16 Netty服务如何优雅关闭

更新时间：2020-07-30 10:27:50

> 不安于小成，然后足以成大器；不诱于小利，然后可以立远功。——方孝孺

前言

你好，我是彤哥。

上一节，我们一起学习了服务写出数据的源码剖析，我们发现，Netty 中将写出数据分成了两个部分，第一部分先缓存起来，第二部分再通过 Java 原生的 SocketChannel 发送出去。

今天，我们再来学习一个新的概念 —— 如何优雅地关闭服务。

好了，开始今天的学习吧。

问题

在 Netty 的编码模板中，我们一般会在 try...finally... 中写上这么一段代码：

```
try {
    // 省略其他代码
    // 9. 等待服务端监听端口关闭，这里会阻塞主线程
    f.channel().closeFuture().sync();
} finally {
    // 10. 优雅地关闭两个线程池
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
```

通过 NioEventLoopGroup 的 shutdownGracefully () 来关闭服务器，那么，今天的问题来了：

1. shutdownGracefully () 内部的逻辑是怎样的？
2. 进行了哪些资源的释放？
3. NioEventLoopGroup 如何知道所有的 EventLoop 都优雅关闭了呢？

好了，让我们带着这些问题进入今天的探索吧。

优雅关闭服务过程

正常来说，服务是不会走到第 10 步的，除非出现异常，因为第 9 步的 sync () 会阻塞 main 线程。

所以，我们这里调试的时候可以先把第 9 步注释掉，让程序能够走到第 10 步，这样便于我们调试。

假设我们已经这样做了，程序断点在了 `bossGroup.shutdownGracefully();` 这一行，让我们跟踪到这个方法内部看看：

```
// io.netty.util.concurrent.AbstractEventExecutorGroup#shutdownGracefully
@Override
public Future<?> shutdownGracefully() {
    // 调用重载方法
    // 第一个参数为静默周期，默认2秒
    // 第二个参数为超时时间，默认15秒
    return shutdownGracefully(DEFAULT_SHUTDOWN_QUIET_PERIOD, DEFAULT_SHUTDOWN_TIMEOUT, TimeUnit.SECONDS);
}
// io.netty.util.concurrent.MultithreadEventExecutorGroup#shutdownGracefully
@Override
public Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit) {
    for (EventExecutor l: children) {
        // 调用孩子的shutdownGracefully()
        // 这里的EventExecutor无疑就是NioEventLoop
        l.shutdownGracefully(quietPeriod, timeout, unit);
    }
    // 返回的是NioEventLoopGroup的terminationFuture
    return terminationFuture();
}
```

可以看到，内部其实是调用了每个 NioEventLoop 的 shutdownGracefully () 方法，最后返回了 NioEventLoopGroup 的 terminationFuture。

我们先看看 NioEventLoop 的 shutdownGracefully () 方法：

```
// io.netty.util.concurrent.SingleThreadEventExecutor#shutdownGracefully
@Override
public Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit) {
    // 参数检验
    ObjectUtil.checkPositiveOrZero(quietPeriod, "quietPeriod");
    if (timeout < quietPeriod) {
        throw new IllegalArgumentException(
            "timeout: " + timeout + " (expected >= quietPeriod (" + quietPeriod + "))");
    }
    ObjectUtil.checkNotNull(unit, "unit");

    // 其它线程正在执行关闭，直接返回
    if (isShuttingDown()) {
        return terminationFuture();
    }

    boolean inEventLoop = inEventLoop();
    boolean wakeup;
```

```
    int oldState;
    for (;;) {
        // 再次检查其它线程正在执行关闭，直接返回
        if (isShuttingDown()) {
            return terminationFuture();
        }
        int newState;
        wakeup = true;
        oldState = state;
        // 我们是在main线程中执行的
        // 所以不在EventLoop中
        if (inEventLoop) {
            newState = ST_SHUTTING_DOWN;
        } else {
            switch (oldState) {
                case ST_NOT_STARTED:
                case ST_STARTED:
                    // 显然，我们的程序已经启动了
                    // 所以，新状态为ST_SHUTTING_DOWN
                    newState = ST_SHUTTING_DOWN;
                    break;
                default:
                    newState = oldState;
                    wakeup = false;
            }
        }
        // key1，更新状态成功，退出循环
        if (STATE_UPDATER.compareAndSet(this, oldState, newState)) {
            break;
        }
    }
    // key2，修改NioEventLoop的属性标识
    gracefulShutdownQuietPeriod = unit.toNanos(quietPeriod);
    gracefulShutdownTimeout = unit.toNanos(timeout);

    if (ensureThreadStarted(oldState)) {
        return terminationFuture;
    }

    // key3，添加一个空任务，唤醒EventLoop
    if (wakeup) {
        taskQueue.offer(WAKEUP_TASK);
        if (!addTaskWakesUp) {
            wakeup(inEventLoop);
        }
    }

    // key4，返回NioEventLoop的terminationFuture
    return terminationFuture();
}
// io.netty.channel.nio.NioEventLoop#wakeup
@Override
protected void wakeup(boolean inEventLoop) {
    if (!inEventLoop && nextWakeupNanos.getAndSet(AWAKE) != AWAKE) {
        selector.wakeup();
    }
}
```

