netty

背景

场景：业务线程生成数据速度远高于IO线程发送速度，ChannelOutboundBuffer队列积压。

原理：writeAndFlush操作将数据封装为Task加入队列，若队列无限制增长，触发堆内存OOM。

问题剖析

netty 流程剖析

服务端请求流程图请求过程

建立连接 ---> 接受请求 ---> 处理请求 ---> 回写响应体 ---> 关闭连接

连接建立

1. channel注册到selector上

```
AbstractNioChannel.java

@Override
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            //把channel注册到selector上
            selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);
```

```
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                // Force the Selector to select now as the "canceled" SelectionKey may still
be
                // cached and not removed because no Select.select(..) operation was called
yet.
                eventLoop().selectNow();
                selected = true;
            } else {
                // We forced a select operation on the selector before but the SelectionKey is
still cached
                // for whatever reason. JDK bug ?
                throw e;
            }
        }
    }
}

void register(EventLoop eventLoop, final ChannelPromise promise) //eventloop() 在注册的时候注册的
是eventLoop

注册到BOSS线程中
 final ChannelFuture initAndRegister() {
    Channel channel = null;
    try {
        //创建一个channel对象
        channel = channelFactory.newChannel();
        //初始化channel配置
        init(channel);
    } catch (Throwable t) {
        if (channel != null) {
            //如果 newChannel 崩溃，channel 可以为 null
            channel.unsafe().closeForcibly();
            // 由于通道尚未注册，我们需要强制使用 GlobalEventExecutor
            return new DefaultChannelPromise(channel,
GlobalEventExecutor.INSTANCE).setFailure(t);
        }
        // 由于通道尚未注册，我们需要强制使用 GlobalEventExecutor
        return new DefaultChannelPromise(new FailedChannel(),
GlobalEventExecutor.INSTANCE).setFailure(t);
    }
    //获取初始化设置的bossGroup,将channel绑定到
    ChannelFuture regFuture = config().group().register(channel);
    if (regFuture.cause() != null) {
        if (channel.isRegistered()) {
            channel.close();
        } else {
            channel.unsafe().closeForcibly();
        }
    }
    /**
        * 如果在这promise没有失败，则一定是以下原因
        * 1.如果我们尝试从事件循环中注册，此时注册已经完成。因为channel注册完成，使用bind(),connect () 是
安全的
        * 2.如果我们尝试从其他线程注册，则注册请求已经成功添加到事件循环的任务队列以供稍后执行
        *  因为bind(),connect () 将在定时任务后执行
        *  应为register(), bind(), and connect()被绑定在相同的线程
        */

    return regFuture;
}
```

2. 事件监听激活在Channel绑定端口后，通过ChannelPipeline触发channelActive事件，最终调用AbstractNioChannel#doBeginRead设置OP_ACCEPT监听：

```
触发事件后调用ProcessSelectedKey, Reactor的Selector 内核态的tcp accept 转为用户态的Event事件
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final NioUnsafe unsafe = ch.unsafe();
    int readyOps = k.readyOps();
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0) {
        unsafe.read(); // 核心处理入口
    }
}

NioMessageUnsafe#read()（NioServerSocketChannel的Unsafe实现）负责处理OP_ACCEPT事件：

// NioMessageUnsafe#read
public void read() {
    try {
        do {
            SocketChannel socketChannel = javaChannel().accept();
            // 创建客户端Channel并触发pipeline传播
            pipeline.fireChannelRead(socketChannel);
        } while (继续接受新连接);
    } catch (IOException e) { /* 异常处理 */ }
}
ServerSocketChannel.accept()接受客户端连接，生成SocketChannel。
新连接的SocketChannel被封装为NioSocketChannel，并通过pipeline.fireChannelRead()传播到业务逻辑处理器。

新连接注册到Worker线程
在ChannelRead事件传播中，通过ServerBootstrap的ServerBootstrapAcceptor将新NioSocketChannel注册到
Worker线程组的NioEventLoop，监听OP_READ事件：

// ServerBootstrapAcceptor#channelRead
 @Override
@SuppressWarnings("unchecked")
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    //对应传入进来的SocketChannel(创建连接的时候创建的)
    final Channel child = (Channel) msg;
    //最外侧添加的自己的写的handler
    child.pipeline().addLast(childHandler);
    //设置属性
    setChannelOptions(child, childOptions, logger);
    setAttributes(child, childAttrs);

    try {
        //对应WorkGroup
        childGroup.register(child).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    } catch (Throwable t) {
        forceClose(child, t);
    }
}

private static void forceClose(Channel child, Throwable t) {
    child.unsafe().closeForcibly();
    logger.warn("Failed to register an accepted channel: {}", child, t);
}
```

