Netty优雅退出机制和原理

原创 2016-06-21 李林锋 InfoQ

> 在实际项目中，Netty作为高性能的异步NIO通信框架，往往用作基础通信框架负责各种协议的接入、解析和调度等，当应用进程优雅退出时，作为通信框架的Netty也需要优雅退出。为什么？怎么做？

老司机简介

李林锋，2007年毕业于东北大学，2008年进入华为公司从事电信软件的设计和开发工作，有多年Java NIO、平台中间件设计和开发经验，精通Netty、Mina、分布式服务框架等,《Netty权威指南》、《分布式服务框架原理与实践》作者。目前从事云平台相关的架构和设计工作。

进程的优雅退出

1、Kill -9 PID带来的问题

在Linux上通常会通过kill -9 pid的方式强制将某个进程杀掉，这种方式简单高效，因此很多程序的停止脚本经常会选择使用kill -9 pid的方式。

无论是Linux的Kill -9 pid还是windows的taskkill /f /pid强制进程退出,都会带来一些副作用：对应用软件而言其效果等同于突然掉电，可能会导致如下一些问题：

1. 缓存中的数据尚未持久化到磁盘中，导致数据丢失；
2. 正在进行文件的write操作，没有更新完成，突然退出，导致文件损坏；
3. 线程的消息队列中尚有接收到的请求消息还没来得及处理，导致请求消息丢失；
4. 数据库操作已经完成，例如账户余额更新，准备返回应答消息给客户端时，消息尚在通信线程的发送队列中排队等待发送，进程强制退出导致应答消息没有返回给客户端，客户端发起超时重试，会带来重复更新问题；
5. 其它问题等...

2、JAVA优雅退出

Java的优雅停机通常通过注册JDK的ShutdownHook来实现，当系统接收到退出指令后，首先标记系统处于退出状态，不再接收新的消息，然后将积压的消息处理完，最后调用资源回收接口将资源销毁，最后各线程退出执行。

通常优雅退出需要有超时控制机制，例如30S，如果到达超时时间仍然没有完成退出前的资源回收等操作，则由停机脚本直接调用kill -9 pid，强制退出。

如何实现Netty的优雅退出

要实现Netty的优雅退出，首先需要了解通用Java进程的优雅退出如何实现。下面我们先讲解下优雅退出的实现原理，并结合实际代码进行讲解。最后看下如何实现Netty的优雅退出。

1、信号简介

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的，它是进程间一种异步通信的机制。以Linux的kill命令为例，kill -s SIGKILL pid (即kill -9 pid) 立即杀死指定pid的进程，SIGKILL就是发送给pid进程的信号。

信号具有平台相关性，Linux平台支持的一些终止进程信号如下所示：

信号名称	用途
SIGKILL	终止进程，强制杀死进程
SIGTERM	终止进程，软件终止信号
SIGTSTP	停止进程，终端来的停止信号

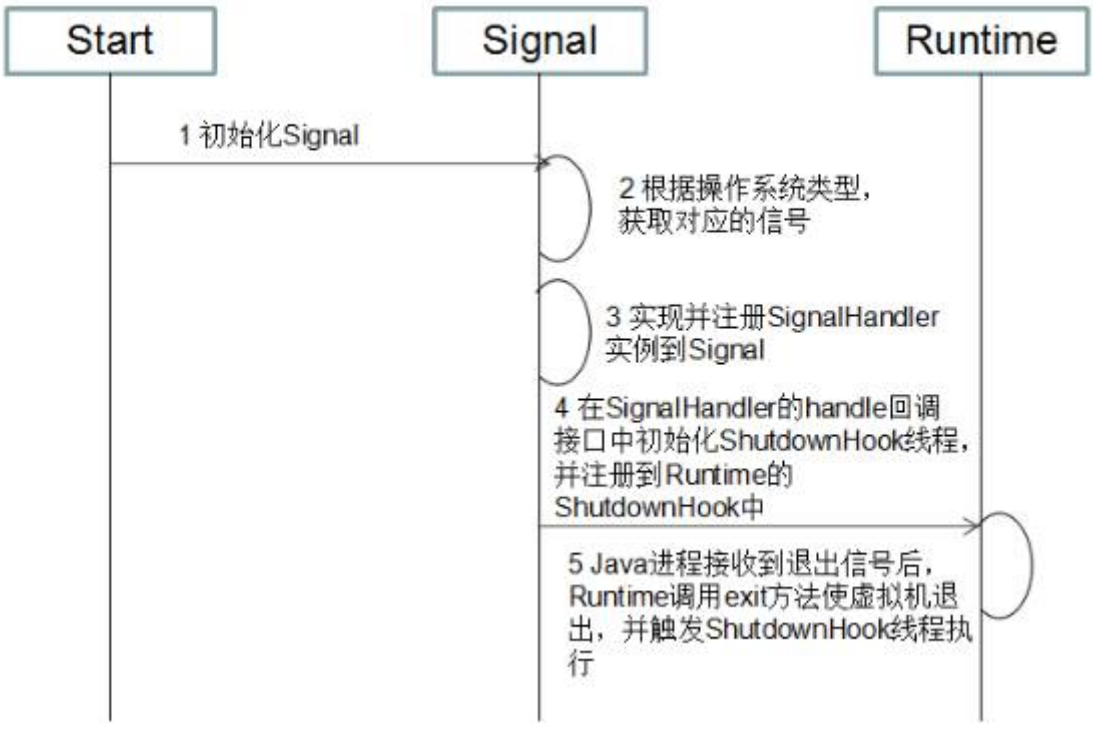
SIGPROF	终止进程，统计分布图用计时器到时
SIGUSR1	终止进程，用户定义信号1
SIGUSR2	终止进程，用户定义信号2
SIGINT	终止进程，中断进程
SIGQUIT	建立CORE文件终止进程，并且生成core文件

