- NETTY

精尽 Netty 源码解析 —— EventLoop（八）之 EventLoop 优雅关闭

1. 概述

文章目录

所以关于 EventLoop 的优雅关闭的分享，先放在后续的计划里。

兴趣的胖友，先准备好了两篇不错的文章：

- Hypercube 《自顶向下深入分析 Netty（四）–优雅退出机制》
- tomas家的小拨浪鼓 《Netty 源码解析 ——— Netty 优雅关闭流程》

为避免可能 《自顶向下深入分析 Netty（四）–优雅退出机制》 被作者删除，笔者这里先复制一份作为备份。

666. Netty优雅退出机制

你也许已经习惯了使用下面的代码，使一个线程池退出：

```
bossGroup.shutdownGracefully();
```

那么它是如何工作的呢？由于bossGroup是一个线程池，线程池的关闭要求其中的每一个线程关闭。而线程的实现是在SingleThreadEventExecutor类，所以我们将再次回到这个类，首先看其中的shutdownGracefully()方法，其中的参数quietPeriod为静默时间，timeout为截止时间，此外还有一个相关参数gracefulShutdownStartTime即优雅关闭开始时间，代码如下：

```
@Override
public Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit) {
    if (isShuttingDown()) {
        return terminationFuture(); // 正在关闭阻止其他线程
    }

    boolean inEventLoop = inEventLoop();
    boolean wakeup;
    int oldState;
    for (;;) {
        if (isShuttingDown()) {
            return terminationFuture(); // 正在关闭阻止其他线程
        }
        int newState;
        wakeup = true;
        oldState = STATE_UPDATER.get(this);
```

```
        if (inEventLoop) {
            newState = ST_SHUTTING_DOWN;
        } else {
            switch (oldState) {
                case ST_NOT_STARTED:
                case ST_STARTED:
                    newState = ST_SHUTTING_DOWN;
                    break;
                default: // 一个线程已修改好线程状态，此时这个线程才执行16行代码
                    newState = oldState;
                    wakeup = false; // 已经有线程唤醒，所以不用再唤醒
            }
        }
        if (STATE_UPDATER.compareAndSet(this, oldState, newState)) {
            break;  // 保证只有一个线程将oldState修改为newState
        }
```



```
R已被修改，则在下一次循环返回

新这两个值
eriod = unit.toNanos(quietPeriod);
t = unit.toNanos(timeout);

    if (oldState == ST_NOT_STARTED) {
        thread.start();
    }
    if (wakeup) {
        wakeup(inEventLoop);
    }
    return terminationFuture();
}
```

这段代码真是为多线程同时调用关闭的情况操碎了心，我们抓住其中的关键点：该方法只是将线程状态修改为ST_SHUTTING_DOWN并不执行具体的关闭操作（类似的shutdown方法将线程状态修改为ST_SHUTDOWN）。for()循环是为了保证修改state的线程（原生线程或者外部线程）有且只有一个。如果你还没有理解这句话，请查阅compareAndSet()方法的说明然后再看一遍。39-44行代码之所以这样处理，是因为子类的实现中run()方法是一个EventLoop即一个循环。40行代码启动线程可以完整走一遍正常流程并且可以处理添加到队列中的任务以及IO事件。43行唤醒阻塞在阻塞点上的线程，使其从阻塞状态退出。要从一个EventLoop循环中退出，有什么好方法吗？可能你会想到这样处理：设置一个标记，每次循环都检测这个标记，如果标记为真就退出。Netty正是使用这种方法，NioEventLoop的run()方法的循环部分有这样一段代码：

```
if (isShuttingDown()) { // 检测线程状态
    closeAll(); // 关闭注册的channel
    if (confirmShutdown()) {
        break;
    }
}
```

查询线程状态的方法有三个，实现简单，一并列出：

```
public boolean isShuttingDown() {
    return STATE_UPDATER.get(this) >= ST_SHUTTING_DOWN;
}

public boolean isShutdown() {
```

```
    return STATE_UPDATER.get(this) >= ST_SHUTDOWN;
}

public boolean isTerminated() {
    return STATE_UPDATER.get(this) == ST_TERMINATED;
}
```

需要注意的是调用shutdownGracefully()方法后线程状态为ST_SHUTTING_DOWN，调用shutdown()方法后线程状态为ST_SHUTDOWN。isShuttingDown()可以一并判断这两种调用方法。closeAll()方法关闭注册到NioEventLoop的所有Channel，代码不再列出。confirmShutdown()方法在SingleThreadEventExecutor类，确定是否可以关闭或者说是否可以从EventLoop循环中跳出。代码如下：

```
protected boolean confirmShutdown() {
    if (!isShuttingDown()) {
        return false;   // 没有调用shutdown相关的方法直接返回
    }
                                // 必须是原生线程
                                tateException("must be invoked from an event loop");


                                ; // 取消调度任务
    if (gracefulShutdownStartTime == 0) {   // 优雅关闭开始时间，这也是一个标记
        gracefulShutdownStartTime = ScheduledFutureTask.nanoTime();
    }

    // 执行完普通任务或者没有普通任务时执行完shutdownHook任务
    if (runAllTasks() || runShutdownHooks()) {
        if (isShutdown()) {
            return true;    // 调用shutdown()方法直接退出
        }
        if (gracefulShutdownQuietPeriod == 0) {
            return true;    // 优雅关闭静默时间为0也直接退出
        }
        wakeup(true);   // 优雅关闭但有未执行任务，唤醒线程执行
        return false;
    }

    final long nanoTime = ScheduledFutureTask.nanoTime();
    // shutdown()方法调用直接返回，优雅关闭截止时间到也返回
    if (isShutdown() || nanoTime - gracefulShutdownStartTime > gracefulShutdownTimeout) {
        return true;
    }
    // 在静默期间每100ms唤醒线程执行期间提交的任务
    if (nanoTime - lastExecutionTime <= gracefulShutdownQuietPeriod) {
        wakeup(true);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // Ignore
        }
        return false;
    }
    // 静默时间内没有任务提交，可以优雅关闭，此时若用户又提交任务则不会被执行
```

