从入门到实战，Netty多线程篇案例集锦

原创 2015-09-10 李林峰 InfoQ

Netty案例集锦系列文章介绍

1 | Netty的特点

Netty入门比较简单，主要原因有如下几点：

- Netty的API封装比较简单，将复杂的网络通信通过BootStrap等工具类做了二次封装，用户使用起来比较简单；
- Netty源码自带的Demo比较多，通过Demo可以很快入门；
- Netty社区资料、相关学习书籍也比较多，学习资料比较丰富。

但是很多入门之后的Netty学习者遇到了很多困惑，例如不知道在实际项目中如何使用Netty、遇到Netty问题之后无从定位等，这些问题严重制约了对Netty的深入掌握和实际项目应用。

Netty相关问题比较难定位的主要原因如下：

1) NIO编程自身的复杂性，涉及到大量NIO类库、Netty自身封装的类库等，当你需要打开黑盒定位问题时，必须对这些类库了如指掌；否则即便定位到问题所在，也不知所以然，更无法修复；

2) Netty复杂的多线程模型，用户在实际使用Netty时，会涉及到Netty自己封装的线程组、线程池、NIO线程，以及业务线程，通信链路的创建、I/O消息的读写会涉及到复杂的线程切换，这会让初学者云山雾绕，调试起来非常痛苦，甚至都不知道从哪里调试；

3) Netty版本的跨度大，从实际商用情况看，涉及到了Netty 3.X、4.X和5.X等多个版本，每个Major版本之间特性变化非常大，即便是Minor版本都存在一些差异，这些功能特性和类库差异会给使用者带来很多问题，版本升级之后稍有不慎就会掉入陷阱。

2 | 案例来源

Netty案例集锦的案例来源于作者在实际项目中遇到的问题总结、以及Netty社区网友的反馈，大多数案例都来源于实际项目，也有少部分是读者在学习Netty中遭遇的比较典型的问题。

3 | 多线程篇

学习和掌握Netty多线程模型是个难点，在实际项目中如何使用好Netty多线程更加困难，很多网上问题和事故都来源于对Netty线程模型了解不透彻所致。鉴于此，Netty案例集锦系列就首先从多线程方面开始。

Netty 3 版本升级遭遇内存泄漏案例

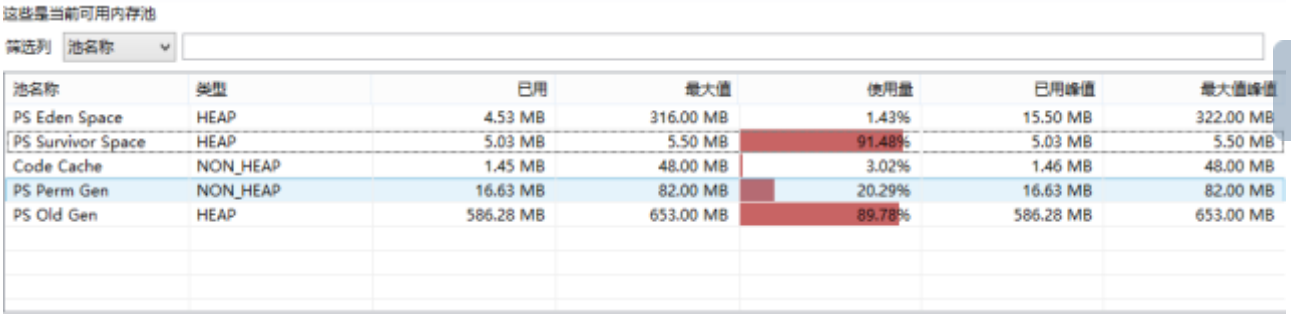
1 | 问题描述

业务代码升级Netty 3到Netty4之后，运行一段时间，Java进程就会宕机，查看系统运行日志发现系统发生了内存泄露（示例堆栈）：

```
java.lang.OutOfMemoryError: Direct buffer memory
        at java.nio.Bits.reserveMemory(Bits.java:658)
        at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)
        at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)
        at io.netty.buffer.PoolArena$DirectArena.newChunk(PoolArena.java:383)
        at io.netty.buffer.PoolArena.allocateNormal(PoolArena.java:143)
        at io.netty.buffer.PoolArena.allocate(PoolArena.java:132)
        at io.netty.buffer.PoolArena.allocate(PoolArena.java:94)
        at io.netty.buffer.PooledByteBufAllocator.newDirectBuffer(PooledByteBufAllocator.java:238)
        at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:155)
        at io.netty.buffer.AbstractByteBufAllocator.directBuffer(AbstractByteBufAllocator.java:146)
        at io.netty.buffer.AbstractByteBufAllocator.ioBuffer(AbstractByteBufAllocator.java:107)
        at io.netty.example.echo.EchoClientHandler.channelRead(EchoClientHandler.java:77)
        at io.netty.channel.ChannelHandlerInvokerUtil.invokeChannelReadNow(ChannelHandlerInvokerUtil.java:74)
        at io.netty.channel.DefaultChannelHandlerInvoker.invokeChannelRead(DefaultChannelHandlerInvoker.java:138)
        at io.netty.channel.DefaultChannelHandlerContext.fireChannelRead(DefaultChannelHandlerContext.java:320)
        at io.netty.channel.DefaultChannelPipeline.fireChannelRead(DefaultChannelPipeline.java:846)
        at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:127)
        at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:485)
        at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:452)
        at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:346)
        at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:794)
        at java.lang.Thread.run(Thread.java:745)
```