Windows平台存在一些差异，它的一些信号举例如下：SIGINT（Ctrl+C中断）、SIGILL、SIGTERM （kill发出的软件终止）、SIGBREAK （Ctrl+Break中断）。

信号选择：为了不干扰正常信号的运作，又能模拟Java异步通知，在Linux上我们需要先选定一种特殊的信号。通过查看信号列表上的描述，发现 SIGUSR1 和 SIGUSR2 是允许用户自定义的信号，我们可以选择SIGUSR2，为了测试方便，在Windows上我们可以选择SIGINT。

2、Java程序的优雅退出

首先看下通用的Java进程优雅退出的流程图：



第一步，应用进程启动的时候，初始化Signal实例，它的代码示例如下：

```
Signal sig = new Signal(getOSSignalType());
```

其中Signal构造函数的参数为String字符串，也就是2.1.1小节中介绍的信号量名称。

第二步，根据操作系统的名称来获取对应的信号名称，代码如下：

```
private String getOSSignalType()
    {
          return System.getProperties().getProperty("os.name").
                 toLowerCase().startsWith("win") ? "INT" : "USR2";
     }
```

判断是否是windows操作系统，如果是则选择SIGINT，接收Ctrl+C中断的指令；否则选择USR2信号，接收SIGUSR2（等价于kill -12 pid）指令。

第三步，将实例化之后的SignalHandler注册到JDK的Signal，一旦Java进程接收到kill -12 或者Ctrl+C则回调handle接口，代码示例如下：

Signal.handle(sig, shutdownHandler);

其中shutdownHandler实现了SignalHandler接口的handle(Signal sgin)方法，代码示例如下：

```
/**
 * @author 李林峰
 *
 */
public class SystemShutdown implements SignalHandler {

    /* (non-Javadoc)
     * @see sun.misc.SignalHandler#handle(sun.misc.Signal)
     */
    @Override
    public void handle(Signal sgin) {
```

第四步，在接收到信号回调的handle接口中，初始化JDK的ShutdownHook线程，并将其注册到Runtime中，示例代码如下：

```
private void invokeShutdownHook()
 {
        Thread t = new Thread(new ShutdownHook(), "ShutdownHook-Thread");
        Runtime.getRuntime().addShutdownHook(t);
 }
```

第五步，接收到进程退出信号后，在回调的handle接口中执行虚拟机的退出操作，示例代码如下：

```
Runtime.getRuntime().exit(0);
```

虚拟机退出时，底层会自动检测用户是否注册了ShutdownHook任务，如果有，则会自动将ShutdownHook线程拉起，执行它的Run方法，用户只需要在ShutdownHook中执行资源释放操作即可，示例代码如下：

