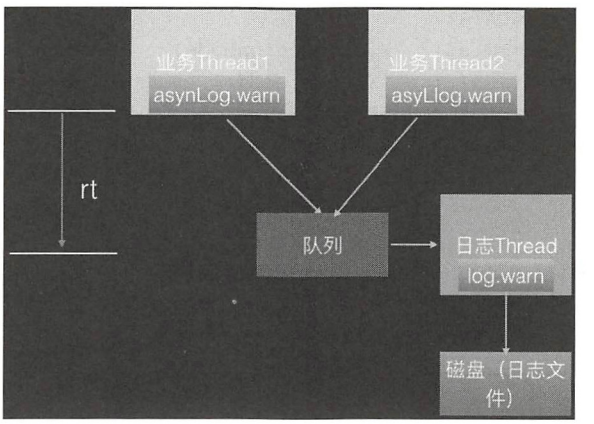

图 11-2

由图 11-2 可知，其实 logback 的异步日志模型是一个多生产者 - 单消费者模型，其通过使用队列把同步日志打印转换为了异步，业务线程只需要通过调用异步 appender 把日志任务放入日志队列，而日志线程则负责使用同步的 appender 进行具体的日志打印。日志打印线程只需要负责生产日志并将其放入队列，而不需要关心消费线程何时把日志具体写入磁盘。

11.1.2　异步日志与具体实现

1. 异步日志

一般配置同步日志打印时会在 logback 的 xml 文件里面配置如下内容。

```
//(1)配置同步日志打印appender
  <appender name="PROJECT" class="ch.qos.logback.core.FileAppender">
      <file>project.log</file>
      <encoding>UTF-8</encoding>
      <append>true</append>

      <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
          <!-- daily rollover -->
```

```
            <fileNamePattern>project.log.%d{yyyy-MM-dd}</fileNamePattern>
            <!-- keep 7 days' worth of history -->
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
%X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer},
ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n
            ]]></pattern>
        </layout>
    </appender>
  //(2) 设置logger
    <logger name="PROJECT_LOGGER" additivity="false">
        <level value="WARN" />
        <appender-ref ref="PROJECT" />
    </logger>
```

然后以如下方式使用。

```
/**
 * Hello world!
 *
 */
public class App
{
    //根据日志logger名称获取具体日志打印logger
    private static Logger logger = LoggerFactory.getLogger("PROJECT_LOGGER");

    public static void main( String[] args )
    {
        logger.warn( "Hello World!" );
        logger.warn("a {},b {}","hello","workd");
    }
}
```

要把同步日志打印改为异步则需要修改 logback 的 xml 配置文件为如下所示。

```
    <appender name="PROJECT" class="ch.qos.logback.core.FileAppender">
        <file>project.log</file>
        <encoding>UTF-8</encoding>
        <append>true</append>

        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- daily rollover -->
            <fileNamePattern>project.log.%d{yyyy-MM-dd}</fileNamePattern>
            <!-- keep 7 days' worth of history -->
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <layout class="ch.qos.logback.classic.PatternLayout">
```

```
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
%X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer},
ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n
            ]]></pattern>
        </layout>
    </appender>

    <appender name="asyncProject" class="ch.qos.logback.classic.AsyncAppender">
        <discardingThreshold>0</discardingThreshold>
        <queueSize>1024</queueSize>
        <neverBlock>true</neverBlock>
        <appender-ref ref="PROJECT" />
    </appender>
     <logger name="PROJECT_LOGGER" additivity="false">
        <level value="WARN" />
        <appender-ref ref="asyncProject" />
    </logger>
```

由以上代码可以看出，AsyncAppender 是实现异步日志的关键，下一节主要讲它的内部实现。

2. 异步日志实现原理

本文使用的 logback-classic 的版本为 1.0.13。 我们首先从 AsyncAppender 的类图结构来认识下 AsyncAppender 的组件构成，如图 11-3 所示。

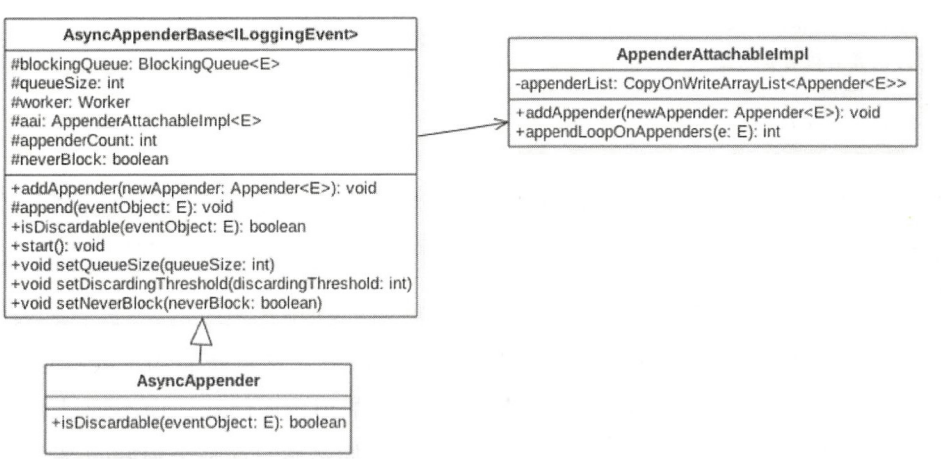

图 11-3

由图 11-3 可知，AsyncAppender 继承自 AsyncAppenderBase，其中后者具体实现了异步日志模型的主要功能，前者只是重写了其中的一些方法。由该图可知，logback 中的异步日志队列是一个阻塞队列，其实就是有界阻塞队列 ArrayBlockingQueue，其中 queueSize 表示有界队列的元素个数，默认为 256 个。

worker 是个线程，也就是异步日志打印模型中的单消费者线程。aai 是一个 appender 的装饰器，里面存放同步日志的 appender，其中 appenderCount 记录 aai 里面附加的同步 appender 的个数。neverBlock 用来指示当日志队列满时是否阻塞打印日志的线程。discardingThreshold 是一个阈值，当日志队列里面的空闲元素个数小于该值时，新来的某些级别的日志会被直接丢弃，下面会具体讲。