图2-1 内存泄漏堆栈

对内存进行监控（切换使用堆内存池，方便对内存进行监控），发现堆内存一直飙升，如下所示（示例堆内存监控）：



池名称	类型	已用	最大值	使用量	已用峰值	最大值峰值
PS Eden Space	HEAP	4.53 MB	316.00 MB	1.43%	15.50 MB	322.00 MB
PS Survivor Space	HEAP	5.03 MB	5.50 MB	91.48%	5.03 MB	5.50 MB
Code Cache	NON_HEAP	1.45 MB	48.00 MB	3.02%	1.46 MB	48.00 MB
PS Perm Gen	NON_HEAP	16.63 MB	82.00 MB	20.29%	16.63 MB	82.00 MB
PS Old Gen	HEAP	586.28 MB	653.00 MB	89.78%	586.28 MB	653.00 MB

图2-2 堆内存监控示例

2 | 问题定位

使用jmap -dump:format=b,file=netty.bin PID 将堆内存dump出来，通过IBM的HeapAnalyzer工具进行分析，发现ByteBuf发生了泄露。

因为使用了Netty 4的内存池，所以首先怀疑是不是申请的ByteBuf没有被释放导致？查看代码，发现消息发送完成之后，Netty底层已经调用ReferenceCountUtil.release(message)对内存进行了释放。这是怎么回事呢？难道Netty 4.X的内存池有Bug，调用release操作释放内存失败？

考虑到Netty 内存池自身Bug的可能性不大，首先从业务的使用方式入手分析：

1）内存的分配是在业务代码中进行，由于使用到了业务线程池做I/O操作和业务操作的隔离，实际上内存是在业务线程中分配的；

2）内存的释放操作是在outbound中进行，按照Netty 3的线程模型，downstream（对应Netty 4的outbound，Netty 4取消了upstream和downstream）的handler也是由业务调用者线程执行的，也就是说申请和释放在同一个业务线程中进行。初次排查并没有发现导致内存泄露的根因，继续分析Netty内存池的实现原理。

Netty 内存池实现原理分析：查看Netty的内存池分配器PooledByteBufAllocator的源码实现，发现内存池实际是基于线程上下文实现的，相关代码如下：

```
final ThreadLocal<PoolThreadCache> threadCache = new
ThreadLocal<PoolThreadCache>() {
    private final AtomicInteger index = new AtomicInteger();
    @Override
    protected PoolThreadCache initialValue() {
        final int idx = index.getAndIncrement();
        final PoolArena<byte[]> heapArena;
        final PoolArena<ByteBuffer> directArena;
        //.....此处代码省略
        return new PoolThreadCache(heapArena, directArena);

    }
```

图2-3

也就是说内存的申请和释放必须在同一线程上下文中，不能跨线程。跨线程之后实际操作的就不是同一块儿内存区域，这会导致很多严重的问题，内存泄露便是其中之一。内存在A线程申请，切换到B线程释放，实际是无法正确回收的。

3 | 问题根因

Netty 4修改了Netty 3的线程模型：在Netty 3的时候，upstream是在I/O线程里执行的，而downstream是在业务线程里执行。当Netty从网络读取一个数据报投递给业务handler的时候，handler是在I/O线程里执行；而当我们在业务线程中调用write和writeAndFlush向网络发送消息的时候,handler是在业务线程里执行，直到最后一个Header handler将消息写入到发送队列中，业务线程才返回。

Netty4修改了这一模型，在Netty 4里inbound(对应Netty 3的upstream)和outbound(对应Netty 3的downstream)都是在NioEventLoop(I/O线程)中执行。当我们在业务线程里通过ChannelHandlerContext.write发送消息的时候，Netty 4在将消息发送事件调度到ChannelPipeline的时候，首先将待发送的消息封装成一个Task，然后放到NioEventLoop的任务队列中，由NioEventLoop线程异步执行。后续所有handler的调度和执行，包括消息的发送、I/O事件的通知，都由NioEventLoop线程负责处理。

在本案例中，ByteBuf在业务线程中申请，在后续的ChannelHandler中释放，ChannelHandler是由Netty的I/O线程(EventLoop)执行的，因此内存的申请和释放不在同一个线程中，导致内存泄漏。