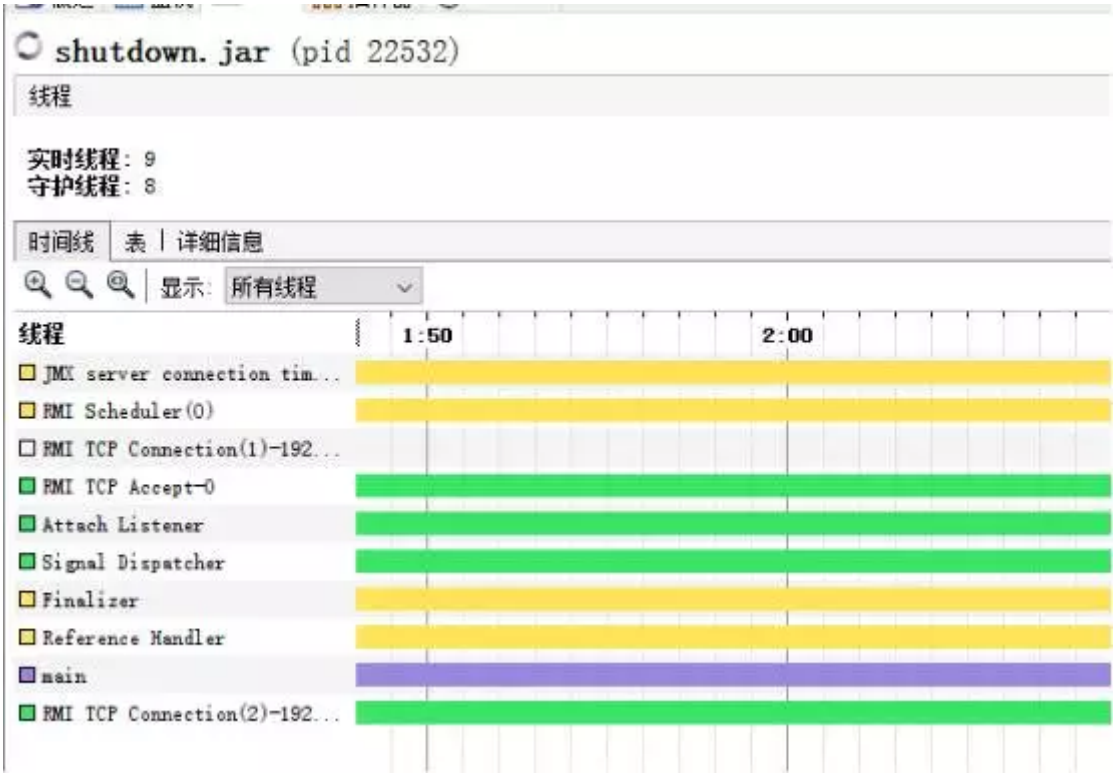
```
class ShutdownHook implements Runnable
{
        @Override
        public void run() {
                System.out.println("ShutdownHook execute start...");
                System.out.print("Netty NioEventLoopGroup shutdownGracefully...");
                try {
                        TimeUnit.SECONDS.sleep(10);//模拟应用进程退出前的处理操作
                } catch (InterruptedException e) {
                                e.printStackTrace();
                }
                System.out.println("ShutdownHook execute end...");
        System.out.println("Sytem shutdown over, the cost time is 10000MS");
                }
}
```

下面我们在Windows环境中对通用的Java优雅退出程序进行测试，打开CMD控制台，拉起待测试程序，如下所示：

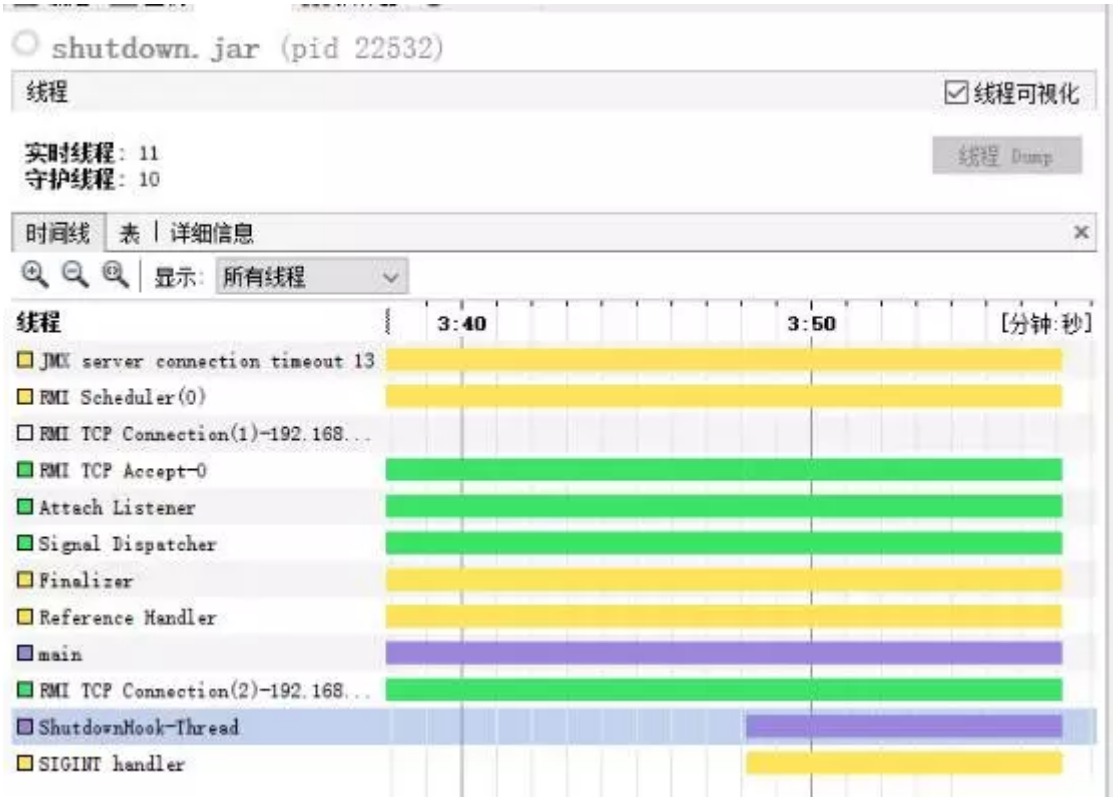
启动进程：

```
C:\SE\eclipse\workspace\netty-5.0.0\classes\io\netty\example\shutdown\signal>jav
a -jar shutdown.jar
```