首先我们来看何时创建日志队列，以及何时启动消费线程，这需要看 AsyncAppenderBase 的 start 方法。该方法在解析完配置 AsyncAppenderBase 的 xml 的节点元素后被调用。

```
public void start() {
  ...
  //(1)日志队列为有界阻塞队列
  blockingQueue = new ArrayBlockingQueue<E>(queueSize);
  //(2)如果没设置discardingThreshold则设置为队列大小的1/5
  if (discardingThreshold == UNDEFINED)
    discardingThreshold = queueSize / 5;
  //(3)设置消费线程为守护线程，并设置日志名称
  worker.setDaemon(true);
  worker.setName("AsyncAppender-Worker-" + worker.getName());
  //(4)设置启动消费线程
  super.start();
  worker.start();
}
```

由以上代码可知，logback 使用的是有界队列 ArrayBlockingQueue，之所以使用有界队列是考虑内存溢出问题。在高并发下写日志的 QPS 会很高，如果设置为无界队列，队列本身会占用很大的内存，很可能会造成 OOM。

这里消费日志队列的 worker 线程被设置为守护线程，这意味着当主线程运行结束并且当前没有用户线程时，该 worker 线程会随着 JVM 的退出而终止，而不管日志队列里面是否还有日志任务未被处理。另外，这里设置了线程的名称，这是个很好的习惯，因为在查找问题时会很有帮助，根据线程名字就可以定位线程。

既然是有界队列，那么肯定需要考虑队列满的问题，是丢弃老的日志任务，还是阻塞日志打印线程直到队列有空余元素呢？要回答这个问题，我们需要看看具体进行日志打印的 AsyncAppenderBase 的 append 方法。

```
protected void append(E eventObject) {
   //(5)调用AsyncAppender重写的isDiscardable方法
   if (isQueueBelowDiscardingThreshold() && isDiscardable(eventObject)) {
       return;
   }
   ...
   //(6)将日志任务放入队列
   put(eventObject);
}

    private boolean isQueueBelowDiscardingThreshold() {
        return (blockingQueue.remainingCapacity() < discardingThreshold);
    }
```

其中代码（5）调用了 AsyncAppender 重写的 isDiscardable 方法，该方法的具体内容为

```
    //(7)
    protected boolean isDiscardable(ILoggingEvent event) {
        Level level = event.getLevel();
        return level.toInt() <= Level.INFO_INT;
    }
```

结合代码（5）和代码 (7) 可知，如果当前日志的级别小于等于 INFO_INT 并且当前队列的剩余容量小于 discardingThreshold 则会直接丢弃这些日志任务。

下面看具体代码 (6) 中的 put 方法。

```
    private void put(E eventObject) {
        //(8)
        if (neverBlock) {
            blockingQueue.offer(eventObject);
        } else {
            try {//(9)
                blockingQueue.put(eventObject);
            } catch (InterruptedException e) {
                // Interruption of current thread when in doAppend method should not
                   be consumed
                // by AsyncAppender
                Thread.currentThread().interrupt();
```

```
            }
        }
    }
```

如果 neverBlock 被设置为 false（默认为 false）则会调用阻塞队列的 put 方法，而 put 是阻塞的，也就是说如果当前队列满，则在调用 put 方法向队列放入一个元素时调用线程会被阻塞直到队列有空余空间。这里可以看下 put 方法的实现。

```
public void put(E e) throws InterruptedException {
    ...

    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        //如果队列满，则调用await方法阻塞当前调用线程
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

这里有必要解释下代码（9），当日志队列满时 put 方法会调用 await() 方法阻塞当前线程，而如果其他线程中断了该线程，那么该线程会抛出 InterruptedException 异常，并且当前的日志任务就会被丢弃。在 logback-classic 的 1.2.3 版本中，则添加了不对中断进行响应的方法。

```
private void put(E eventObject) {
    if (neverBlock) {
        blockingQueue.offer(eventObject);
    } else {
        putUninterruptibly(eventObject);
    }
}

private void putUninterruptibly(E eventObject) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                blockingQueue.put(eventObject);
                break;
```

```
                } catch (InterruptedException e) {
                    interrupted = true;
                }
            }
        } finally {
            if (interrupted) {
                Thread.currentThread().interrupt();
            }
        }
    }
```

如果当前日志打印线程在调用 blockingQueue.put 时被其他线程中断，则只是记录中断标志，然后继续循环调用 blockingQueue.put，尝试把日志任务放入日志队列。新版本的这个实现通过使用循环保证了即使当前线程被中断，日志任务最终也会被放入日志队列。

如果 neverBlock 被设置为 true 则会调用阻塞队列的 offer 方法，而该方法是非阻塞的，所以如果当前队列满，则会直接返回，也就是丢弃当前日志任务。这里回顾下 offer 方法的实现。