这个方法就结束了，让我们整理下这个方法的主要逻辑：

- 修改状态为 ST_SHUTTING_DOWN；
- 修改两个属性，一个是静默周期，一个是超时时间；
- 往队列中放了一个空任务，并唤醒 NioEventLoop，实际上是唤醒的 selector，也就是 selector.select () 的位置；
- 返回 NioEventLoop 的 terminationFuture，这个 terminationFuture 怎么跟 NioEventLoopGroup 的

terminationFuture 联系起来的呢？或者说，NioEventLoopGroup 怎么知道所有的 NioEventLoop 都关闭成功了呢？

可以看到，这个方法中并没有对 “优雅关闭” 做什么实质的处理，那么，真正关闭的处理逻辑在哪里呢？

经过前面的学习，我们知道，NioEventLoop 相当于一个线程，它的 run () 方法中有对 selector 的轮询，而这里又唤醒了 selector，所以，我们大胆猜测，处理逻辑应该是在 NioEventLoop 的 run () 方法中，让我们再仔细研究一下这个 run () 方法：

```
@Override
protected void run() {
    int selectCnt = 0;
    // 轮询
    for (;;) {
        try {
            int strategy;
            try {
                // 策略，这里面会检测如果有任务，调用的是selectNow()，也就是不阻塞
                strategy = selectStrategy.calculateStrategy(selectNowSupplier, hasTasks());
                switch (strategy) {
                    case SelectStrategy.CONTINUE:
                        continue;
                    case SelectStrategy.BUSY_WAIT:
                    case SelectStrategy.SELECT:
                        long curDeadlineNanos = nextScheduledTaskDeadlineNanos();
                        if (curDeadlineNanos == -1L) {
                            curDeadlineNanos = NONE; // nothing on the calendar
                        }
                        nextWakeupNanos.set(curDeadlineNanos);
                        try {
                            if (!hasTasks()) {
                                // 如果没有任务，才会调用这里的select()，默认是阻塞的
                                // 通过前面的唤醒，唤醒的是这里的select()
                                strategy = select(curDeadlineNanos);
                            }
                        } finally {
                            nextWakeupNanos.lazySet(AWAKE);
                        }
                    default:
                }

                // 省略其他内容
            }
            try {
                // 判断是否处于关闭中
                if (isShuttingDown()) {
                    // key1，关闭什么？
                    closeAll();
                    // key2，确定关闭什么？
                    if (confirmShutdown()) {
                        return;
                    }
                }
            } catch (Throwable t) {
                handleLoopException(t);
            }
        }
    }
```

针对优雅关闭的时候，肯定不能让 run () 方法阻塞在 select () 上，所以前面向队列中添加了一个空任务，当有任务的时候，它调用的就是 selectNow () 方法，selectNow () 方法不管有没有读取到任务都会直接返回，不会产生任何阻塞，最后，程序逻辑会走到 `isShuttingDown()` 这个判断的地方，这里有两个比较重要的方法：