查看线程信息，发现注册的ShutdownHook线程没有启动，符合预期：



在控制台执行Ctrl+C，使进程退出，示例如下：



如上图所示，我们定义的ShutdownHook线程在JVM退出时被执行，作为测试程序，它休眠10S之后退出，控制台打印的相关信息如下：

```
nal>java -jar shutdown.jar
Receive signal is : INT | number is : 2
ShutdownHook execute start...
Netty NioEventLoopGroup shutdownGracefully...
ShutdownHook execute end...
Sytem shutdown over, the cost time is 10000MS

C:\SE\eclipse\workspace\netty-5.0.0\classes\io\netty\
```

下面我们总结下通用的Java程序优雅退出的技术要点：

3、Netty的优雅退出

在实际项目中，Netty作为高性能的异步NIO通信框架，往往用作基础通信框架负责各种协议的接入、解析和调度等，例如在RPC和分布式服务框架中，往往会使用Netty作为内部私有协议的基础通信框架。

当应用进程优雅退出时，作为通信框架的Netty也需要优雅退出，主要原因如下：

1. 尽快的释放NIO线程、句柄等资源；
2. 如果使用flush做批量消息发送，需要将积攒在发送队列中的待发送消息发送完成；
3. 正在write或者read的消息，需要继续处理；
4. 设置在NioEventLoop线程调度器中的定时任务，需要执行或者清理。

下面我们看下Netty优雅退出涉及的主要操作和资源对象：

Netty的优雅退出总结起来有三大步操作：

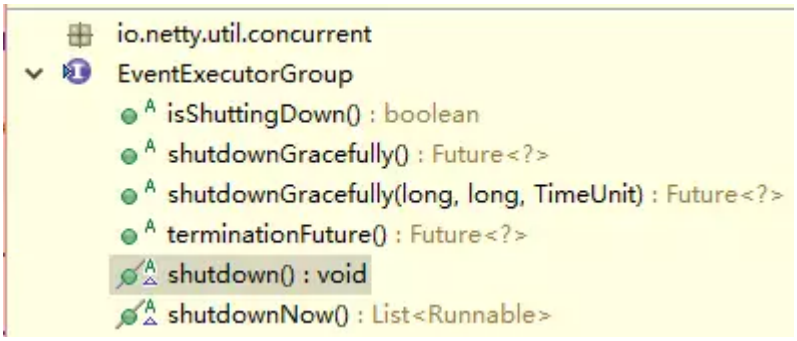
1. 把NIO线程的状态位设置成ST_SHUTTING_DOWN状态，不再处理新的消息（不允许再对外发送消息）；
2. 退出前的预处理操作：把发送队列中尚未发送或者正在发送的消息发送完、把已经到期或者在退出超时之前到期的定时任务执行完成、把用户注册到NIO线程的退出Hook任务执行完成；
3. 资源的释放操作：所有Channel的释放、多路复用器的去注册和关闭、所有队列和定时任务的清空取消，最后是NIO线程的退出。

下面我们具体看下如何实现Netty的优雅退出：

Netty优雅退出的接口和总入口在EventLoopGroup，调用它的shutdownGracefully方法即可，相关代码如下：

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

除了无参的shutdownGracefully方法，还可以指定退出的超时时间和周期，相关接口定义如下：



EventLoopGroup的shutdownGracefully工作原理下个章节做详细讲解，结合Java通用的优雅退出机制，即可实现Netty的优雅退出，相关伪代码如下：

```
//统一定义JVM退出事件，并将JVM退出事件作为主题对进程内部发布
//所有需要优雅退出的消费者订阅JVM退出事件主题
```

```
//监听JVM退出的ShutdownHook被启动之后，发布JVM退出事件
//消费者监听到JVM退出事件，开始执行自身的优雅退出
//如果所有的非守护线程都成功完成优雅退出，进程主动退出
//如果到了退出的超时时间仍然没正常退出，则由停机脚本通过kill -9 pid强杀进程，强制退出
```