```
public boolean offer(E e) {
    ...
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //如果队列满则直接返回false。
        if (count == items.length)
            return false;
        else {
            enqueue(e);
            return true;
        }
    } finally {
        lock.unlock();
    }
}
```

最后来看 addAppender 方法都做了什么。

```
public void addAppender(Appender<E> newAppender) {
    if (appenderCount == 0) {
        appenderCount++;
        ...
        aai.addAppender(newAppender);
    } else {
```

```
            addWarn("One and only one appender may be attached to AsyncAppender.");
            addWarn("Ignoring additional appender named [" + newAppender.getName() + "]");
        }
    }
```

由如上代码可知，一个异步 appender 只能绑定一个同步 appender。这个 appender 会被放到 AppenderAttachableImpl 的 appenderList 列表里面。

到这里我们已经分析完了日志生产线程把日志任务放入日志队列的实现，下面一起来看消费线程是如何从队列里面消费日志任务并将其写入磁盘的。由于消费线程是一个线程，所以就从 worker 的 run 方法开始。

```
class Worker extends Thread {

        public void run() {

            AsyncAppenderBase<E> parent = AsyncAppenderBase.this;
            AppenderAttachableImpl<E> aai = parent.aai;

            //(10)一直循环直到该线程被中断
            while (parent.isStarted()) {
                try {//(11)从阻塞队列获取元素
                    E e = parent.blockingQueue.take();
                    aai.appendLoopOnAppenders(e);
                } catch (InterruptedException ie) {
                    break;
                }
            }

            //(12)到这里说明该线程被中断，则把队列里面的剩余日志任务
            //刷新到磁盘
            for (E e : parent.blockingQueue) {
                aai.appendLoopOnAppenders(e);
                parent.blockingQueue.remove(e);
            }
            ...
..          }
    }
```

其中代码（11）使用 take 方法从日志队列获取一个日志任务，如果当前队列为空则当前线程会被阻塞直到队列不为空才返回。获取到日志任务后会调用 AppenderAttachableImpl 的 aai.appendLoopOnAppenders 方法，该方法会循环调用通过 addAppender 注入的同步日志，

appener 具体实现把日志打印到磁盘。

11.1.3　小结

本节结合 logback 中异步日志的实现介绍了并发组件 ArrayBlockingQueue 的使用，包括 put、offer 方法的使用场景以及它们之间的区别，take 方法的使用，同时也介绍了如何使用 ArrayBlockingQueue 来实现一个多生产者 - 单消费者模型。另外使用 ArrayBlockingQueue 时需要注意合理设置队列的大小以免造成 OOM，队列满或者剩余元素比较少时，要根据具体场景制定一些抛弃策略以避免队列满时业务线程被阻塞。

11.2　Tomcat 的 NioEndPoint 中 ConcurrentLinkedQueue 的使用

本节讲解 apache-tomcat-7.0.32-src 源码中 ConcurrentLinkedQueue 的使用。首先介绍 Tomcat 的容器结构以及 NioEndPoint 的作用，以便后面能够更加平滑地切入话题，如图 11-4 所示是 Tomcat 的容器结构。

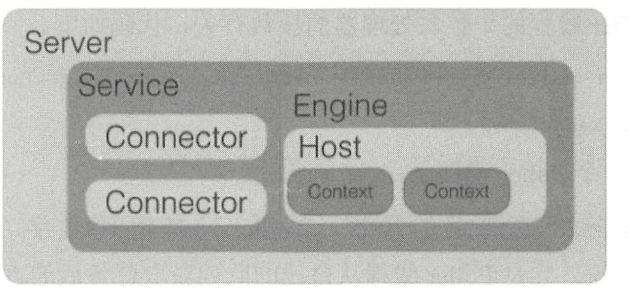

图 11-4

其中，Connector 是一个桥梁，它把 Server 和 Engine 连接了起来，Connector 的作用是接受客户端的请求，然后把请求委托给 Engine 容器处理。在 Connector 的内部具体使用 Endpoint 进行处理，根据处理方式的不同 Endpoint 可分为 NioEndpoint、JIoEndpoint、AprEndpoint。本节介绍 NioEndpoint 中的并发组件队列的使用。为了让读者更好地理解，有必要先说下 NioEndpoint 的作用。首先来看 NioEndpoint 中的三大组件的关系图（见图 11-5）。

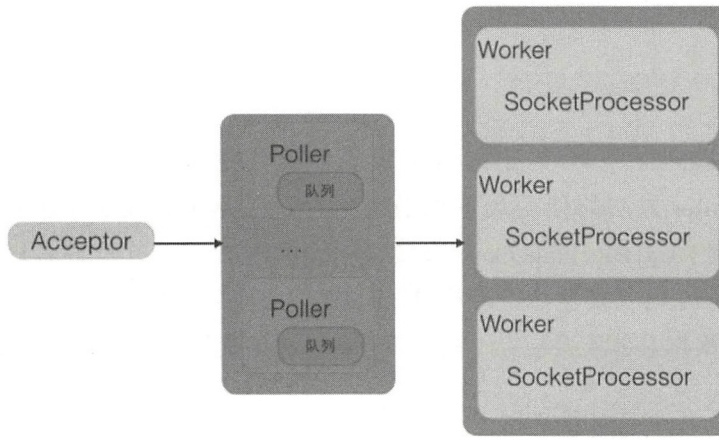

图 11-5

- Acceptor 是套接字接受线程（Socket acceptor thread），用来接受用户的请求，并把请求封装为事件任务放入 Poller 的队列，一个 Connector 里面只有一个 Acceptor。
- Poller 是套接字处理线程（Socket poller thread），每个 Poller 内部都有一个独有的队列，Poller 线程则从自己的队列里面获取具体的事件任务，然后将其交给 Worker 进行处理。Poller 线程的个数与处理器的核数有关，代码如下。