接受请求

```
1.注册到Worker线程

NioEventLoop#register
 public void register(final SelectableChannel ch, final int interestOps, final NioTask<?>
task) {
    ObjectUtil.checkNotNull(ch, "ch");
    if (interestOps == 0) {
        throw new IllegalArgumentException("interestOps must be non-zero.");
    }
    if ((interestOps & ~ch.validOps()) != 0) {
        throw new IllegalArgumentException(
                "invalid interestOps: " + interestOps + "(validOps: " + ch.validOps() + ')');
    }
    ObjectUtil.checkNotNull(task, "task");

    if (isShutdown()) {
        throw new IllegalStateException("event loop shut down");
    }

    if (inEventLoop()) {
        register0(ch, interestOps, task);
    } else {
        try {
            // Offload to the EventLoop as otherwise
java.nio.channels.spi.AbstractSelectableChannel.register
            // may block for a long time while trying to obtain an internal lock that may be
hold while selecting.
            submit(new Runnable() {
                @Override
                public void run() {
                    register0(ch, interestOps, task);
                }
            }).sync();
        } catch (InterruptedException ignore) {
            // Even if interrupted we did schedule it so just mark the Thread as interrupted.
            Thread.currentThread().interrupt();
        }
    }
}

2. 监听事件


private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    if (!k.isValid()) {
        final EventLoop eventLoop;
        try {
            eventLoop = ch.eventLoop();
        } catch (Throwable ignored) {
            // If the channel implementation throws an exception because there is no event
loop, we ignore this
            // because we are only trying to determine if ch is registered to this event loop
and thus has authority
            // to close ch.
            return;
        }
        // Only close ch if ch is still registered to this EventLoop. ch could have
deregistered from the event loop
        // and thus the SelectionKey could be cancelled as part of the deregistration process,
but the channel is
        // still healthy and should not be closed.
        // See https://github.com/netty/netty/issues/5125
        if (eventLoop == this) {
            // close the channel if the key is not valid anymore
            unsafe.close(unsafe.voidPromise());
        }
```

```
        return;
    }

    try {
        int readyOps = k.readyOps();
        // We first need to call finishConnect() before try to trigger a read(...) or write(...) as otherwise
        // the NIO JDK channel implementation may throw a NotYetConnectedException.
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            // remove OP_CONNECT as otherwise Selector.select(..) will always return without blocking
            // See https://github.com/netty/netty/issues/924
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);

            unsafe.finishConnect();
        }

        // Process OP_WRITE first as we may be able to write some queued buffers and so free memory.
        <!-- 数据过多无法一次性写入的时候会由OP_WRITE事件产生 -->
        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            // Call forceFlush which will also take care of clear the OP_WRITE once there is nothing left to write
            ch.unsafe().forceFlush();
        }

        // Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead
        // to a spin loop
        //请求读处理（断开连接）或接入连接
        //2种请求方式对应NioMessageChannel和NioByteChannel   2种对应的处理逻辑
        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
            unsafe.read();
        }
    } catch (CancelledKeyException ignored) {
        unsafe.close(unsafe.voidPromise());
    }
}
```

