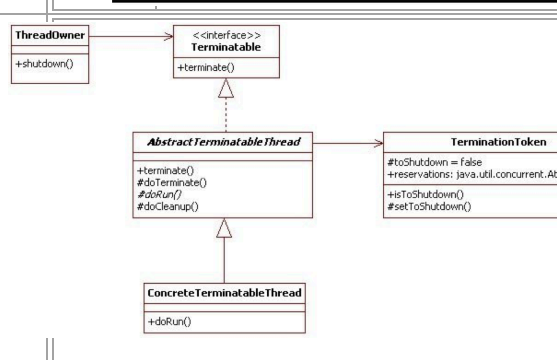
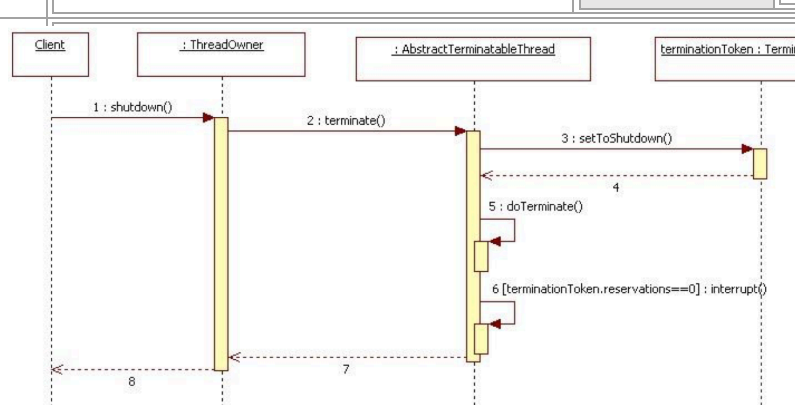