```
protected int pollerThreadCount = Math.min(2,Runtime.getRuntime().
availableProcessors());
```

  这里最多有 2 个 Poller 线程。

- Worker 是实际处理请求的线程，Worker 只是组件名字，真正做事情的是 SocketProcessor，它是 Poller 线程从自己的队列获取任务后的真正任务执行者。

可见，Tomcat 使用队列把接受请求与处理请求操作进行解耦，实现异步处理。其实 Tomcat 中 NioEndPoint 中的每个 Poller 里面都维护一个 ConcurrentLinkedQueue，用来缓存请求任务，其本身也是一个多生产者 - 单消费者模型。

11.2.1 生产者——Acceptor 线程

Acceptor 线程的作用是接受客户端发来的连接请求并将其放入 Poller 的事件队列。首先看下 Acceptor 处理请求的简明时序图（见图 11-6）。

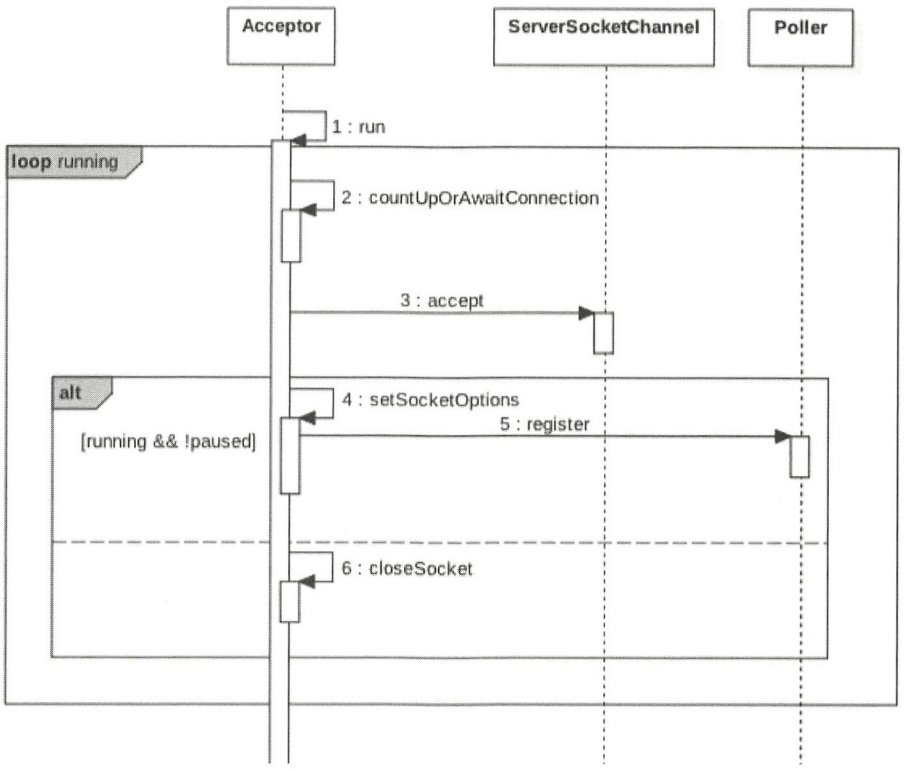

图 11-6

下面分析 Acceptor 的源码，看其如何把接受的套接字连接放入队列。

```
protected class Acceptor extends AbstractEndpoint.Acceptor {

        @Override
        public void run() {

            int errorDelay = 0;

            //(1) 一直循环直到接收到shutdown命令
            while (running) {

                ...

                if (!running) {
                    break;
                }
```

```
                state = AcceptorState.RUNNING;

                try {
                    //(2)如果达到max connections个请求则等待
                    countUpOrAwaitConnection();

                    SocketChannel socket = null;
                    try {
                        //(3)从TCP缓存获取一个完成三次握手的套接字，没有则阻塞
                        socket = serverSock.accept();
                    } catch (IOException ioe) {
                        ...
                    }
                    errorDelay = 0;
                   if (running && !paused) {
                        //(4)设置套接字参数并封装套接字为事件任务，然后放入Poller的队列
                        if (!setSocketOptions(socket)) {
                            countDownConnection();
                            closeSocket(socket);
                        }
                    } else {
                        countDownConnection();
                        closeSocket(socket);
                    }
                   ....
                } catch (SocketTimeoutException sx) {
                ....
            }
            state = AcceptorState.ENDED;
        }
    }
```

代码（1）中的无限循环用来一直等待客户端的连接，循环退出条件是调用了shutdown 命令。

代码（2）用来控制客户端的请求连接数量，如果连接数量达到设置的阈值，则当前请求会被挂起。

代码（3）从 TCP 缓存获取一个完成三次握手的套接字，如果当前没有，则当前线程会被阻塞挂起。

当代码（3）获取到一个连接套接字后，代码（4）会调用 setSocketOptions 设置该套接字。

```
 protected boolean setSocketOptions(SocketChannel socket) {
        // 处理链接
        try {

            ...
            //封装链接套接字为channel并注册到Poller队列
            getPoller0().register(channel);
        } catch (Throwable t) {
            ...
            return false;
        }
        return true;
}
```

代码（5）将连接套接字封装为一个 channel 对象，并将其注册到 poller 对象的队列。