3. unsafe.read

unsafe是NioByteUnsafe实例（定义在AbstractNioByteChannel中），负责底层SocketChannel的I/O操作。

```
public final void read() {
    final ChannelConfig config = config();
    if (shouldBreakReadReady(config)) {
        clearReadPending();
        return;
    }
    final ChannelPipeline pipeline = pipeline();
    //ByteBuf分配器
    final ByteBufAllocator allocator = config.getAllocator();
    //抉择下一次分配多少ByteBuf
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false;
    try {
        do {
            //分配适合的大小  ---- 内存分配策略核心部分 --- AdaptiveRecvByteBufAllocator
            byteBuf = allocHandle.allocate(allocator);
            //读并且记录读了多少，如果堵满了，下次continue的化就直接扩容
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            if (allocHandle.lastBytesRead() <= 0) {
                // nothing was read. release the buffer.
                byteBuf.release();
```

```
                byteBuf = null;
                close = allocHandle.lastBytesRead() < 0;
                if (close) {
                    // There is nothing left to read as we received an EOF.
                    readPending = false;
                }
                break;
            }

            allocHandle.incMessagesRead(1);
            readPending = false;
            //pipeline上执行，业务逻辑的处理就在这个地方
            pipeline.fireChannelRead(byteBuf);
            byteBuf = null;
        } while (allocHandle.continueReading());
        //计算这次事件总共读了多少数据，计算下次分配大小
        allocHandle.readComplete();
        //相当于完成本次读事件处理
        pipeline.fireChannelReadComplete();

        if (close) {
            closeOnRead(pipeline);
        }
    } catch (Throwable t) {
        handleReadException(pipeline, byteBuf, t, close, allocHandle);
    } finally {
        // Check if there is a readPending which was not processed yet.
        // This could be for two reasons:
        // * The user called Channel.read() or ChannelHandlerContext.read() in
channelRead(...) method
        // * The user called Channel.read() or ChannelHandlerContext.read() in
channelReadComplete(...) method
        //
        // See https://github.com/netty/netty/issues/2254
        if (!readPending && !config.isAutoRead()) {
            removeReadOp();
        }
    }
}
```

业务处理

DefaultChannelPipeline 添加各类Handler后在不同阶段会触发不同的方法 以下是Netty中 `DefaultChannelPipeline`在不同阶段触发的事件方法分类及对应场景的整理表格，结合入站（Inbound）和出站（Outbound）事件类型进行说明：

阶段/场景	事件类型	触发方法	传播方向	说明
通道初始化	Inbound	`fireChannelRegistered()`	从Head到Tail	通道注册到EventLoop后触发，用于初始化资源（如`ChannelInitializer`）
	Inbound	`fireChannelActive()`	从Head	通道激活（如TCP连接建立或端口绑定完成）时触发

阶段/场景	事件类型	触发方法	传播方向	说明
			到 Tail	
数据接收（读操作）	Inbound	`fireChannelRead(Object msg)`	从 Head 到 Tail	读取到数据时触发，需在Handler中调用`ctx.fireChannelRead()`传递
	Inbound	`fireChannelReadComplete()`	从 Head 到 Tail	单次读取循环完成时触发，用于批量处理后的逻辑（如刷新响应）
数据发送（写操作）	Outbound	`bind(SocketAddress)` / `connect(SocketAddress)`	从 Tail 到 Head	绑定端口或发起连接时触发，最终由HeadContext执行系统调用
	Outbound	`write(Object msg)` / `writeAndFlush(Object msg)`	从 Tail 到 Head	数据写入缓冲区，`flush()`触发底层发送
连接状态变更	Inbound	`fireChannelInactive()`	从 Head 到 Tail	连接关闭时触发（如TCP连接断开）
	Inbound	`fireChannelUnregistered()`	从 Head 到 Tail	通道从EventLoop注销时触发，用于资源清理
异常处理	Inbound	`fireExceptionCaught(Throwable cause)`	从 Head 到 Tail	处理过程中发生异常时触发，默认由TailContext记录日志
流量控制	Inbound	`fireChannelWritabilityChanged()`	从 Head 到 Tail	发送缓冲区水位变化时触发（如高水位线阻塞）
用户自定	Inbound	`fireUserEventTriggered(Object event)`	从 Head	用户通过`ctx.fireUserEventTriggered()`