总结一下：JVM的ShutdownHook被触发之后，调用所有EventLoopGroup实例的shutdownGracefully方法进行优雅退出。由于Netty自身对优雅退出有较完善的支持，所以实现起来相对比较简单。

4、 一些误区

在实际工作中，由于对优雅退出和资源释放的原理不太清楚，或者对Netty的接口不太了解，很容易把优雅退出和资源释放混淆，导致出现各种问题。

如下案例：本意是想把某个Channel关闭，但是却调用了Channel关联的EventLoop的shutdownGracefully，导致把EventLoop线程和注册在该线程持有的多路复用器上所有的Channel都关闭了，错误代码如下所示：

```
ctx.channel().eventLoop().shutdownGracefully();
```

正确的做法如下所示：调用channel的close方法，关闭链路，释放与该Channel相关的资源：

```
ctx.channel().close();
```

除非是整个进程优雅退出，一般情况下不会调用EventLoopGroup和EventLoop的shutdownGracefully方法，更多的是链路channel的关闭和资源释放。

Netty优雅退出原理分析

Netty优雅退出涉及到线程组、线程、链路、定时任务等，底层实现细节非常复杂，下面我们就层层分解，通过源码来剖析它的实现原理。

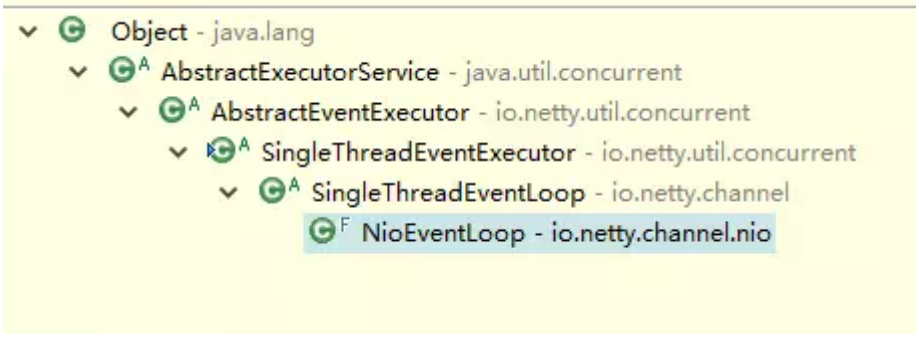
1、NioEventLoopGroup

NioEventLoopGroup实际是NioEventLoop的线程组，它的优雅退出比较简单，直接遍历EventLoop数组，循环调用它们的shutdownGracefully方法，源码如下：

```
@Override
public Future<?> shutdownGracefully(long quietPeriod, long timeout,
        TimeUnit unit) {
    for (EventExecutor l: children) {
        l.shutdownGracefully(quietPeriod, timeout, unit);
    }
    return terminationFuture();
}
```

2、NioEventLoop

调用NioEventLoop的shutdownGracefully方法，首先就是要修改线程状态为正在关闭状态，它的实现在父类SingleThreadEventExecutor中，它们的继承关系如下：



SingleThreadEventExecutor的shutdownGracefully代码比较简单，就是修改线程的状态位，需要注意的是修改时需要对并发调用做判断，如果是由NioEventLoop自身调用，则不需要加锁，否则需要加锁，代码如下：

```
synchronized (stateLock) {
    if (isShuttingDown()) {
        return terminationFuture();
    }

    gracefulShutdownQuietPeriod = unit.toNanos(quietPeriod);
    gracefulShutdownTimeout = unit.toNanos(timeout);

    if (inEventLoop) {
        assert state == ST_STARTED;
        state = ST_SHUTTING_DOWN;
```

解释下为什么要加锁，因为shutdownGracefully是public的方法，任何能够获取到NioEventLoop的代码都可以调用它，在Netty中，业务代码通常不需要直接获取NioEventLoop并操作它，但是Netty对NioEventLoop做了比较厚的封装，它不仅仅只能读写消息，还能够执行定时任务，并作为线程池执行用户自定义Task。因此在Channel中将获取NioEventLoop的方法开放了出来，这就意味着用户只要能够获取到Channel，理论上就会存在并发执行shutdownGracefully的可能，因此在优雅退出的时候做了并发保护。

完成状态修改之后，剩下的操作主要在NioEventLoop中进行，代码如下：

```
if (isShuttingDown()) {
    closeAll();
    if (confirmShutdown()) {
        break;
    }
}
```

我们继续看下closeAll的实现，它的原理是把注册在selector上的所有Channel都关闭，但是有些Channel正在发送消息，暂时还不能关，需要稍后再执行，核心代码如下：

```
Set<SelectionKey> keys = selector.keys();
Collection<AbstractNioChannel> channels = new ArrayList<AbstractNioChannel>
for (SelectionKey k: keys) {
    Object a = k.attachment();
    if (a instanceof AbstractNioChannel) {
        channels.add((AbstractNioChannel) a);
    } else {
        k.cancel();
        @SuppressWarnings("unchecked")
        NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
        invokeChannelUnregistered(task, k, null);
    }
}

for (AbstractNioChannel ch: channels) {
    ch.unsafe().close(ch.unsafe().voidPromise());
}
```