- closeAll ()，关闭了什么？
- confirmShutdown ()，确定关闭了什么？

我们先来看看 `closeAll()` 这个方法，这里有个调试技巧，请看源码：

```
// io.netty.channel.nio.NioEventLoop#closeAll
private void closeAll() {
    // 再次调用selectNow()方法
    selectAgain();
    // selector中所有的SelectionKey
    Set<SelectionKey> keys = selector.keys();
    Collection<AbstractNioChannel> channels = new ArrayList<AbstractNioChannel>(keys.size());
    for (SelectionKey k: keys) {
        // 最好在这里打个断点，因为有些NioEventLoop里面是没有绑定Channel的，所以没有Channel需要关闭
        // 在这里打个断点，NioServerSocketChannel对应的NioEventLoop肯定会到这里
        // 这里取出来的附件就是NioServerSocketChannel
        Object a = k.attachment();
        if (a instanceof AbstractNioChannel) {
            // 把要关闭的Channel加到集合中
            channels.add((AbstractNioChannel) a);
        } else {
            // NioTask当前版本没有地方使用
            k.cancel();
            @SuppressWarnings("unchecked")
            NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
            invokeChannelUnregistered(task, k, null);
        }
    }

    // 遍历集合
    for (AbstractNioChannel ch: channels) {
        // key，调用Channel的unsafe进行关闭
        ch.unsafe().close(ch.unsafe().voidPromise());
    }
}
```

`closeAll()` 方法中主要就是对 Channel 进行关闭，这些 Channel 的关闭最后又是调用他们的 unsafe 进行关闭的，让我们跟踪到 unsafe 里面看看到底做了哪些操作：

```
// io.netty.channel.AbstractChannel.AbstractUnsafe#close
@Override
public final void close(final ChannelPromise promise) {
    assertEventLoop();

    ClosedChannelException closedChannelException = new ClosedChannelException();
    // 调用重载方法
    close(promise, closedChannelException, closedChannelException, false);
}
// io.netty.channel.AbstractChannel.AbstractUnsafe#close
private void close(final ChannelPromise promise, final Throwable cause,
                   final ClosedChannelException closeCause, final boolean notify) {
    if (!promise.setUncancellable()) {
        return;
    }

    // 使用closeInitiated防止重复关闭
    // closeInitiated初始值为false，所以这个大if可以跳过
    if (closeInitiated) {
        // 省略不相关代码
        return;
    }

    // 下面的逻辑只会执行一次
    closeInitiated = true;
```

```
    final boolean wasActive = isActive();
    // 写出数据时的缓存
    final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    // 置为空表示不允许再写出数据了
    this.outboundBuffer = null;
    // 默认为空
    Executor closeExecutor = prepareToClose();
    if (closeExecutor != null) {
        // 对于NioServerSocketChannel，默认为空，不会走到这，省略这段代码
    } else {
        try {
            // key，一看就是干正事的方法
            doClose0(promise);
        } finally {
            if (outboundBuffer != null) {
                // todo 未发送的数据将失败，为什么不让它们发送出去呢？
                outboundBuffer.failFlushed(cause, notify);
                outboundBuffer.close(closeCause);
            }
        }
        if (inFlush0) {
            invokeLater(new Runnable() {
                @Override
                public void run() {
                    // 触发channelInactive()和channelDeregister()方法
                    fireChannelInactiveAndDeregister(wasActive);
                }
            });
        } else {
            // 触发channelInactive()和channelDeregister()方法
            fireChannelInactiveAndDeregister(wasActive);
        }
    }
}
private void doClose0(ChannelPromise promise) {
    try {
        // 干正事的方法
        doClose();
        // 将closeFuture设置为已关闭
        closeFuture.setClosed();
        // 将promise设置为已成功
        safeSetSuccess(promise);
    } catch (Throwable t) {
        closeFuture.setClosed();
        safeSetFailure(promise, t);
    }
}
@Override
protected void doClose() throws Exception {
    // 关闭Java原生的Channel，我们这里为ServerSocketChannel
    javaChannel().close();
}
// io.netty.channel.nio.AbstractNioChannel#doDeregister
// 这个方法是在fireChannelInactiveAndDeregister()中调用的
@Override
protected void doDeregister() throws Exception {
    // 取消SelectionKey
    eventLoop().cancel(selectionKey());
}
```

总结一下，`close()` 的过程关闭了哪些资源：

- 写出数据的缓存置空，不允许再写出数据；
- 缓存中未发送的数据将失败；
- 关闭 Java 原生的 Channel；

- closeFuture 置为关闭状态；
- 取消 Channel 关联的 SelectionKey；
- 调用 channelInactive () 和 channelDeregister () 方法；