```
//具体注册到事件队列
public void register(final NioChannel socket) {
   ...
    PollerEvent r = eventCache.poll();
            ka.interestOps(SelectionKey.OP_READ);//this is what OP_REGISTER turns into.
            if ( r==null) r = new PollerEvent(socket,ka,OP_REGISTER);
            else r.reset(socket,ka,OP_REGISTER);

    addEvent(r);
}
public void addEvent(Runnable event) {
       events.offer(event);
    ...
}
```

其中，events 的定义如下：

```
        protected ConcurrentLinkedQueue<Runnable> events = new ConcurrentLinkedQueue
<Runnable>();
```

由此可见，events 是一个无界队列 ConcurrentLinkedQueue，根据前文讲的，使用队列作为同步转异步的方式要注意设置队列大小，否则可能造成 OOM。当然 Tomcat 肯定不会忽略这个问题，从代码（2）可以看出，Tomcat 让用户配置了一个最大连接数，超过这个数则会等待。

11.2.2 消费者——Poller 线程

Poller 线程的作用是从事件队列里面获取事件并进行处理。首先我们从时序图来全局了解下 Poller 线程的处理逻辑（见图 11-7）。

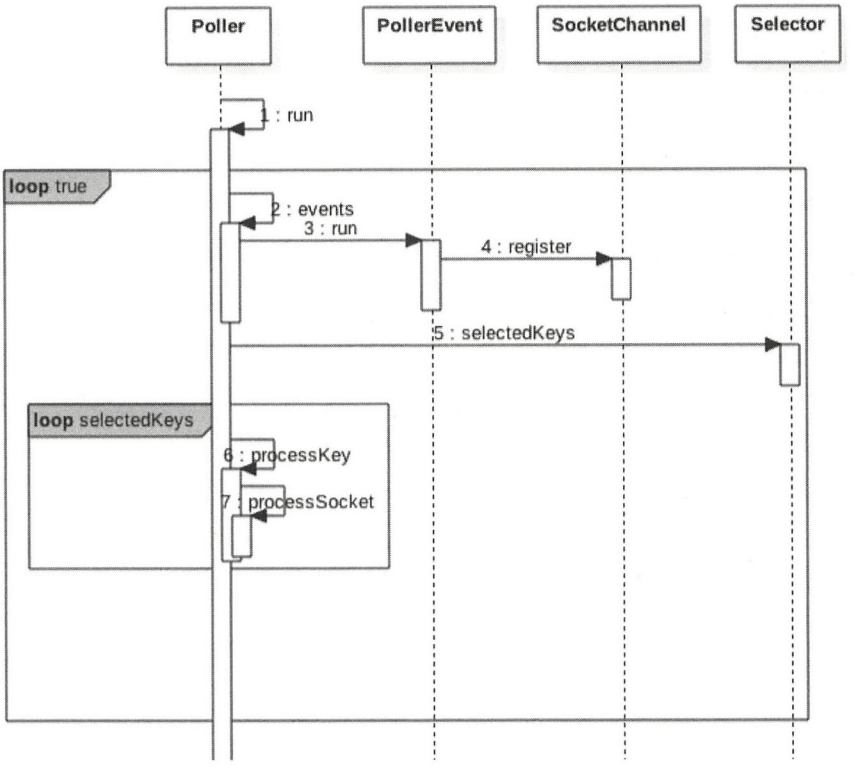

图 11-7

同理，我们看一下 Poller 线程的 run 方法代码逻辑。

```
public void run() {
    while (true) {
        try {
            ...
            if (close) {
               ...
            } else {
            //(6)从事件队列获取事件
                hasEvents = events();
```

```
        }
        try {
            ...
        } catch ( NullPointerException x ) {...
        }

        Iterator<SelectionKey> iterator =
            keyCount > 0 ? selector.selectedKeys().iterator() : null;
        //(7)遍历所有注册的channel并对感兴趣的事件进行处理
        while (iterator != null && iterator.hasNext()) {
            SelectionKey sk = iterator.next();
            KeyAttachment attachment = (KeyAttachment)sk.attachment();

            if (attachment == null) {
                iterator.remove();
            } else {
                attachment.access();
                iterator.remove();
             //(8)具体调用SocketProcessor进行处理
                processKey(sk, attachment);
            }
        }//while

       ...
    } catch (OutOfMemoryError oom) {
        ...
    }
 }//while
...
}
```

其中，代码（6）从 poller 的事件队列获取一个事件，events() 的代码如下。

```
public boolean events() {
    boolean result = false;

    //从队列获取任务并执行
    Runnable r = null;
    while ( (r = events.poll()) != null ) {
        result = true;
        try {
            r.run();
            ...
            } catch ( Throwable x ) {
```

```
                ...
            }
        }

        return result;
    }
```

这里是使用循环来实现的，目的是为了避免虚假唤醒。

其中代码（7）和代码（8）则遍历所有注册的 channel，并对感兴趣的事件进行处理。