循环调用Channel Unsafe的close方法，下面我们跳转到Unsafe中，对close方法进行分析。

3、AbstractUnsafe

AbstractUnsafe的close方法主要做了如下几件事：

1．判断当前该链路是否有消息正在发送，如果有则将关闭操作封装成Task放到eventLoop中稍后再执行：

```
public final void close(final ChannelPromise promise) {
    if (inFlush0) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                close(promise);
            }
        });
        return;
    }
```

2．将发送队列清空，不再允许发送新的消息：

```
boolean wasActive = isActive();
ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
this.outboundBuffer = null; // Disallow adding any messages
```

3．调用SocketChannel的close方法，关闭链路：

```
@Override
protected void doClose() throws Exception {
    javaChannel().close();
}
```

4．调用pipeline的fireChannelInactive，触发链路关闭通知事件：

```
if (wasActive && !isActive()) {
    invokeLater(new Runnable() {
        @Override
        public void run() {
            pipeline.fireChannelInactive();
        }
    });
}
```

5．最后是调用deregister，从多路复用器上取消SelectionKey：

```
@Override
protected void doDeregister() throws Exception {
    eventLoop().cancel(selectionKey());
}
```

至此，优雅退出流程已经完成，这是否意味着NioEventLoop线程可以退出了，其实并非如此。

在此处，只是做了Channel的关闭和从Selector上的去注册，总结如下：

1. 通过inFlush0来判断当前是否正在发送消息，如果是，则不执行Channel关闭动作，放入NIO线程的任务队列中稍后再执行close()操作；
2. 因为已经不允许新的发送消息加入，一旦发送操作完成，就执行链路关闭、触发链路关闭事件和从Selector上取消注册操作。

之前已经说了，NioEventLoop除了I/O读写之外，还兼具定时任务执行、关闭ShutdownHook的执行等，如果此时有到期的定时任务，即使Chanel已经关闭，但是仍然需要继续执行，线程不能

退出。下面我们具体分析下TaskQueue的处理流程。

4、TaskQueue

NioEventLoop执行完closeAll（）操作之后，需要调用confirmShutdown看是否真的能够退出，它的处理逻辑如下：

1．执行TaskQueue中排队的Task，代码如下：

```
protected boolean runAllTasks() {
    fetchFromDelayedQueue();
    Runnable task = pollTask();
    if (task == null) {
        return false;
    }

    for (;;) {
        try {
            task.run();
        } catch (Throwable t) {
            logger.warn("A task raised an exception.", t);
        }

        task = pollTask();
        if (task == null) {
            lastExecutionTime = ScheduledFutureTask.nanoTime();
            return true;
        }
    }
}
```

2．执行注册到NioEventLoop中的ShutdownHook，代码如下：

```
private boolean runShutdownHooks() {
    boolean ran = false;
    // Note shutdown hooks can add / remove shutdown hooks.
    while (!shutdownHooks.isEmpty()) {
        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;
            }
        }
    }
```

3．判断是否到达优雅退出的指定超时时间，如果达到或者过了超时时间，则立即退出，代码如下：

```
if (isShutdown() || nanoTime - gracefulShutdownStartTime > gracefulShutdownTimeout) {
    return true;
}
```

4．如果没到达指定的超时时间，暂时不退出，每隔100MS检测下是否有新的任务加入，有则继续执行：

```
if (nanoTime - lastExecutionTime <= gracefulShutdownQuietPeriod) {
    // Check if any tasks were added to the queue every 100ms.
    // TODO: Change the behavior of takeTask() so that it returns
    wakeup(true);
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        // Ignore
    }

    return false;
}
```

在confirmShutdown方法中，夹杂了一些对已经废弃的shutdown（ ）方法的处理，例如：

```
protected boolean confirmShutdown() {
    if (!isShuttingDown()) {
        return false;
    }
```

调用新的shutdownGracefully系列方法，该判断条件是永远都不会成立的，因此对于已经废弃的shutdown相关的处理逻辑，不再详细分析。

到此为止，confirmShutdown方法讲解完毕，confirmShutdown返回true，则NioEventLoop线程正式退出，Netty的优雅退出完成，代码如下：

```
if (confirmShutdown()) {
    break;
}
```

5、疑问解答

runAllTasks重复执行问题

在NioEventLoop的run方法中，已经调用了runAllTasks方法，为何紧随其后，在confirmShutdown中有继续调用runAllTasks方法呢，疑问代码如下：

```
runAllTasks(ioTime * (100 - ioRatio) / ioRatio);

if (isShuttingDown()) {
    closeAll();
    if (confirmShutdown()) {
        break;
    }
}
```