到这里，整个 `closeAll()` 就完了，我们再来看看 `confirmShutdown()`：

```
protected boolean confirmShutdown() {
    // 不是正在关闭，返回false
    if (!isShuttingDown()) {
        return false;
    }

    if (!inEventLoop()) {
        throw new IllegalStateException("must be invoked from an event loop");
    }

    // 取消定时任务
    cancelScheduledTasks();

    // 设置优雅关闭服务的开始时间
    if (gracefulShutdownStartTime == 0) {
        gracefulShutdownStartTime = ScheduledFutureTask.nanoTime();
    }

    // 运行所有任务和所有shudown的钩子任务
    if (runAllTasks() || runShutdownHooks()) {
        // 有任何一个任务运行，都会进入这里
        // 已经关闭了，返回true
        if (isShutdown()) {
            return true;
        }

        // 如果静默周期为0，返回true
        if (gracefulShutdownQuietPeriod == 0) {
            return true;
        }
        // 否则添加一个空任务，返回false
        taskQueue.offer(WAKEUP_TASK);
        return false;
    }

    // 没有任何任务运行
    final long nanoTime = ScheduledFutureTask.nanoTime();

    // 如果已经关闭，或者超时了，返回true
    if (isShutdown() || nanoTime - gracefulShutdownStartTime > gracefulShutdownTimeout) {
        return true;
    }

    // 如果当前时间减去上一次运行的时间在静默周期以内
    if (nanoTime - lastExecutionTime <= gracefulShutdownQuietPeriod) {
        // 添加一个空任务，并休眠100ms
        taskQueue.offer(WAKEUP_TASK);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // Ignore
        }
        // 返回false
        return false;
    }

    // 超过了静默周期，返回true
    return true;
}
```

这里首先要明确返回值的意义，请看下面简化版的 `run()` 方法，如果 `confirmShutdown()` 返回 true，将跳出循环，那么，这个 run () 方法也就结束了，如果返回 false，将重新执行这里面的逻辑，直到返回 true。

```
protected void run() {
    for(;;) {
        if (isShuttingDown()) {
            closeAll();
            if (confirmShutdown()) {
                return;
            }
        }
    }
}
```

静默周期的存在，会使得 Netty 尽量运行完所有的任务直到超时。如果真的没有任务了，或者超时了，会怎样呢？程序将进入下面的这个方法：

```
// io.netty.util.concurrent.SingleThreadEventExecutor#doStartThread
private void doStartThread() {
    assert thread == null;
    // key1，这里是真正启动线程的地方
    executor.execute(new Runnable() {
        @Override
        public void run() {
            thread = Thread.currentThread();
            if (interrupted) {
                thread.interrupt();
            }

            boolean success = false;
            updateLastExecutionTime();
            try {
                // key2，这个run()方法就是之前NioEventLoop中的run()方法
                SingleThreadEventExecutor.this.run();
                success = true;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                for (;;) {
                    int oldState = state;
                    if (oldState >= ST_SHUTTING_DOWN || STATE_UPDATER.compareAndSet(
                            SingleThreadEventExecutor.this, oldState, ST_SHUTTING_DOWN)) {
                        break;
                    }
                }

                if (success && gracefulShutdownStartTime == 0) {
                    if (logger.isErrorEnabled()) {
                        logger.error("Buggy " + EventExecutor.class.getSimpleName() + " implementation; " +
                                SingleThreadEventExecutor.class.getSimpleName() + ".confirmShutdown() must " +
                                "be called before run() implementation terminates.");
                    }
                }

                try {
                    // 再次执行confirmShutdown()直到没有任务或者超时
                    for (;;) {
                        if (confirmShutdown()) {
                            break;
                        }
                    }

                    // 修改状态为ST_SHUTDOWN，之后不能再添加任何任务
                    for (;;) {
                        int oldState = state;
```

```
                    if (oldState >= ST_SHUTDOWN || STATE_UPDATER.compareAndSet(
                        SingleThreadEventExecutor.this, oldState, ST_SHUTDOWN)) {
                        break;
                    }
                }

                // 最后一次运行confirmShutdown()方法，把剩余的任务全部运行完
                confirmShutdown();
            } finally {
                try {
                    // 执行cleanup()方法
                    cleanup();
                } finally {
                    // 移除所有的threadLocal
                    FastThreadLocal.removeAll();

                    // 更新状态为ST_TERMINATED
                    STATE_UPDATER.set(SingleThreadEventExecutor.this, ST_TERMINATED);
                    // threadLock标识减一，将触发某些事件
                    threadLock.countDown();
                    // 销毁的任务数，为什么还会有任务呢？
                    // 时刻考虑并发特性，其它线程有可能在上面最后一次运行confirmShutdown()之后又往当前线程添加了任务
                    int numUserTasks = drainTasks();
                    if (numUserTasks > 0 && logger.isWarnEnabled()) {
                        logger.warn("An event executor terminated with " +
                                "non-empty task queue (" + numUserTasks + ')');
                    }
                    // key，NioEventLoop的terminationFuture已成功
                    terminationFuture.setSuccess(null);
                }
            }
        }
    }
});
}
```