```
    public boolean processSocket(NioChannel socket, SocketStatus status, boolean
dispatch) {
        try {
            ...
            SocketProcessor sc = processorCache.poll();
            if ( sc == null ) sc = new SocketProcessor(socket,status);
            else sc.reset(socket,status);
            if ( dispatch && getExecutor()!=null ) getExecutor().execute(sc);
            else sc.run();
        } catch (RejectedExecutionException rx) {
            ...
        } catch (Throwable t) {
            ...
            return false;
        }
        return true;
    }
```

11.2.3 小结

本节通过分析 Tomcat 中 NioEndPoint 的实现源码介绍了并发组件 ConcurrentLinkedQueue 的使用。NioEndPoint 的思想也是使用队列将同步转为异步，并且由于 ConcurrentLinkedQueue 是无界队列，所以需要让用户提供一个设置队列大小的接口以防止队列元素过多导致 OOM。

11.3 并发组件 ConcurrentHashMap 使用注意事项

ConcurrentHashMap 虽然为并发安全的组件，但是使用不当仍然会导致程序错误。本节通过简单的案例来复现这些问题，并给出开发时如何避免的策略。

这里借用直播的一个场景，在直播业务中，每个直播间对应一个 topic，每个用户进入直播间时会把自己设备的 ID 绑定到这个 topic 上，也就是一个 topic 对应一堆用户设备。可以使用 map 来维护这些信息，其中 key 为 topic，value 为设备的 list。下面使用代码来模拟多用户同时进入直播间时 map 信息的维护。

```
public class TestMap {
    //(1)创建map,key为topic,value为设备列表
    static ConcurrentHashMap<String, List<String>> map = new ConcurrentHashMap<>();
    public static void main(String[] args) {
        //(2)进入直播间topic1, 线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device1");
                list1.add("device2");

                map.put("topic1", list1);
                System.out.println(JSON.toJSONString(map));
            }
        });
        //(3)进入直播间topic1, 线程two
        Thread threadTwo = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device11");
                list1.add("device22");

                map.put("topic1", list1);

                System.out.println(JSON.toJSONString(map));
            }
        });

        //(4)进入直播间topic2, 线程three
        Thread threadThree = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device111");
                list1.add("device222");

                map.put("topic2", list1);
```

```
                System.out.println(JSON.toJSONString(map));
            }
        });

        //(5)启动线程
        threadOne.start();
        threadTwo.start();
        threadThree.start();
    }
}
```

代码（1）创建了一个并发 map，用来存放 topic 及与其对应的设备列表。

代码（2）和代码（3）模拟用户进入直播间 topic1，代码（4）模拟用户进入直播间 topic2。

代码（5）启动线程。

运行代码，输出结果如下。

```
{"topic1":["device11","device22"],"topic2":["device111","device222"]}
{"topic1":["device11","device22"],"topic2":["device111","device222"]}
{"topic1":["device11","device22"],"topic2":["device111","device222"]}
```

或者输出如下。

```
{"topic1":["device1","device2"],"topic2":["device111","device222"]}
{"topic1":["device1","device2"],"topic2":["device111","device222"]}
{"topic1":["device1","device2"],"topic2":["device111","device222"]}
```

可见，topic1 房间中的用户会丢失一部分，这是因为 put 方法如果发现 map 里面存在这个 key，则使用 value 覆盖该 key 对应的老的 value 值。而 putIfAbsent 方法则是，如果发现已经存在该 key 则返回该 key 对应的 value，但并不进行覆盖，如果不存在则新增该 key，并且判断和写入是原子性操作。使用 putIfAbsent 替代 put 方法后的代码如下。

```
public class TestMap2 {
    //(1)创建map,key为topic,value为设备列表
    static ConcurrentHashMap<String, List<String>> map = new ConcurrentHashMap<>();
    public static void main(String[] args) {
        //(2)进入直播间topic1,  线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                List<String> list1 = new ArrayList<>();
                list1.add("device1");
```

```
            list1.add("device2");
            //(2.1)
            List<String> oldList = map.putIfAbsent("topic1", list1);
            if(null != oldList){
                oldList.addAll(list1);
            }
            System.out.println(JSON.toJSONString(map));
        }
    });
    //(3)进入直播间topic1, 线程two
    Thread threadTwo = new Thread(new  Runnable() {
        public void run() {
            List<String> list1 = new ArrayList<>();
            list1.add("device11");
            list1.add("device22");

            List<String> oldList = map.putIfAbsent("topic1", list1);
            if(null != oldList){
                oldList.addAll(list1);
            }

            System.out.println(JSON.toJSONString(map));
        }
    });

    //(4)进入直播间topic2, 线程three
    Thread threadThree = new Thread(new  Runnable() {
        public void run() {
            List<String> list1 = new ArrayList<>();
            list1.add("device111");
            list1.add("device222");

            List<String> oldList = map.putIfAbsent("topic2", list1);
            if(null != oldList){
                oldList.addAll(list1);
            }
            System.out.println(JSON.toJSONString(map));
        }
    });

    //(5)启动线程
    threadOne.start();
    threadTwo.start();
```

```
        threadThree.start();
    }
}
```

在如上代码（2.1）中，使用 map.putIfAbsent 方法添加新设备列表，如果 topic1 在 map 中不存在，则将 topic1 和对应设备列表放入 map。要注意的是，这个判断和放入是原子性操作，放入后会返回 null。如果 topic1 已经在 map 里面存在，则调用 putIfAbsent 会返回 topic1 对应的设备列表，若发现返回的设备列表不为 null 则把新的设备列表添加到返回的设备列表里面，从而问题得到解决。

运行结果为