Netty 3的I/O事件处理流程：

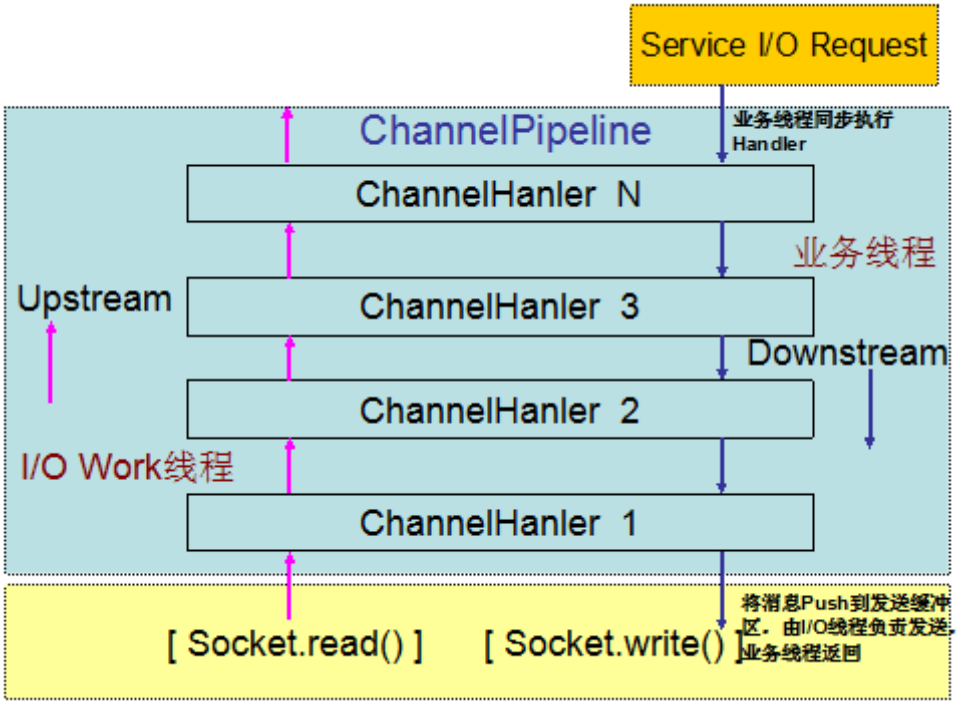


图2-4 Netty 3的I/O线程模型

Netty 4的I/O消息处理流程：

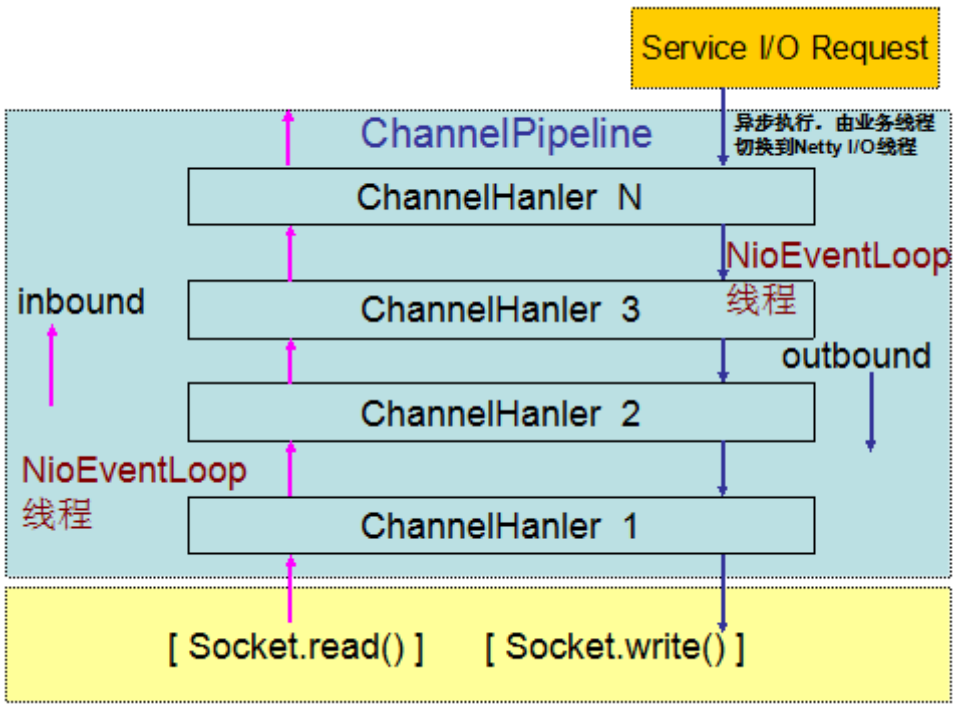


图2-5 Netty 4 I/O线程模型

4 | 案例总结

Netty 4.X版本新增的内存池确实非常高效，但是如果使用不当则会导致各种严重的问题。诸如内存泄露这类问题，功能测试并没有异常，如果相关接口没有进行压测或者稳定性测试而直接上线，则会导致严重的线上问题。

内存池PooledByteBuf的使用建议：

1）申请之后一定要记得释放，Netty自身Socket读取和发送的ByteBuf系统会自动释放，用户不需要做二次释放；如果用户使用Netty的内存池在应用中做ByteBuf的对象池使用，则需要自己主动释放；

2）避免错误的释放：跨线程释放、重复释放等都是非法操作，要避免。特别是跨线程申请和释放，往往具有隐蔽性，问题定位难度较大；

3）防止隐式的申请和分配：之前曾经发生过一个案例，为了解决内存池跨线程申请和释放问题，有用户对内存池做了二次包装，以实现多线程操作时，内存始终由包装的管理线程申请和释放，这样可以屏蔽用户业务线程模型和访问方式的差异。谁知运行一段时间之后再次发生了内存泄露，最后发现原来调用ByteBuf的write操作时，如果内存容量不足，会自动进行容量扩展。扩展操作由业务线程执行，这就绕过了内存池管理线程，发生了“引用逃逸”；

4）避免跨线程申请和使用内存池，由于存在“引用逃逸”等隐式的内存创建，实际上跨线程申请和使用内存池是非常危险的行为。尽管从技术角度看可以实现一个跨线程协调的内存池机制，甚至重写PooledByteBufAllocator，但是这无疑会增加很多复杂性，通常也使用不到。如果确实存在跨线程的ByteBuf传递，而且无法保证ByteBuf在另一个线程中会重新分配大小等操作，最简单保险的方式就是在线程切换点做一次ByteBuf的拷贝，但这会造成性能下降。

比较好的一种方案就是如果存在跨线程的ByteBuf传递，对ByteBuf的写操作要在分配线程完成，另一个线程只能做读操作。操作完成之后发送一个事件通知分配线程，由分配线程执行内存释放操作。

Netty 3 版本升级性能下降案例

1 | 问题描述

业务代码升级Netty 3到Netty4之后，并没有给产品带来预期的性能提升，有些甚至还发生了非常严重的性能下降，这与Netty 官方给出的数据并不一致。

Netty 官方性能测试对比数据：我们比较了两个分别建立在Netty 3和4基础上echo协议服务器。（Echo非常简单，这样，任何垃圾的产生都是Netty的原因，而不是协议的原因）。我使它们服务于相同的分布式echo协议客户端，来自这些客户端的16384个并发连接重复发送256字节的随机负载，几乎使千兆以太网饱和。