这个方法中有几个关键的地方：

- 多次执行 confirmShutdown () 方法，确保所有的任务都尽量完成；
- 执行 cleanup () 方法，又清理什么呢？
- 移除所有的 ThreadLocal 相关的资源；
- 最终状态设置为 ST_TERMINATED；
- threadLock 的值减一，它是一个 CountDownLatch，应该触发什么事件呢？
- NioEventLoop 的 terminationFuture 已成功完成，它又如何与 NioEventLoopGroup 的 termination 扯上关系？

我们着重看一下上面打问号的三个地方，先来看看 `cleanup()` 这个方法：

```
// io.netty.channel.nio.NioEventLoop#cleanup
@Override
protected void cleanup() {
    try {
        selector.close();
    } catch (IOException e) {
        logger.warn("Failed to close a selector.", e);
    }
}
```

其实很简单，它关闭了 selector。

再来看看 threadLock 这个变量，它的作用是什么，在上面它调用了 `countDown()` 方法，那么，一定存在某个地方有个 `await()` 方法等着我们，让我们找找看，果不其然，在下面这个地方找到了它：

```
// io.netty.util.concurrent.SingleThreadEventExecutor#awaitTermination
@Override
public boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException {
    ObjectUtil.checkNotNull(unit, "unit");
    if (inEventLoop()) {
        throw new IllegalStateException("cannot await termination of the current thread");
    }

    threadLock.await(timeout, unit);

    return isTerminated();
}
public boolean isTerminated() {
    return state == ST_TERMINATED;
}
```

这个方法的作用就是等待终止，它在哪里调用的呢？大胆猜测，它是在它的父亲，也就是 NioEventLoopGroup，那里调用的，找找看有没有类似的方法，还真有：

```
// io.netty.util.concurrent.MultithreadEventExecutorGroup#awaitTermination
@Override
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
    long deadline = System.nanoTime() + unit.toNanos(timeout);
    // 循环每一个NioEventLoop，等待它们终止
    loop: for (EventExecutor l: children) {
        for (;;) {
            long timeLeft = deadline - System.nanoTime();
            if (timeLeft <= 0) {
                break loop;
            }
            if (l.awaitTermination(timeLeft, TimeUnit.NANOSECONDS)) {
                break;
            }
        }
    }
    // 返回整个Group的终止状态
    return isTerminated();
}
```

那么，这个 Group 的等待终止方法又是在哪里调用的呢？源码中并没有调用的地方，它其实是留给用户自己调用的，比如，我们可以在主线程中调用这个方法，判断 Group 完全终止后做些什么事情。

最后，我们再来看看父亲的 terminationFuture 怎么跟孩子的 terminationFuture 关联起来的。

通过前面的分析，我们知道，NioEventLoopGroup 的 terminationFuture 是在其 shutdownGracefully () 中返回的，而 NioEventLoop 的 terminationFuture 是在其 shutdownGracefully () 方法中返回的，且它们的返回值并没有显式地联系起来，那么，它们到底有没有联系呢？又是怎样联系起来的呢？

让我们先找一下 NioEventLoop 的 terminationFuture 在何时置为完成的，通过上面的分析，发现是在 doStartThread () 这个方法中，当所有资源全部释放完毕之后，将其 terminationFuture 置为完成的，那么，父亲的 terminationFuture 正常来说应该需要等到所有孩子的 terminationFuture 完成之后才能置为完成，它在哪里呢？这个有点难找，最后，找到是在下面这个构造方法里面：

```
protected MultithreadEventExecutorGroup(int nThreads, Executor executor,
                                        EventExecutorChooserFactory chooserFactory, Object... args) {
    // 省略其他代码

    // 创建一个监听器
    final FutureListener<Object> terminationListener = new FutureListener<Object>() {
        @Override
        public void operationComplete(Future<Object> future) throws Exception {
            // 每个孩子完成时，terminatedChildren加一
            // 如果等于孩子数量，说明全部完成了
            if (terminatedChildren.incrementAndGet() == children.length) {
                terminationFuture.setSuccess(null);
            }
        }
    };

    // 给每个孩子都添加上这个监听器
    for (EventExecutor e: children) {
        e.terminationFuture().addListener(terminationListener);
    }

    // 省略其他代码
}
```