文章目录

```
    return true;
}
```

我们总结一下，调用shutdown()方法从循环跳出的条件有：
(1).执行完普通任务
(2).没有普通任务，执行完shutdownHook任务
(3).既没有普通任务也没有shutdownHook任务
调用shutdownGracefully()方法从循环跳出的条件有：
(1).执行完普通任务且静默时间为0
(2).没有普通任务，执行完shutdownHook任务且静默时间为0
(3).静默期间没有任务提交
(4).优雅关闭截止时间已到
注意上面所列的条件之间是**或**的关系，也就是说满足任意一条就会从EventLoop循环中跳出。我们可以将静默时间看为一段观察期，在此期间如果没有任务执行，说明可以跳出循环；如果此期间有任务执行，执行完后立即进入下一个观察期继续观察；如果连续多个观察期一直有任务执行，那么截止时间到则跳出循环。我们看一下shutdownGracefully()的默认参数：


```
racefully() {
lly(2, 15, TimeUnit.SECONDS);
```

可知，Netty默认的shutdownGracefully()机制为：在2秒的静默时间内如果没有任务，则关闭；否则15秒截止时间到达时关闭。换句话说，在15秒时间段内，如果有超过2秒的时间段没有任务则关闭。至此，我们明白了从EvnetLoop循环中跳出的机制，最后，我们抵达终点站：线程结束机制。这一部分的代码实现在线程工厂的生成方法中：

```
thread = threadFactory.newThread(new Runnable() {
        @Override
        public void run() {
            boolean success = false;
            updateLastExecutionTime();
            try {
                SingleThreadEventExecutor.this.run();   // 模板方法，EventLoop实现
                success = true;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                for (;;) {
                    int oldState = STATE_UPDATER.get(SingleThreadEventExecutor.this);
                    // 用户调用了关闭的方法或者抛出异常
                    if (oldState >= ST_SHUTTING_DOWN || STATE_UPDATER.compareAndSet(
                            SingleThreadEventExecutor.this, oldState, ST_SHUTTING_DOWN)) {
                        break;  // 抛出异常也将状态置为ST_SHUTTING_DOWN
                    }
                }
                if (success && gracefulShutdownStartTime == 0) {
                    // time=0，说明confirmShutdown()方法没有调用，记录日志
                }

                try {
                    for (;;) {
                        // 抛出异常时，将普通任务和shutdownHook任务执行完毕
                        // 正常关闭时，结合前述的循环跳出条件
                        if (confirmShutdown()) {
                            break;
```

```
                        }
                    }
                } finally {
                    try {
                        cleanup();
                    } finally {
                        // 线程状态设置为ST_TERMINATED，线程终止
                        STATE_UPDATER.set(SingleThreadEventExecutor.this, ST_TERMINATED);
                        threadLock.release();
                        if (!taskQueue.isEmpty()) {
                            //  关闭时，任务队列中添加了任务，记录日志
                        }
                        terminationFuture.setSuccess(null); // 异步结果设置为成功
                    }
                }
            }
```

文章目录

方法run()时，须调用confirmShutdown()方法，不调用的话会有错误日志。25-31行的for()循环主要是对异常情况的处理，但同时也兼顾了正常调用关闭方法的情况。可以将抛出异常的情况视为静默时间为0的shutdownGracefully()方法，这样便于理解循环跳出条件。34行代码cleanup()的默认实现什么也不做，NioEventLoop覆盖了基类，实现关闭NioEventLoop持有的selector：

```
protected void cleanup() {
    try {
        selector.close();
    } catch (IOException e) {
        logger.warn("Failed to close a selector.", e);
    }
}
```

关于Netty优雅关闭的机制，还有最后一点细节，那就是runShutdownHooks()方法：

```
private boolean runShutdownHooks() {
    boolean ran = false;
    while (!shutdownHooks.isEmpty()) {
        // 使用copy是因为shutdwonHook任务中可以添加或删除shutdwonHook任务
        List<Runnable> copy = new ArrayList<Runnable>(shutdownHooks);
        shutdownHooks.clear();
        for (Runnable task: copy) {
            try {
                task.run();
            } catch (Throwable t) {
                logger.warn("Shutdown hook raised an exception.", t);
            } finally {
                ran = true;
            }
        }
    }
    if (ran) {
        lastExecutionTime = ScheduledFutureTask.nanoTime();
    }
```

```
    return ran;
}
```

此外，还有threadLock.release()方法，如果你还记得字段定义，threadLock是一个初始值为0的信号量。一个初值为0的信号量，当线程请求锁时只会阻塞，这有什么用呢？awaitTermination()方法揭晓答案，用来使其他线程阻塞等待原生线程关闭：

```
public boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException {
    // 由于tryAcquire()永远不会成功，所以必定阻塞timeout时间
    if (threadLock.tryAcquire(timeout, unit)) {
        threadLock.release();
    }
    return isTerminated();
}
```

文章目录