根据测试结果，Netty 4：

- GC中断频率是原来的1/5： 45.5 vs. 9.2次/分钟
- 垃圾生成速度是原来的1/5： 207.11 vs 41.81 MiB/秒

2 | 问题定位

首先通过JMC等性能分析工具对性能热点进行分析，示例如下（信息安全等原因，只给出分析过程示例截图）：

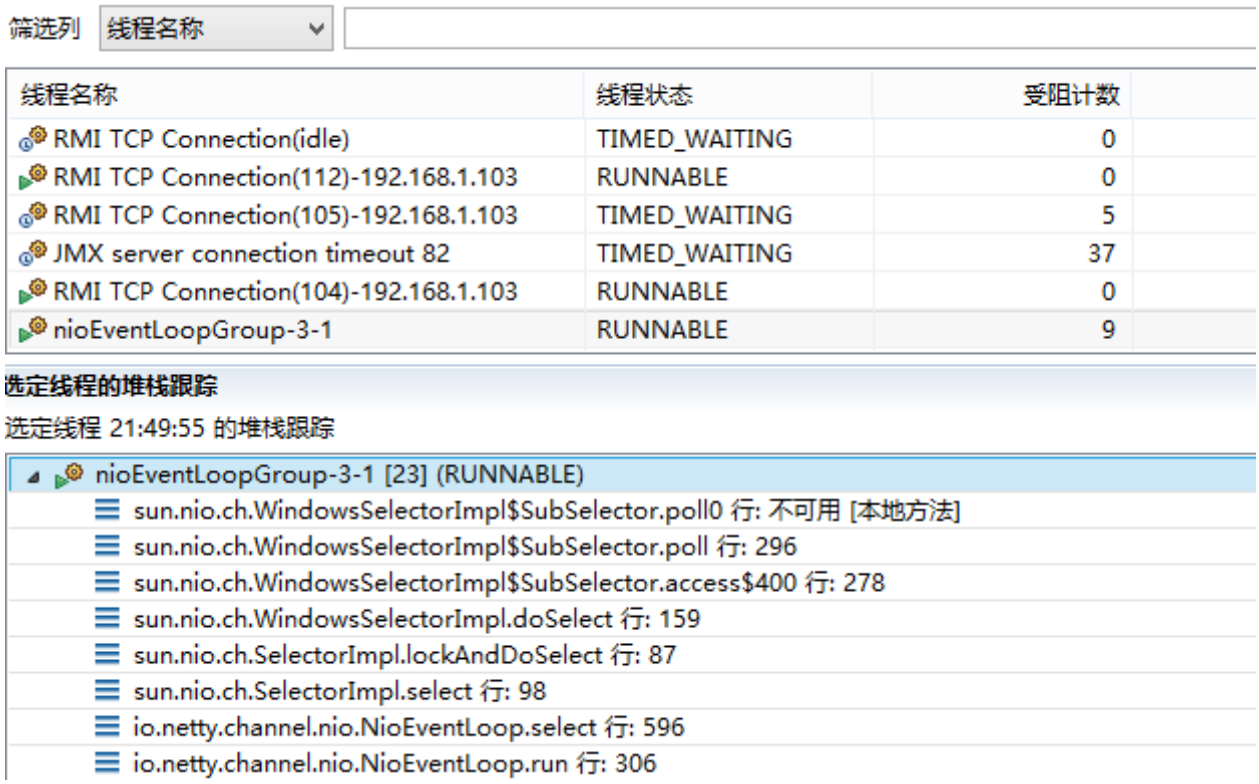


图3-1 性能热点线程堆栈

通过对热点方法的分析，发现在消息发送过程中，有两处热点：

1）消息发送性能统计相关Handler;
2）编码Handler。

对使用Netty 3版本的业务产品进行性能对比测试，发现上述两个Handler也是热点方法。既然都是热点，为啥切换到Netty4之后性能下降这么厉害呢?

通过方法的调用树分析发现了两个版本的差异：在Netty 3中，上述两个热点方法都是由业务线程负责执行；而在Netty 4中，则是由NioEventLoop(I/O)线程执行。对于某个链路，业务是拥有多个线程的线程池，而NioEventLoop只有一个，所以执行效率更低，返回给客户端的应答时延就大。时延增大之后，自然导致系统并发量降低，性能下降。

找出问题根因之后，针对Netty 4的线程模型对业务进行专项优化，将耗时的编码等操作迁移到业务线程中执行，为I/O线程减负，性能达到预期，远超过了Netty 3老版本的性能。