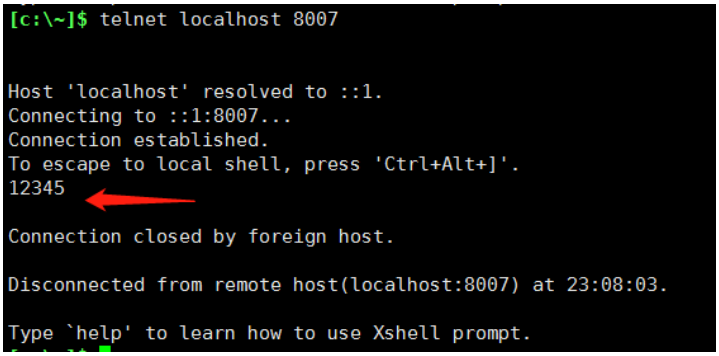
在 `MultithreadEventExecutorGroup` 的构造方法中，给每个孩子都添加了一个监听器，当孩子完成时，完成数量加一，直到等于孩子数量，则把父亲的 terminationFuture 设置为已成功完成，所以，两者的 terminationFuture 是有关系的。

等等，其实还没完，细心的同学会发现，中间有一个注释我打了一个 `todo` 标识：

```
// io.netty.channel.AbstractChannel.AbstractUnsafe#close
private void close(final ChannelPromise promise, final Throwable cause,
                   final ClosedChannelException closeCause, final boolean notify) {
    // 省略其他代码
    if (closeExecutor != null) {
        // 对于NioServerSocketChannel，默认为空，不会走到这，省略这段代码
    } else {
        try {
            // key，一看就是干正事的方法
            doClose0(promise);
        } finally {
            if (outboundBuffer != null) {
                // todo 未发送的数据将失败，为什么不让它们发送出去呢？
                outboundBuffer.failFlushed(cause, notify);
                outboundBuffer.close(closeCause);
            }
        }
    }
    // 省略其他代码
}
```

为什么“优雅关闭”，却不把缓存中的消息发送出去呢？我百思不得其解，为了验证源码没有错，我们可以做如下实验：

将 EchoServerHandler 中 channelReadComplete () 方法中的 ctx.flush() 方法注释掉，然后取消所有断点，只保留 main () 方法中 bossGroup.shutdownGracefully() 处的打点，启动服务器，当断点停留在此处的时候，使用 XSHELL 连接到服务端，并发送 “12345” 字符串，此时，按 F9 放开断点，服务端将优雅停机，观察客户端是否收到 “12345” 这条消息：