```
{"topic1":["device1","device2","device11","device22"],"topic2":["device111","devi
ce222"]}
{"topic1":["device1","device2","device11","device22"],"topic2":["device111","devi
ce222"]}
{"topic1":["device1","device2","device11","device22"],"topic2":["device111","devi
ce222"]}
```

总结：put(K key, V value) 方法判断如果 key 已经存在，则使用 value 覆盖原来的值并返回原来的值，如果不存在则把 value 放入并返回 null。而 putIfAbsent(K key, V value) 方法则是如果 key 已经存在则直接返回原来对应的值并不使用 value 覆盖，如果 key 不存在则放入 value 并返回 null，另外要注意，判断 key 是否存在和放入是原子性操作。

11.4 SimpleDateFormat 是线程不安全的

SimpleDateFormat 是 Java 提供的一个格式化和解析日期的工具类，在日常开发中经常会用到，但是由于它是线程不安全的，所以多线程共用一个 SimpleDateFormat 实例对日期进行解析或者格式化会导致程序出错。本节来揭示它为何是线程不安全的，以及如何避免该问题。

11.4.1 问题复现

为了复现问题，编写如下代码。

```
public class TestSimpleDateFormat {
    //(1)创建单例实例
    static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

```
    public static void main(String[] args) {
        //(2)创建多个线程，并启动
        for (int i = 0; i <10 ; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {//(3)使用单例日期实例解析文本
                        System.out.println(sdf.parse("2017-12-13 15:17:27"));
                    } catch (ParseException e) {
                        e.printStackTrace();
                    }
                }
            });
            thread.start();//(4)启动线程
        }
    }
}
```

代码（1）创建了 SimpleDateFormat 的一个实例，代码（2）创建 10 个线程，每个线程都共用同一个 sdf 对象对文本日期进行解析。多运行几次代码就会抛出 java.lang.NumberFormatException 异常，增加线程的个数有利于复现该问题。

11.4.2　问题分析

为了便于分析，首先来看 SimpleDateFormat 的类图结构（见图 11-8）。

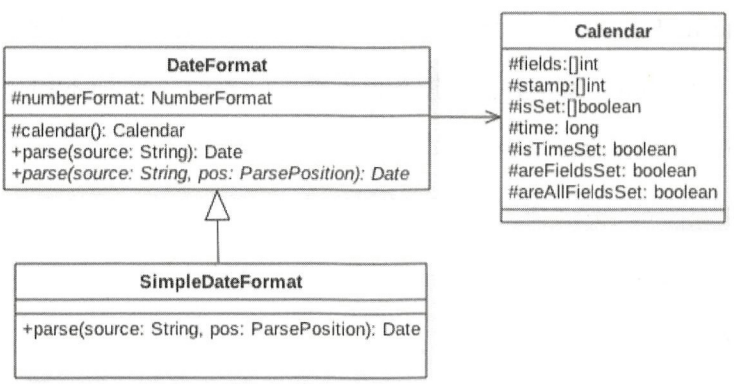

图 11-8

可以看到，每个 SimpleDateFormat 实例里面都有一个 Calendar 对象，后面我们就会知道，SimpleDateFormat 之所以是线程不安全的，就是因为 Calendar 是线程不安全的。后者

之所以是线程不安全的，是因为其中存放日期数据的变量都是线程不安全的，比如 fields、time 等。

下面从代码层面来看下 parse 方法做了什么事情。

```
public Date parse(String text, ParsePosition pos)
{

    //(1)解析日期字符串，并将解析好的数据放入CalendarBuilder的实例calb中
    ...

    Date parsedDate;
    try {//(2)使用calb中解析好的日期数据设置calendar
        parsedDate = calb.establish(calendar).getTime();
        ...
    }

    catch (IllegalArgumentException e) {
       ...
        return null;
    }

    return parsedDate;
}
```

代码（1）的主要作用是解析日期字符串并把解析好的数据放入 CalendarBuilder 的实例 calb 中。CalendarBuilder 是一个建造者模式，用来存放后面需要的数据。

代码（2）使用 calb 中解析好的日期数据设置 calendar，calb.establish 的代码如下。

```
Calendar establish(Calendar cal) {
   ...
   //(3)重置日期对象cal的属性值
   cal.clear();
   //(4) 使用calb中的属性设置cal
   ...
   //(5)返回设置好的cal对象
   return cal;
}
```

代码（3）重置 Calendar 对象里面的属性值，如下所示。

```
 public final void clear()
{
```

```
        for (int i = 0; i < fields.length; ) {
            stamp[i] = fields[i] = 0; // UNSET == 0
            isSet[i++] = false;
        }
        areAllFieldsSet = areFieldsSet = false;
        isTimeSet = false;
    }
```

代码（4）使用 calb 中解析好的日期数据设置 cal 对象。

代码（5）返回设置好的 cal 对象。

从以上代码可以看出，代码（3）、代码（4）和代码（5）并不是原子性操作。当多个线程调用 parse 方法时，比如线程 A 执行了代码（3）和代码（4），也就是设置好了 cal 对象，但是在执行代码（5）之前，线程 B 执行了代码（3），清空了 cal 对象。由于多个线程使用的是一个 cal 对象，所以线程 A 执行代码（5）返回的可能就是被线程 B 清空的对象，当然也有可能线程 B 执行了代码（4），设置被线程 A 修改的 cal 对象，从而导致程序出现错误。

那么怎么解决呢？

- 第一种方式：每次使用时 new 一个 SimpleDateFormat 的实例，这样可以保证每个实例使用自己的 Calendar 实例，但是每次使用都需要 new 一个对象，并且使用后由于没有其他引用，又需要回收，开销会很大。
- 第二种方式：出错的根本原因是因为多线程下代码（3）、代码（4）和代码（5）三个步骤不是一个原子性操作，那么容易想到的是对它们进行同步，让代码（3）、代码（4）和代码（5）成为原子性操作。可以使用 synchronized 进行同步，具体如下。

```
public class TestSimpleDateFormat {
    // (1)创建单例实例
    static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args) {
        // (2)创建多个线程，并启动
        for (int i = 0; i < 10; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {// (3)使用单例日期实例解析文本
                        synchronized (sdf) {
                            System.out.println(sdf.parse("2017-12-13 15:17:27"));
```

```
                    }
                } catch (ParseException e) {
                    e.printStackTrace();
                }
            }
        });
        thread.start();// (4)启动线程
    }
}
```