Netty 3的业务线程调度模型图如下所示：充分利用了业务多线程并行编码和Handler处理的优势，周期T内可以处理N条业务消息：

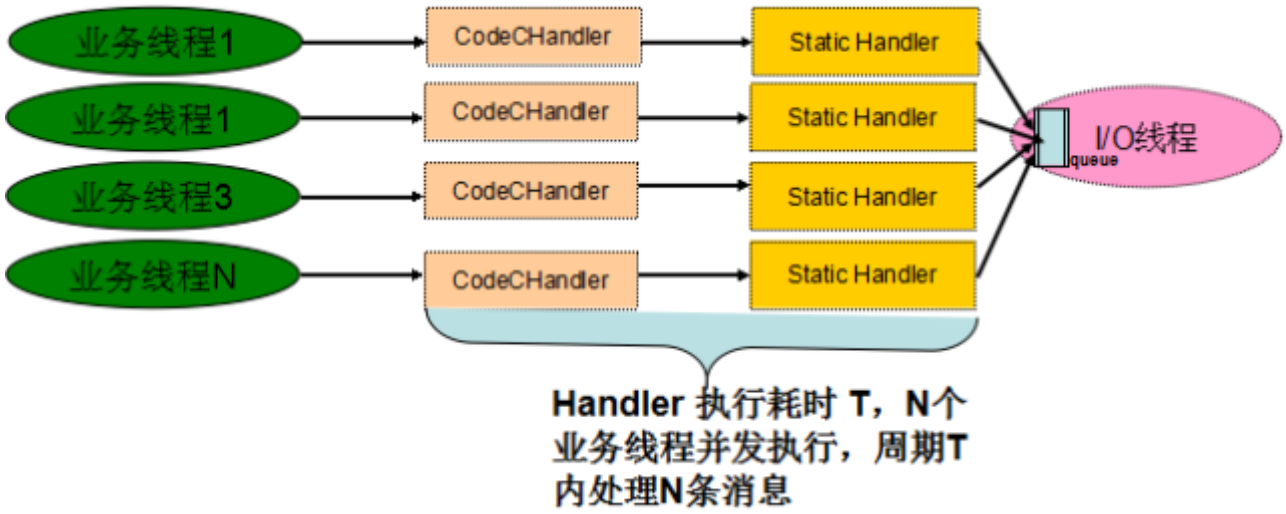


图3-2 Netty 3 Handler执行线程模型

切换到Netty 4之后，业务耗时Handler被I/O线程串行执行，因此性能发生比较大的下降:

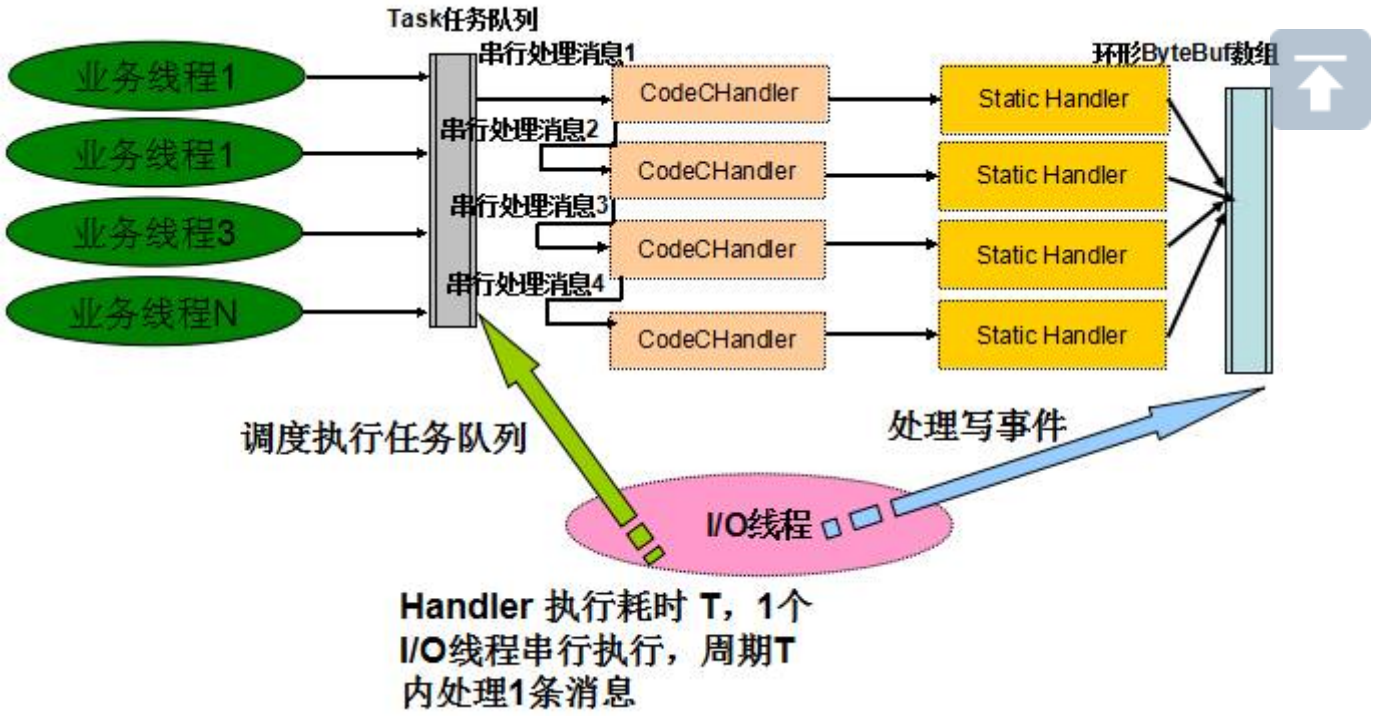


图3-3 Netty 4 Handler执行线程模型

3 | 问题总结

该问题的根因还是由于Netty 4的线程模型变更引起，线程模型变更之后，不仅影响业务的功能，甚至对性能也会造成很大的影响。

对Netty的升级需要从功能、兼容性和性能等多个角度进行综合考虑，切不可只盯着API变更这个芝麻，而丢掉了性能这个西瓜。API的变更会导致编译错误，但是性能下降却隐藏于无形之中，稍不留意就会中招。

对于讲究快速交付、敏捷开发和灰度发布的互联网应用，升级的时候更应该要当心。

Netty业务Handler接收不到消息案例

1 | 问题描述

我的服务碰到一个问题，经常有请求上来到MessageDecoder就结束了，没有继续往LogicServerHandler里面送，觉得很奇怪，是不是线程池满了？我想请教：

1）netty 5如何打印executor线程的占用情况，如空闲线程数？

2）executor设置的大小一般如何进行计算的?