CH05 Two-phase Termination

2017年4月22日 星期六
下午9:52

用途	<table><tr><td>目的</td><td colspan="2">通过设置标志位, 让线程安全终止</td></tr><tr><td>阶段</td><td colspan="2"><div>阶段1: 准备阶段。该阶段的主要动作是“通知”目标线程（欲停止的线程）准备进行停止。这一步会设置一个标志变量用于指示目标线程可以准备停止了。但是, 由于目标线程可能正处于阻塞状态（等待锁的获得）、等待状态（如调用Object.wait）或者I/O（如InputStream.read）等待状态, 即便设置了这个标志, 目标线程也无法立即“看到”这个标志而做出相应动作。因此, 这一阶段还需要通过调用目标线程的interrupt方法, 以期目标线程能够通过捕获相关的异常侦测到该方法调用, 从而中断其阻塞状态、等待状态。对于能够对interrupt方法调用做出响应的方法（参见表5-1）, 目标线程代码可以通过捕获这些方法抛出的InterruptedException来侦测线程停止信号。但也有一些方法（如InputStream.read）并不对interrupt调用做出响应, 此时需要我们手工处理, 如同步的Socket I/O操作中通过关闭socket, 使处于I/O等待的socket抛出java.net.SocketException。</div><div>阶段2: 执行阶段。该阶段的主要动作是检查准备阶段所设置的线程停止标志和信号, 在此基础上决定线程停止的时机, 并进行适当的“清理”操作。</div></td></tr></table>		目的	通过设置标志位, 让线程安全终止		阶段	<div>阶段1: 准备阶段。该阶段的主要动作是“通知”目标线程（欲停止的线程）准备进行停止。这一步会设置一个标志变量用于指示目标线程可以准备停止了。但是, 由于目标线程可能正处于阻塞状态（等待锁的获得）、等待状态（如调用Object.wait）或者I/O（如InputStream.read）等待状态, 即便设置了这个标志, 目标线程也无法立即“看到”这个标志而做出相应动作。因此, 这一阶段还需要通过调用目标线程的interrupt方法, 以期目标线程能够通过捕获相关的异常侦测到该方法调用, 从而中断其阻塞状态、等待状态。对于能够对interrupt方法调用做出响应的方法（参见表5-1）, 目标线程代码可以通过捕获这些方法抛出的InterruptedException来侦测线程停止信号。但也有一些方法（如InputStream.read）并不对interrupt调用做出响应, 此时需要我们手工处理, 如同步的Socket I/O操作中通过关闭socket, 使处于I/O等待的socket抛出java.net.SocketException。</div> <div>阶段2: 执行阶段。该阶段的主要动作是检查准备阶段所设置的线程停止标志和信号, 在此基础上决定线程停止的时机, 并进行适当的“清理”操作。</div>																			
目的	通过设置标志位, 让线程安全终止																									
阶段	<div>阶段1: 准备阶段。该阶段的主要动作是“通知”目标线程（欲停止的线程）准备进行停止。这一步会设置一个标志变量用于指示目标线程可以准备停止了。但是, 由于目标线程可能正处于阻塞状态（等待锁的获得）、等待状态（如调用Object.wait）或者I/O（如InputStream.read）等待状态, 即便设置了这个标志, 目标线程也无法立即“看到”这个标志而做出相应动作。因此, 这一阶段还需要通过调用目标线程的interrupt方法, 以期目标线程能够通过捕获相关的异常侦测到该方法调用, 从而中断其阻塞状态、等待状态。对于能够对interrupt方法调用做出响应的方法（参见表5-1）, 目标线程代码可以通过捕获这些方法抛出的InterruptedException来侦测线程停止信号。但也有一些方法（如InputStream.read）并不对interrupt调用做出响应, 此时需要我们手工处理, 如同步的Socket I/O操作中通过关闭socket, 使处于I/O等待的socket抛出java.net.SocketException。</div> <div>阶段2: 执行阶段。该阶段的主要动作是检查准备阶段所设置的线程停止标志和信号, 在此基础上决定线程停止的时机, 并进行适当的“清理”操作。</div>																									
附录	<table><tr><th>中断响</th><th>方法（或者类）</th><th>响应 interrupt 调用抛出的异常</th></tr><tr><td></td><td>Object.wait() 、Object.wait(long timeout) 、Object.wait(long timeout, int nanos)</td><td>InterruptedException</td></tr><tr><td></td><td>Thread.sleep(long millis) 、 Thread.sleep(long millis, int nanos)</td><td>InterruptedException</td></tr><tr><td></td><td>Thread.join() 、 Thread.join(long millis) 、 Thread.Join(long millis, int nanos)</td><td>InterruptedException</td></tr><tr><td></td><td>java.util.concurrent.BlockingQueue.take()</td><td>InterruptedException</td></tr><tr><td></td><td>java.util.concurrent.locks.Lock.lockInterruptibly()</td><td>InterruptedException</td></tr><tr><td></td><td>java.nio.channels.InterruptibleChannel</td><td>java.nio.channels.ClosedByInterruptException</td></tr></table>		中断响	方法（或者类）	响应 interrupt 调用抛出的异常		Object.wait() 、Object.wait(long timeout) 、Object.wait(long timeout, int nanos)	InterruptedException		Thread.sleep(long millis) 、 Thread.sleep(long millis, int nanos)	InterruptedException		Thread.join() 、 Thread.join(long millis) 、 Thread.Join(long millis, int nanos)	InterruptedException		java.util.concurrent.BlockingQueue.take()	InterruptedException		java.util.concurrent.locks.Lock.lockInterruptibly()	InterruptedException		java.nio.channels.InterruptibleChannel	java.nio.channels.ClosedByInterruptException			