阶段/场景	事件类型	触发方法	传播方向	说明
义事件			到Tail	触发的自定义事件
出站操作控制	Outbound	`read()` / `deregister()` / `close()`	从Tail到Head	主动发起读取、注销或关闭连接操作

补充说明：

1. 传播方向： • 入站事件（如数据接收）从`HeadContext`向`TailContext`传播。

   • 出站事件（如数据发送）从当前Handler向`HeadContext`传播，最终由`Unsafe`类执行底层I/O操作。

2. 关键源码逻辑： • 事件触发入口：`NioEventLoop`通过`processSelectedKey()`检测I/O事件，调用`pipeline.fireXXX()`方法。

   • 动态处理器链：通过`addLast()`将Handler封装为`AbstractChannelHandlerContext`节点，插入双向链表。

3. 设计亮点： • 责任链模式：通过双向链表实现事件动态编排，支持业务逻辑分层处理（如编解码、业务逻辑、日志）。

   • 线程安全：通过`synchronized`块保证多线程下链表操作安全。

如需深入源码细节，可参考`DefaultChannelPipeline`类中的`fireChannelActive()`和`bind()`方法实现。

响应体写入

```
通过ChannelHandlerContext或Channel调用writeAndFlush()，本质上触发AbstractChannelHandlerContext中的
统一入口方法。无论从TailContext还是当前Handler调用，最终会通过findContextOutbound()向前查找出站处理器链。

触发TailContext.writeAndFlush()
  → 事件传播至Encoder.encode()转换Java对象为ByteBuf
  → HeadContext调用Unsafe.write()写入ChannelOutboundBuffer
  → Entry加入链表，更新totalPendingSize及水位状态
  → 调用flush()后，遍历Entry链表发送数据至Socket
  → 移除已发送Entry，触发Promise回调

@Override
public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {
    write(msg, true, promise);
    return promise;
}

private void write(Object msg, boolean flush, ChannelPromise promise) {
    ObjectUtil.checkNotNull(msg, "msg");
    try {
        if (isNotValidPromise(promise, true)) {
            ReferenceCountUtil.release(msg);
            // cancelled
```

```
            return;
        }
    } catch (RuntimeException e) {
        ReferenceCountUtil.release(msg);
        throw e;
    }

    final AbstractChannelHandlerContext next = findContextOutbound(flush ?
            (MASK_WRITE | MASK_FLUSH) : MASK_WRITE);
    //引用计数用，用来检测内存泄漏
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            next.invokeWrite(m, promise);
        }
    } else {
        final WriteTask task = WriteTask.newInstance(next, m, promise, flush);
        if (!safeExecute(executor, task, promise, m, !flush)) {
            // We failed to submit the WriteTask. We need to cancel it so we decrement the
pending bytes
            // and put it back in the Recycler for re-use later.
            //
            // See https://github.com/netty/netty/issues/8343.
            task.cancel();
        }
    }
}

 @Override
public final void write(Object msg, ChannelPromise promise) {
    assertEventLoop();
    //判断channel是否已经关闭了
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        try {
            // release message now to prevent resource-leak
            ReferenceCountUtil.release(msg);
        } finally {
            // If the outboundBuffer is null we know the channel was closed and so
            // need to fail the future right away. If it is not null the handling of the rest
            // will be done in flush0()
            // See https://github.com/netty/netty/issues/2362
            safeSetFailure(promise,
                    newClosedChannelException(initialCloseCause, "write(Object,
ChannelPromise)"));
        }
        return;
    }

    int size;
    try {
        // 零拷贝
        msg = filterOutboundMessage(msg);
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        try {
            ReferenceCountUtil.release(msg);
        } finally {
            safeSetFailure(promise, t);
        }
        return;
    }
```

```
    //消息发送到buf里面
    outboundBuffer.addMessage(msg, size, promise);
}


 @Override
protected final Object filterOutboundMessage(Object msg) {
    if (msg instanceof ByteBuf) {
        ByteBuf buf = (ByteBuf) msg;
        if (buf.isDirect()) {
            return msg;
        }

        return newDirectBuffer(buf);
    }

    if (msg instanceof FileRegion) {
        return msg;
    }

    throw new UnsupportedOperationException(
            "unsupported message type: " + StringUtil.simpleClassName(msg) + EXPECTED_TYPES);
}


 public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    if (tailEntry == null) {
        flushedEntry = null;
    } else {
        Entry tail = tailEntry;
        tail.next = entry;
    }
    tailEntry = entry;//添加到队尾
    if (unflushedEntry == null) {
        unflushedEntry = entry;
    }

    // increment pending bytes after adding message to the unflushed arrays.
    // See https://github.com/netty/netty/issues/1619
    incrementPendingOutboundBytes(entry.pendingSize, false);
}

 public void addFlush() {
    // There is no need to process all entries if there was already a flush before and no new
messages
    // where added in the meantime.
    //
    // See https://github.com/netty/netty/issues/2577
    Entry entry = unflushedEntry;
    if (entry != null) {
        if (flushedEntry == null) {
            // there is no flushedEntry yet, so start with the entry
            flushedEntry = entry;
        }
        do {
            flushed ++;
            if (!entry.promise.setUncancellable()) {
                // Was cancelled so make sure we free up memory and notify about the freed
bytes
                int pending = entry.cancel();
                decrementPendingOutboundBytes(pending, false, true);
            }
            entry = entry.next;
        } while (entry != null);

        // All flushed so reset unflushedEntry
        unflushedEntry = null;
```

```
        }
    }
```