业务代码示例如下:

```
 */
public class ServerChannelInitializer extends ChannelInitializer<SocketChannel> {
    private static final EventExecutorGroup executor = new DefaultEventExecutorGroup(500);

    @Override
    protected void initChannel(SocketChannel sc) throws Exception {

        ChannelPipeline pipeline = sc.pipeline();

        // logs
        pipeline.addLast("logger", new LoggingHandler(LogLevel.DEBUG));

        // readTimeoutHandler
        pipeline.addLast("readTimeoutHandler", new ReadTimeoutHandler(30));

        // Decoders
        pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(2048, 0, 2, 0, 2));
        pipeline.addLast("isoMessageDecoder", new MessageDecoder());

        // Encoder
        pipeline.addLast("frameEncoder", new LengthFieldPrepender(2));
        pipeline.addLast("isoMessageEncoder", new MessagePrepender());

        // and then business logic
        pipeline.addLast(executor, "logicHandler", new LogicServerHandler());
    }

}
```

2 | 问题定位

从服务端初始化代码来看，并没有什么问题，业务LogicServerHandler没有接收到消息，有如下几种可能:

1）客户端并没有将消息发送到服务端，可以在服务端LoggingHandler中打印日志查看;

2）服务端部分消息解码发生异常，导致消息被丢弃/忽略，没有走到LogicServerHandler中;

3）执行业务Handler的DefaultEventExecutor中的线程太繁忙，导致任务队列积压，长时间得不到处理。

通过抓包结合日志分析，可能导致问题的原因1和2排除，需要继续对可能原因3进行排查。

Netty 5如何打印executor线程的占用情况，如空闲线程数？回答这些问题，首先要了解Netty的线程组和线程池机制。

Netty的EventExecutorGroup实际就是一组EventExecutor，它的定义如下：

```
public abstract class MultithreadEventExecutorGroup extends AbstractEventExecu

    private final EventExecutor[] children;
    private final Set<EventExecutor> readonlyChildren;
    private final AtomicInteger childIndex = new AtomicInteger();
```

通常通过它的next方法从线程组中获取一个线程池，代码如下：

```
@Override
public EventExecutor next() {
    return children[Math.abs(childIndex.getAndIncrement() % children.length)];
}
```

```
/**
 * Return the {@link EventExecutorGroup} which is the parent
 */
EventExecutorGroup parent();
```

Netty EventExecutor的典型实现有两个：DefaultEventExecutor和SingleThreadEventLoop，在本案例中，因为使用的是DefaultEventExecutorGroup，所以实际执行业务Handler的线程池就是DefaultEventExecutor，它继承自SingleThreadEventExecutor，从名称就可以看出它是个单线程的线程池。它的工作原理如下：

1）DefaultEventExecutor聚合JDK的Executor和Thread, 首次执行Task的时候启动线程，将线程池状态修改为运行态；

2）Thread run方法循环从队列中获取Task执行，如果队列为空，则同步阻塞，线程无限循环执行，直到接收到退出信号。

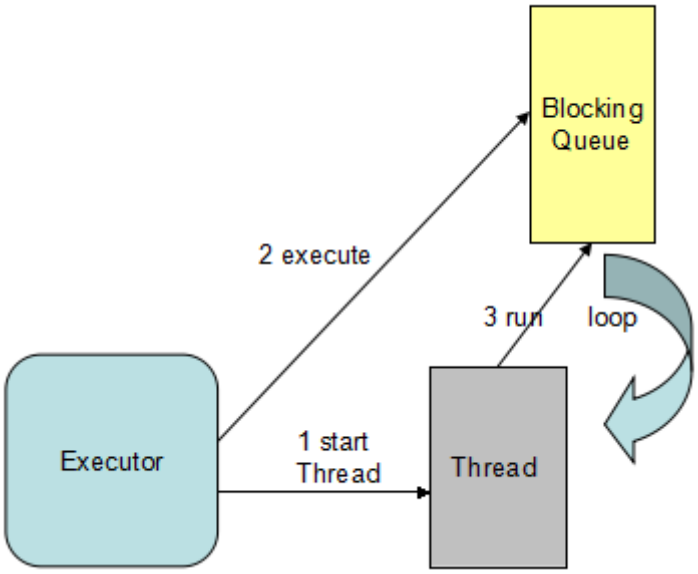


图4-1 DefaultEventExecutor工作原理

用户想通过Netty提供的DefaultEventExecutorGroup来并发执行业务Handler，但实际上却是单线程SingleThreadEventExecutor在串行执行业务逻辑，当服务端消息接收速度超过业务逻辑执行速度时，就会导致业务消息积压在SingleThreadEventExecutor的消息队列中得不到及时处理，现象就是业务Handler好像得不到执行，部分业务消息丢失。

讲解完Netty线程模型后，问题原因也定位出来了。其实我们发现，可以通过EventExecutor获取EventExecutorGroup的信息，然后获取整个EventExecutor线程组信息，最后打印线程负载信息，代码如下：

```
Set<EventExecutor> executorGroups = ctx.executor().parent().children();
        for(EventExecutor ext : executorGroups)
        {
           int size = ((SingleThreadEventExecutor)ext).pendingTasks();
           logger.info(ext.toString() + " pending size in queue is : --> " + size);
        }
```

执行结果如下：

```
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@5a2023f3 pending size in queue is : --> 0
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@741854be pending size in queue is : --> 0
七月 23, 2015 11:07:52 下午 io.netty.example.echo.EchoServerHandler printExecutorGroupInfo
信息: io.netty.util.concurrent.DefaultEventExecutor@1d652020 pending size in queue is : --> 0
```