中断响	方法（或者类）	响应 interrupt 调用抛出的异常																								
	Object.wait() 、Object.wait(long timeout) 、Object.wait(long timeout, int nanos)	InterruptedException																								
	Thread.sleep(long millis) 、 Thread.sleep(long millis, int nanos)	InterruptedException																								
	Thread.join() 、 Thread.join(long millis) 、 Thread.Join(long millis, int nanos)	InterruptedException																								
	java.util.concurrent.BlockingQueue.take()	InterruptedException																								
	java.util.concurrent.locks.Lock.lockInterruptibly()	InterruptedException																								
	java.nio.channels.InterruptibleChannel	java.nio.channels.ClosedByInterruptException																								
类图	<div></div> <table><tr><td>ThreadOwner</td><td>用户代码</td></tr><tr><td>Terminatable#terminatable</td><td>提供给用户代码的入口</td></tr><tr><td>TerminationToken</td><td>线程终止标志位 reservations反映还有多少任务未完成(如果目标线程希望处理完所有任务再结束这个变量有帮助)</td></tr><tr><td>AbstractTerminatableThread</td><td>封装信号处理, 检查标志位等操作</td></tr><tr><td>terminate()</td><td>用户代码入口具体实现</td></tr><tr><td>doTerminate()</td><td>留给子类实现额外操作, 如果关闭socketIO</td></tr><tr><td>doRun()</td><td>留给子类实现线程逻辑</td></tr><tr><td>doCleanup()</td><td>留给子类实现清理操作等</td></tr><tr><td>ConcreteTerminatableThread</td><td>封装业务逻辑, 由客户代码实现</td></tr><tr><td>doRun()</td><td>根据情况决定是否实现</td></tr><tr><td>doTerminate()</td><td></td></tr><tr><td>doCleanup()</td><td></td></tr></table>		ThreadOwner	用户代码	Terminatable#terminatable	提供给用户代码的入口	TerminationToken	线程终止标志位 reservations反映还有多少任务未完成(如果目标线程希望处理完所有任务再结束这个变量有帮助)	AbstractTerminatableThread	封装信号处理, 检查标志位等操作	terminate()	用户代码入口具体实现	doTerminate()	留给子类实现额外操作, 如果关闭socketIO	doRun()	留给子类实现线程逻辑	doCleanup()	留给子类实现清理操作等	ConcreteTerminatableThread	封装业务逻辑, 由客户代码实现	doRun()	根据情况决定是否实现	doTerminate()		doCleanup()	
ThreadOwner	用户代码																									
Terminatable#terminatable	提供给用户代码的入口																									
TerminationToken	线程终止标志位 reservations反映还有多少任务未完成(如果目标线程希望处理完所有任务再结束这个变量有帮助)																									
AbstractTerminatableThread	封装信号处理, 检查标志位等操作																									
terminate()	用户代码入口具体实现																									
doTerminate()	留给子类实现额外操作, 如果关闭socketIO																									
doRun()	留给子类实现线程逻辑																									
doCleanup()	留给子类实现清理操作等																									
ConcreteTerminatableThread	封装业务逻辑, 由客户代码实现																									
doRun()	根据情况决定是否实现																									
doTerminate()																										
doCleanup()																										
时序	<div></div> <div>第1\2\3\4步: TerminationToken#toShutdown设置为true</div> <div>第5步: doTerminate(), 由客户代码提供的子类实现, 执行关闭socketIO等操作</div> <div>第6步: reservation == 0, 由线程来设置, 表示所有任务都执行完, 或者线程不关心是否还有未执行的任务, 可以结束。此时才调用interrupt()</div> <div>第7\8步: shutdown()返回, 此时只是触发了线程的优雅结束机制, 线程仍然可能在执行清理工作等, 并未完全退出</div>																									
考虑	Two-phase Termination模式使得我们可以对各种形式的目标线程进行优雅地停止。如目标线程调用了能够对interrupt方法调用做出响应的阻塞方法、目标线程调用了不能对interrupt方法调用做出响应的阻塞方法、目标线程作为消费者处理其他线程生产的“产品”在其停止前需处理完现有“产品”等。Two-phase Termination模式实现的线程停止可能出现延迟, 即客户端代码调用完ThreadOwner.shutdown后, 该线程可																									