很遗憾，客户端确实没有收到 “12345” 这条消息，说明，我们源码读得没错，Netty 优雅停机时确实不会把未发送的消息继续发送完。

好了，到这里，今天所有的问题就都解决完了，让我们来总结一下 “优雅关闭服务” 的过程：

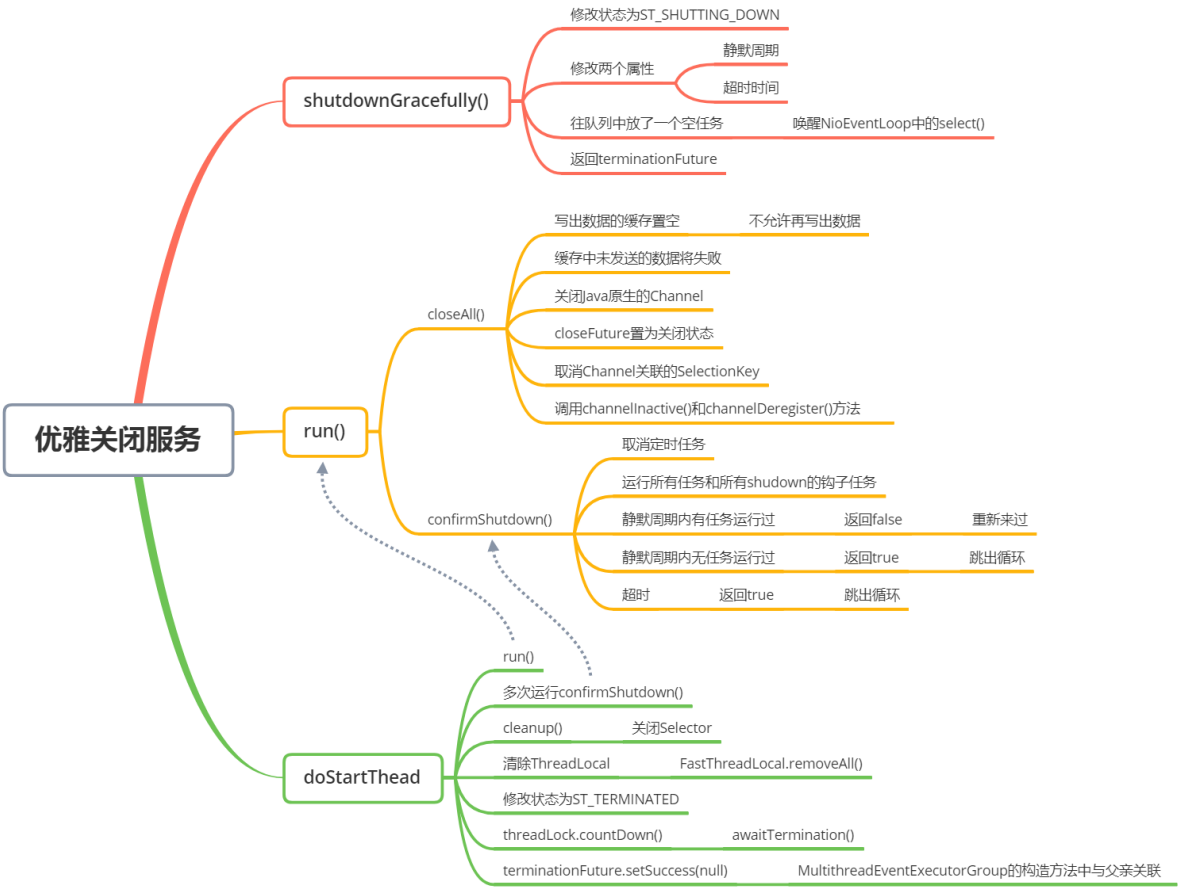
1. 优雅关闭服务分成两个部分，调用 shutdownGracefully ()，只是修改一些状态和属性的值，并唤醒 NioEventLoop 中的 select () 方法；
2. 真正的处理逻辑在 NioEventLoop 的 run () 方法中；
3. 关闭的关键方法又分为 closeAll () 和 confirmShutdown () 两个方法；
4. closeAll () 中主要是对 Channel 的关闭，跟 Channel 相关的资源释放都在这里，比如缓存消息的失败、SelectionKey 的取消、Java 原生 Channel 的关闭等；
5. confirmShutdown () 中主要是对队列中的任务或者钩子任务进行处理，主要是通过一个叫做静默周期的参数来控制尽量执行完所有任务，但是，也不能无限期等待，所以，还有一个超时时间进行控制；
6. 接着，程序的逻辑来到了真正启动线程的地方，也就是 doStartThread () 方法，它里面有个非常重要的方法 cleanup ()，并清理了所有 ThreadLocal 相关的资源，最后把 NioEventLoop 的状态设置为 ST_TERMINATED；
7. cleanup () 方法中主要是对 Selector 进行关闭；
8. 然后，我们分析了 NioEventLoopGroup 与 NioEventLoop 在程序终止时相关的联系，包括 threadLock 和 terminationFuture 等；
9. 最后，我们验证了 Netty 中的优雅关闭服务的时候确实不会真正发送缓存中的内容。

后记

今天的代码量有点大，且很绕，其中牵涉到很多的线程切换、ChannelFuture 的状态切换等，并不像之前的源码剖析那么顺畅，请大家保持耐心，文章只能起到引领的作用，具体的源码还是要大家亲自跑一遍才能理解里面的弯弯绕绕。

至此，从数据流向的角度来剖析源码的全部章节就讲完了，从下一章开始，我们将从 Netty 核心知识的角度来剖析源码，彼时，我们将会讲解很多 Java 中的高阶技巧，敬请期待！

思维导图



}

15 Netty服务如何写出数据

17 如何从源码的角度深入剖析 ByteBuffer