3 | 问题总结

事实上，Netty为了防止多线程执行某个Handler（Channel）引起线程安全问题，实际只有一个线程会执行某个Handler，代码如下：

```
// Pin one of the child executors once and remember it so that the same child executor
// is used to fire events for the same channel.
ChannelHandlerInvoker invoker = childInvokers.get(group);
if (invoker == null) {
    EventExecutor executor = group.next();
    if (executor instanceof EventLoop) {
        invoker = ((EventLoop) executor).asInvoker();
    } else {
        invoker = new DefaultChannelHandlerInvoker(executor);
    }
    childInvokers.put(group, invoker);
}
```

需要指出的是，SingleThreadEventExecutor的pendingTasks可能是个耗时的操作，因此调用的时候需要注意：

```
/**
 * Return the number of tasks that are pending for processing.
 *
 * <strong>Be aware that this operation may be expensive as it depends on
 * the internal implementation of the
 * SingleThreadEventExecutor. So use it was care!</strong>
 */
```

实际就像JDK的线程池，不同的业务场景、硬件环境和性能标就会有不同的配置，无法给出标准的答案。需要进行实际测试、评估和调优来灵活调整。

最后再总结回顾下问题，对于案例中的代码，实际上在使用单线程处理某个Handler的LogicServerHandler，作者可能想并发多线程执行这个Handler，提升业务处理性能，但实际并没有达到设计效果。

如果业务性能存在问题，并不奇怪，因为业务实际是单线程串行处理的！当然，如果业务存在多个Channel，则每个/多个Channel会对应一个线程（池），也可以实现多线程处理，这取决于客户端的接入数。

案例中代码的线程处理模型如下所示（单个链路模型）：

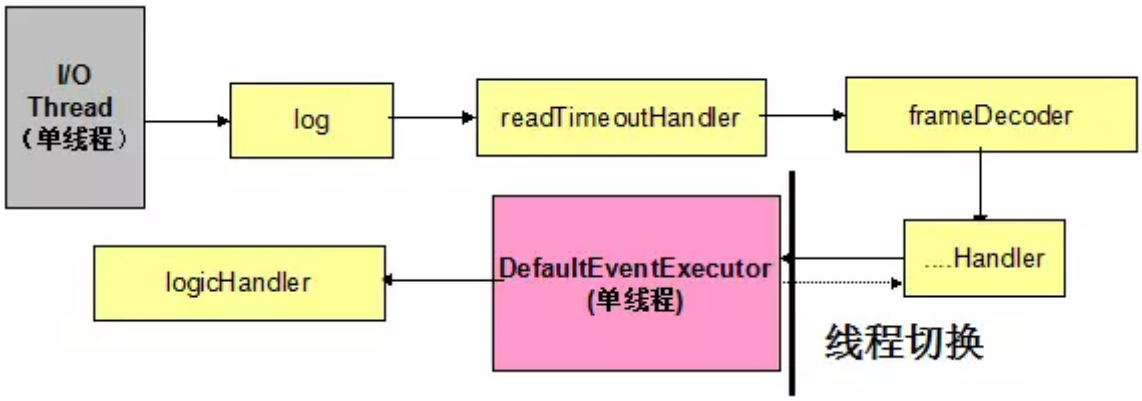


图4-3 单线程执行业务逻辑线程模型图

Netty 4 ChannelHandler线程安全疑问

1 | 问题咨询

我有一个非线程安全的类ThreadUnsafeClass，这个类会在channelRead方法中被调用。我下面这样的调用方法在多线程环境下安全吗？谢谢！

代码示例如下：

```
public class MyHandler extends ChannelInboundHandlerAdapter {

private ThreadUnsafeClass unsafe = new ThreadUnsafeClass();

public void channelRead(ChannelHandlerContext ctx, Object msg) {
//下面的代码是否ok？
unsafe.doSomething(ctx, msg);
}
……
}
```

2 | 解答

Netty 4优化了Netty 3的线程模型，其中一个非常大的优化就是用户不需要再担心ChannelHandler会被并发调用，总结如下：

1）ChannelHandler's的方法不会被Netty并发调用；
2）用户不再需要对ChannelHandler的各个方法做同步保护；
3）ChannelHandler实例不允许被多次添加到ChannelPiple中，否则线程安全将得不到保证。

根据上述分析，MyHandler的channelRead方法不会被并发调用，因此不存在线程安全问题。

3 | 一些特例

ChannelHandler的线程安全存在几个特例，总结如下：

1）如果ChannelHandler被注解为 @Sharable，全局只有一个handler实例，它会被多个Channel的Pipeline共享，会被多线程并发调用，因此它不是线程安全的；