能切任运行。

实现要领

停止标志

本章案例使用了TerminationToken作为目标线程可以准备停止的标志。从清单5-4的代码我们可以看到，TerminationToken使用了toShutdown这个boolean变量作为主要的停止标志，而非使用Thread.isInterrupted()。这是因为，调用目标线程的interrupt方法无法保证目标线程的isInterrupted()方法返回值为true；目标线程可能调用一些代码，它们捕获InterruptedException后没有通过调用Thread.currentThread().interrupt()保留线程中断状态。另外，toShutdown这个变量为了保证内存可见性而又能避免使用显式锁的开销，采用了volatile修饰。这点也很重要，笔者曾经见过一些采用boolean变量作为线程停止标志的代码，只是这些变量没有用volatile修饰，对其访问也没有加锁，这就可能无法停止目标线程。

另外，某些场景下多个可停止线程实例可能需要共用一个线程停止标志。例如，多个可停止线程实例“消耗”同一个队列中的数据。当该队列为空且不再有新的数据入队列的时候，“消耗”该队列数据的所有可停止线程都应该被停掉。AbstractTerminatableThread类（源码见清单5-3）的构造器支持传入一个TerminationToken实例就是为了支持这种场景。

停止顺序

如果关心未完成任务如何处理，先关闭提供任务的线程（生产者），再关闭处理任务的线程（消费者）
例子：电子书1201

隐藏可止线程

为了保证可停止的线程不被其他代码误停止，一般我们将可停止线程隐藏在线程所有者背后，而使系统中其他代码无法直接访问该线程。正如本案例代码（见清单5-1）所展示：AlarmMgr定义了一个private字段alarmSendingThread用于引用告警发送线程（可停止的线程），系统中的其他代码只能通过调用AlarmMgr的shutdown方法来请求该线程停止，而非通过引用该线程对象自身来停止它。

JDK停

类java.util.concurrent.ThreadPoolExecutor就使用了Two-phase Termination模式来停止其内部维护的工作者线程。当客户端代码调用ThreadPoolExecutor实例的shutdown方法请求其关闭时，ThreadPoolExecutor会先将其运行状态设置为SHUTDOWN。工作者线程的run方法会判断其所属的ThreadPoolExecutor实例的运行状态。若ThreadPoolExecutor实例的运行状态为SHUTDOWN，则工作者线程会一直取工作队列中的任务进行执行，直到工作队列为空时该工作者线程就停止了。可见，ThreadPoolExecutor实例的停止过程也是分为准备阶段（设置其运行状态为SHUTDOWN）和执行阶段（工作者队列取空工作队列中的任务，然后终止线程）。

例子

电子书1060

类	角色	说明
AlarmMgr	ThreadOwner	调用alarmSendingThread.shutdown()触发两阶段终止 变量volatile boolean shutdownRequested防止alarmSendingThread.shutdown被调用两次
AlarmSendingThread	ConcreteTermionatableThread	电子书1091排版有误，其实是 public AlarmSendingThread() { //阻塞队列 alarmQueue = new ArrayBlockingQueue<AlarmInfo>(100); //ConcurrentHashMap submittedAlarmRegistry = new ConcurrentHashMap<String, AtomicInteger>(); alarmAgent.init(); } @override protected void doRun() throws Exception { AlarmInfo alarmInfo = alarmQueue.take(); //阻塞取，阻塞时收到中断应当会抛异常 terminationToken.reversions.decrementAndGet(); //Automatic变量的DAG原子操作 ... try { alarmAgent.sendRequest(); } catch (Exception e) { ... } } //外部对象传递alarm给这个AlarmSendingThread public int sendAlarm(final AlarmInfo alarmInfo) { ... if (terminationToken.isToShutdown()) { //不再接受新的任务 return -1; } ... //已经发生的报警，仅增加远程server上记录的报警次数 //新发生的报警，要增加terminationToken和入队操作 alarmQueue.put(alarmInfo); terminationToken.reversions.incrementAndGet(); ... } //设置好结束标志位后的清理操作 @override void doCleanup() { ... //关闭到远程服务器的连接 ... }
其他		见可复用代码