进行同步意味着多个线程要竞争锁，在高并发场景下这会导致系统响应性能下降。

- 第三种方式：使用 ThreadLocal，这样每个线程只需要使用一个 SimpleDateFormat 实例，这相比第一种方式大大节省了对象的创建销毁开销，并且不需要使多个线程同步。使用 ThreadLocal 方式的代码如下。

```
public class TestSimpleDateFormat2 {
    // (1)创建threadlocal实例
    static ThreadLocal<DateFormat> safeSdf = new ThreadLocal<DateFormat>(){
        @Override
        protected SimpleDateFormat initialValue(){
            return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        }
    };

    public static void main(String[] args) {
        // (2)创建多个线程，并启动
        for (int i = 0; i < 10; ++i) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {// (3)使用单例日期实例解析文本
                            System.out.println(safeSdf.get().parse("2017-12-13
                             15:17:27"));
                    } catch (ParseException e) {
                        e.printStackTrace();
                    }finally {
                        //(4)使用完毕记得清除，避免内存泄漏
                        safeSdf.remove();
                    }
                }
            });
            thread.start();// (5)启动线程
```

```
        }
    }
}
```

代码（1）创建了一个线程安全的 SimpleDateFormat 实例，代码（3）首先使用 get() 方法获取当前线程下 SimpleDateFormat 的实例。在第一次调用 ThreadLocal 的 get()方法时，会触发其 initialValue 方法创建当前线程所需要的 SimpleDateFormat 对象。另外需要注意的是，在代码（4）中，使用完线程变量后，要进行清理，以避免内存泄漏。

11.4.3　小结

本节通过简单介绍 SimpleDateFormat 的原理解释了为何 SimpleDateFormat 是线程不安全的，应该避免在多线程下使用 SimpleDateFormat 的单个实例。

11.5　使用 Timer 时需要注意的事情

当一个 Timer 运行多个 TimerTask 时，只要其中一个 TimerTask 在执行中向 run 方法外抛出了异常，则其他任务也会自动终止。

11.5.1　问题的产生

这里做了一个小的 demo 来复现问题，代码如下。

```
public class TestTimer {
   //创建定时器对象
    static Timer timer = new Timer();

    public static void main(String[] args) {
     //添加任务1,延迟500ms执行
        timer.schedule(new TimerTask() {

            @Override
            public void run() {
                System.out.println("---one Task---");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
                throw new RuntimeException("error ");
            }
        }, 500);
        //添加任务2，延迟1000ms执行
        timer.schedule(new TimerTask() {

            @Override
            public void run() {
                for (;;) {
                    System.out.println("---two Task---");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }, 1000);

    }
}
```

如上代码首先添加了第一个任务，让其在 500ms 后执行。然后添加了第二个任务在 1s 后执行，我们期望当第一个任务输出 ---one Task--- 后，等待 1s，第二个任务输出 ---two Task---，但是执行代码后，输出结果为

```
---one Task---
Exception in thread "Timer-0" java.lang.RuntimeException: error
    at com.zlx.Timer.TestTimer$1.run(TestTimer.java:22)
    at java.util.TimerThread.mainLoop(Timer.java:555)
    at java.util.TimerThread.run(Timer.java:505)
```

11.5.2 Timer 实现原理分析

下面简单介绍 Timer 的原理，如图 11-9 所示是 Timer 的原理模型。

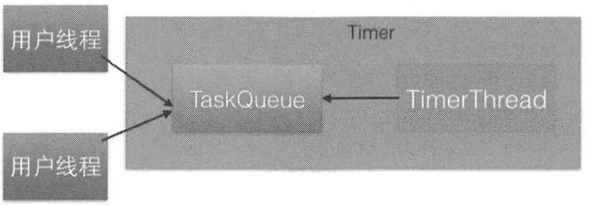

图11-9

- TaskQueue是一个由平衡二叉树堆实现的优先级队列，每个Timer对象内部有一个TaskQueue队列。用户线程调用Timer的schedule方法就是把TimerTask任务添加到TaskQueue队列。在调用schedule方法时，long delay参数用来指明该任务延迟多少时间执行。
- TimerThread是具体执行任务的线程，它从TaskQueue队列里面获取优先级最高的任务进行执行。需要注意的是，只有执行完了当前的任务才会从队列里获取下一个任务，而不管队列里是否有任务已经到了设置的delay时间。一个Timer只有一个TimerThread线程，所以可知Timer的内部实现是一个多生产者 - 单消费者模型。

从该实现模型我们知道，要探究上面的问题只需研究TimerThread的实现就可以了。TimerThread的run方法的主要逻辑代码如下。

```
public void run() {
    try {
        mainLoop();
    } finally {
        // Someone killed this Thread, behave as if Timer cancelled
        synchronized(queue) {
            newTasksMayBeScheduled = false;
            queue.clear();  // Eliminate obsolete references
        }
    }
}

 private void mainLoop() {
        while (true) {
            try {
                TimerTask task;
                boolean taskFired;
                //从队列里面获取任务时要加锁
                synchronized(queue) {
                    ...
```