2）如果存在跨ChannelHandler的实例级变量共享，需要特别注意，它可能不是线程安全的。

非线程安全的跨ChannelHandler变量原理如下：

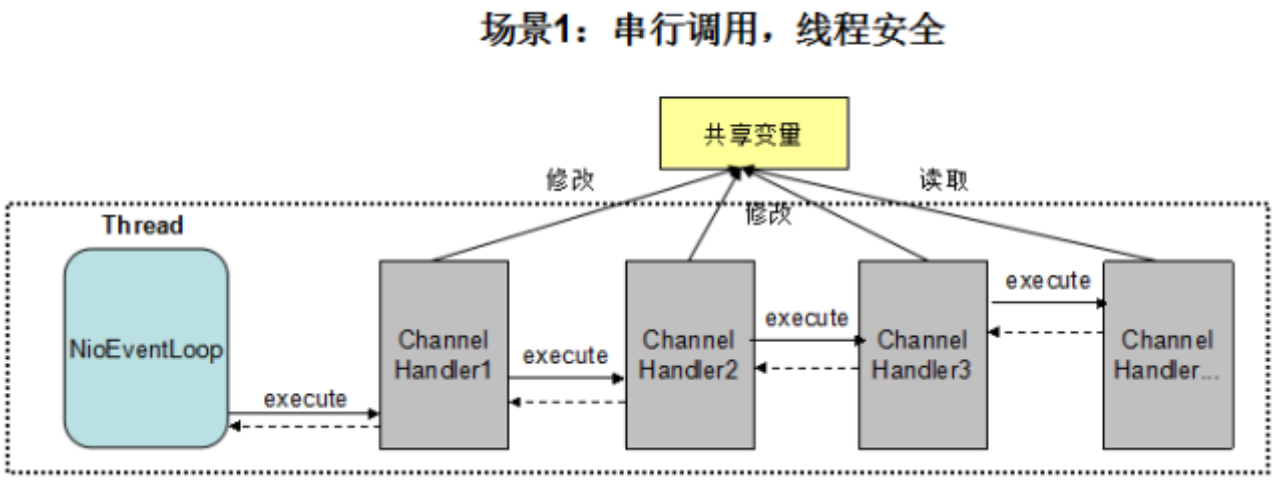


图5-1 串行调用，线程安全

Netty支持在添加ChannelHandler的时候，指定执行该Handler的EventExecutorGroup，这就意味着在整个ChannelPipeline执行过程中，可能会发生线程切换。此时，如果同一个对象在多个ChannelHandler中被共享，可能会被多线程并发操作，原理如下：

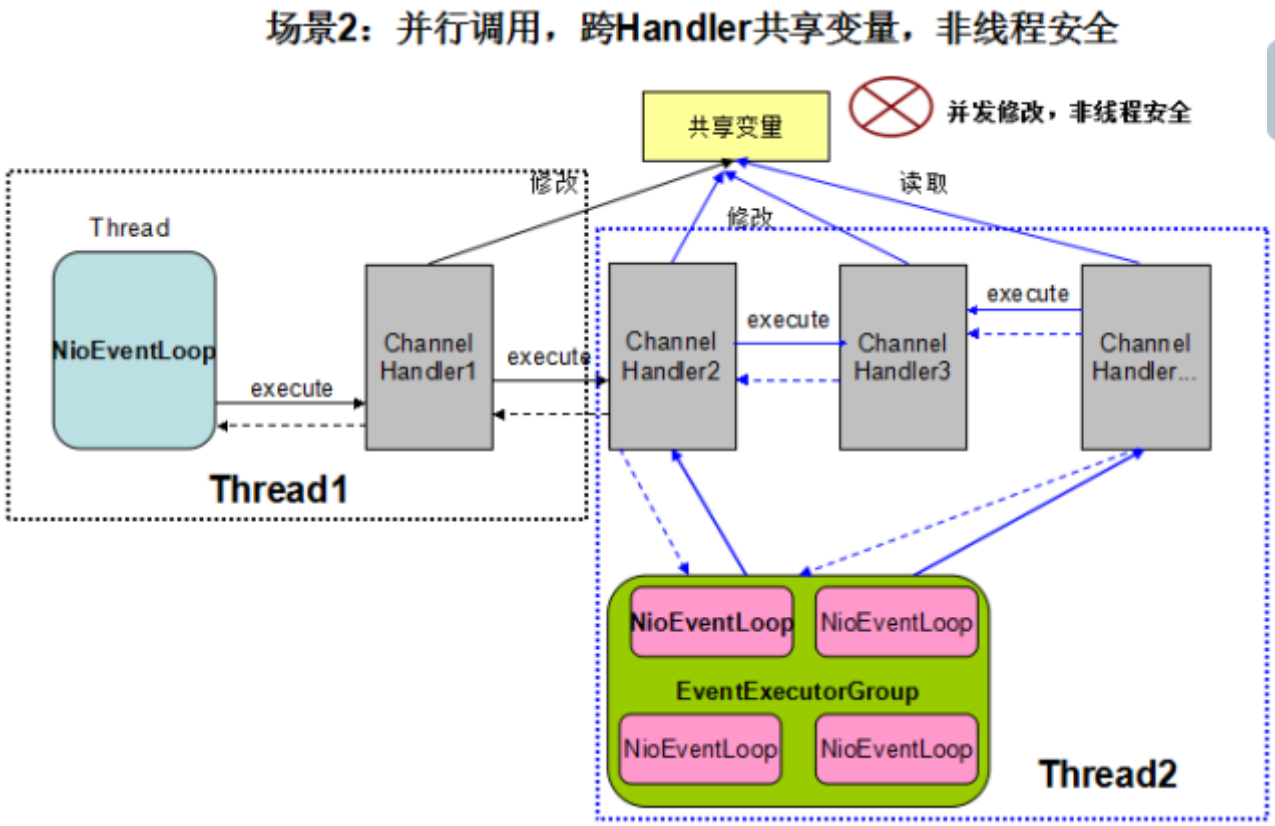


图5-2 并行调用，多Handler共享成员变量，非线程安全

作者 李林锋，2007年毕业于东北大学，2008年进入华为公司从事高性能通信软件的设计和开发工作，有7年NIO设计和开发经验，精通Netty、Mina等NIO框架和平台中间件，现任华为软件平台架构部架构师，《Netty权威指南》作者。目前从事华为下一代中间件和PaaS平台的架构设计工作。

回复关键词：React | 架构师 | 运维 | 云 | 开源 | 物联网 | Kubernetes | 架构 | 人工智能 | Kafka | Docker | Netty | CoreOS | QCon | Github | Swift | 敏捷 | 语言 | 程序员

长按识别二维码

干货 | 快讯 | 专访 | 解惑 | CTO

关注你关注的，加入技术人的专属社区