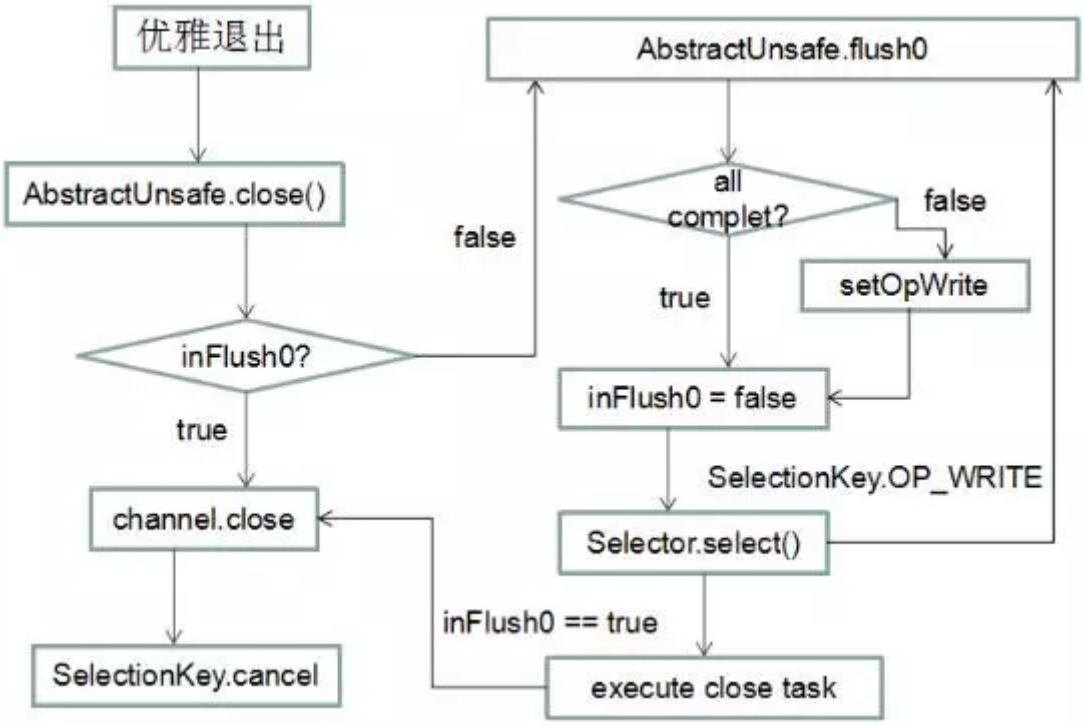
原因主要有两个：

1. 为了防止定时任务Task或者用户自定义的线程Task的执行过多占用NioEventLoop线程的调度资源，Netty对NioEventLoop线程I/O操作和非I/O操作时间做了比例限制，即限制非I/O操作的执行时间，如上图红框中代码所示。有了执行时间限制，因此可能会导致已经到期的定时任务、普通任务没有执行完，需要等待下次Selector轮询继续执行。在线程退出之前，需要对本该执行但是没有执行完成的Task进行扫尾处理，所以在confirmShutdown中再次调用了runAllTasks方法；

2. 在调用runAllTasks方法之后，执行confirmShutdown之前，用户向NioEventLoop中添加了新的普通任务或者定时任务，因此需要在退出之前再次遍历并处理一遍Task Queue。

优雅退出是否能够保证所有在通信线程排队的消息全部发送出去

实际是无法保证的，它只能保证如果现在正在发送消息过程中，调用了优雅退出方法，此时不会关闭链路，继续发送，如果发送操作完成，无论是否还有消息尚未发送出去，在下一轮Selector的轮询中，链路将会关闭，没有发送完成的消息将会被丢弃，甚至是半包消息。它的处理原理图如下：



它的原理比较复杂，现对主要逻辑处理进行解读：

1. 调用优雅退出之后，是否关闭链路，判断标准是inFlush0是否为true，如果为False，则会执行链路关闭操作；
2. 如果用户是类似批量发送，例如每达到N条或者定时触发flush操作，则在此期间调用优雅退出方法，inFlush0为False，链路关闭，积压的待发送消息会被丢弃掉；
3. 如果优雅退出时链路正好在发送消息过程中，则它不会立即退出，等待发送完成之后，下次Selector轮询的时候才退出。在这种场景下，又有两种可能的场景：

场景A：如果一次把积压的消息全部发送完，没有发生写半包，则不会发生消息丢失；

场景B：如果一次没有把消息发送完成，此时Netty会监听写事件，触发Selector的下一次轮询并发送消息，代码如下：

```
protected final void setOpWrite() {
    final SelectionKey key = selectionKey();
    final int interestOps = key.interestOps();
    if ((interestOps & SelectionKey.OP_WRITE) == 0) {
        key.interestOps(interestOps | SelectionKey.OP_WRITE);
    }
}
```