可复用代码

AbstractTerminatableThread

```
/**  * 可停止的抽象线程。  *  * 模式角色：Two-phaseTermination.AbstractTerminatableThread  *  * @author Viscent Huang  */ public abstract class AbstractTerminatableThread extends Thread implements     Terminatable {      // 模式角色：Two-phaseTermination.TerminationToken     public final TerminationToken terminationToken;      public AbstractTerminatableThread() {         this(new TerminationToken());     }      /**      *      */ }
```

```

    * @param terminationToken
    *          线程间共享的线程终止标志实例
    */
    public AbstractTerminatableThread(TerminationToken terminationToken) {
        super();
        this.terminationToken = terminationToken;
        terminationToken.register(this);
    }
    /**
    * 留给子类实现其线程处理逻辑。
    *
    * @throws Exception
    */
    protected abstract void doRun() throws Exception;
    /**
    * 留给子类实现。用于实现线程停止后的一些清理动作。
    *
    * @param cause
    */
    protected void doCleanup(Exception cause) {
        // 什么也不做
    }
    /**
    * 留给子类实现。用于执行线程停止所需的操作。
    */
    protected void doTerminate() {
        // 什么也不做
    }
    @Override
    public void run() {
        Exception ex = null;
        try {
            for (;;) {
                // 在执行线程的处理逻辑前先判断线程停止的标志。
                if (terminationToken.isToShutdown()
                    && terminationToken.reservations.get() <= 0) {
                    break;
                }
                doRun();
            }
        } catch (Exception e) {
            // 使得线程能够响应interrupt调用而退出
            ex = e;
        } finally {
            try {
                doCleanup(ex);
            } finally {
                terminationToken.notifyThreadTermination(this);
            }
        }
    }
    @Override
    public void interrupt() {
        terminate();
    }
    /**
    * 请求停止线程。
    *
    * @see io.github.viscent.mtpattern.tpt.Terminatable#terminate()
    */
    @Override
    public void terminate() {
        terminationToken.setToShutdown(true);
        try {
            doTerminate();
        }
    }

```

```

    } finally {

        // 若无待处理的任务，则试图强制终止线程
        if (terminationToken.reservations.get() <= 0) {
            super.interrupt();
        }
    }
}

public void terminate(boolean waitUtilThreadTerminated) {
    terminate();
    if (waitUtilThreadTerminated) {
        try {
            this.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
}

```

```

TerminationToken public class TerminationToken {

    // 使用volatile修饰，以保证无须显式锁的情况下该变量的内存可见性
    protected volatile boolean toShutdown = false;
    public final AtomicInteger reservations = new AtomicInteger(0);

    /**
     * 在多个可停止线程实例共享一个TerminationToken实例的情况下，该队列用于记录那些共享
     * TerminationToken实例的可停止线程，以便尽可能减少锁的使用的情况下，实现这些线程的停止。
     */
    private final Queue<WeakReference<Terminatable>> coordinatedThreads;

    public TerminationToken() {
        coordinatedThreads = new ConcurrentLinkedQueue<WeakReference<Terminatable>> ();
    }

    public boolean isToShutdown() {
        return toShutdown;
    }

    protected void setToShutdown(boolean toShutdown) {
        this.toShutdown = true;
    }

    protected void register(Terminatable thread) {
        coordinatedThreads.add(new WeakReference<Terminatable>(thread));
    }

    /**
     * 通知TerminationToken实例：共享该实例的所有可停止线程中的一个线程停止了，
     * 以便其停止其他未被停止的线程。
     * @param thread
     *         已停止的线程
     */
    protected void notifyThreadTermination(Terminatable thread) {
        WeakReference<Terminatable> wrThread;
        Terminatable otherThread;
        while (null != (wrThread = coordinatedThreads.poll())) {
            otherThread = wrThread.get();
            if (null != otherThread && otherThread != thread) {
                otherThread.terminate();
            }
        }
    }
}

```