Selector轮询时，首先处理读写事件，然后再处理定时任务和普通任务，因此在链路关闭之前，还有最后一次继续发送的机会，代码如下：

```
if (selectedKeys != null) {
    processSelectedKeysOptimized(selectedKeys.flip());
} else {
    processSelectedKeysPlain(selector.selectedKeys());
}
final long ioTime = System.nanoTime() - ioStartTime;

final int ioRatio = this.ioRatio;
runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
```

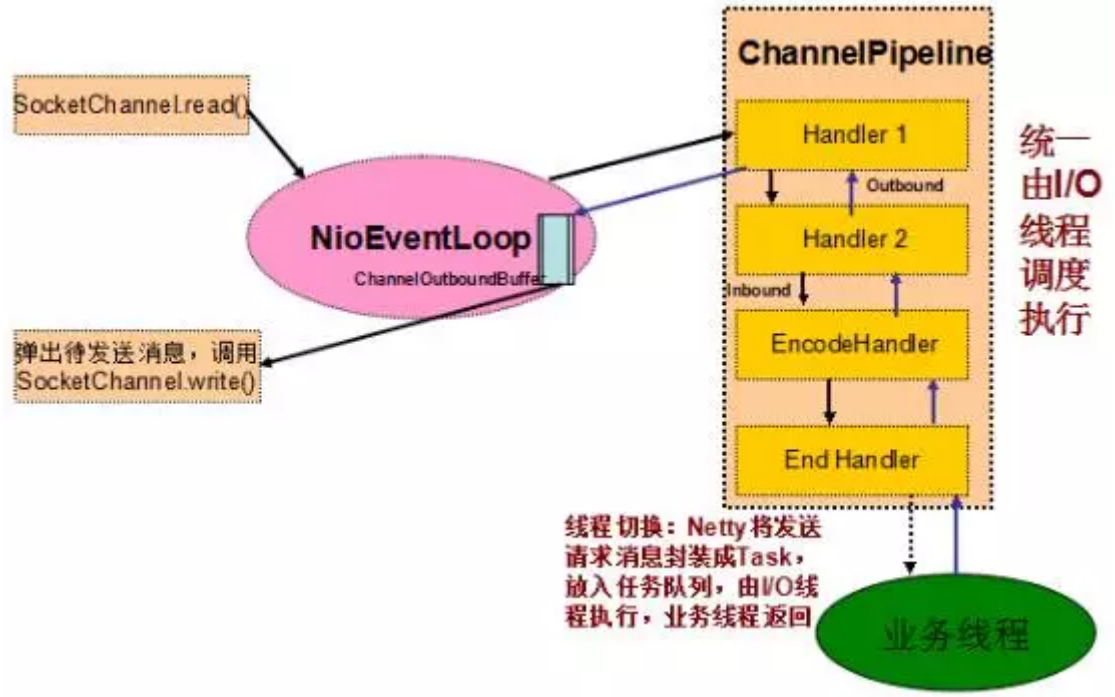
如果非常不幸，再次发送仍然没有把积压的消息全部发送完毕，再次发生了写半包，那无论是否有积压消息，执行AbstractUnsafe.close的Task还是会把链路给关闭掉，原因是只要完成一次消息发送操作，Netty就会把inFlush0置为false，代码如下：

```
try {
    doWrite(outboundBuffer);
} catch (Throwable t) {
    outboundBuffer.failFlushed(t);
} finally {
    inFlush0 = false;
}
```

链路关闭之后，所有尚未发送的消息都将被丢弃。

可能有些读者会有疑问，如果在第二次发送之后，执行AbstractUnsafe.close之前，业务正好又调用了flush操作，inFlush0是否会被修改成True呢？这个是不可能的，因为从Netty 4.X之后线程模型发生了变更，flush操作不是由用户线程执行，而是由Channel对应的NioEventLoop线程执行，所以在两者之间不会发生inFlush0被修改的情况。

Netty 4.X之后的线程模型如下所示：



另外，由于优雅退出有超时时间，如果在超时时间内没有完成积压消息的发送，也会发生消息丢弃的情况。

对于上述场景，需要应用层来保证相关的可靠性，或者对Netty的优雅退出机制进行优化。

- **InfoQ大咖说直播预告：**

▽

延展阅读（点击标题）：

- 京东618：从演习、监控到预案，京东无线全面备战

- Twitter开源软件项目列表
- 保持简单：Uber流处理架构演进的四字箴言 | 附124页PPT下载

本文系InfoQ原创首发，未经授权谢绝转载。

长按二维码识别关注InfoQ