总体逻辑

将上述流程总结如下述总体逻辑：

请求处理流程

1. EventLoop线程模型

- Boss-Worker分工：
  - Boss线程组：仅处理OP_ACCEPT事件，轻量化设计。
  - Worker线程组：处理OP_READ/OP_WRITE事件，绑定客户端Channel生命周期。
- 线程选择策略：通过EventExecutorChooser（如轮询或幂等选择器）分配Channel到不同线程。

2. ChannelPipeline与Handler

- 动态编排：服务端通过ServerBootstrapAcceptor将客户端Channel的Pipeline初始化为用户配置的childHandler链。
- 零拷贝优化：通过ByteBuf和FileRegion实现高效数据传输，减少内存复制。

1. 异步与Future机制ChannelFuture：连接操作返回ChannelFuture，支持addListener()异步回调

内存剖析

请求内存申请过程

内存申请

阶段	内存操作	释放时机	引用机制
OP_ACCEPT	分配元数据（Channel/Pipeline）	Channel关闭时自动释放	框架管理
OP_READ	分配接收缓冲区（PooledByteBuf）	默认由TailContext释放，或业务手动释放	引用计数
业务处理	可能分配新ByteBuf（如响应数据）	业务调用`writeAndFlush()`后由框架释放	引用计数+发送完成回调
OP_WRITE	转换堆内存为堆外内存（零拷贝优化）	数据发送完成后自动释放	框架自动管理
异常/关闭	释放所有未回收的ByteBuf	`channelInactive()`或`exceptionCaught()`	强制释放

内存核心机制

Netty 的内存管理机制是其高性能网络通信的核心，通过池化技术、零拷贝、引用计数等策略优化内存使用效率并降低GC压力。以下是其核心机制及实现原理的详细解析：

内存分类与池化技术

1. 堆内存（Heap Buffer）与直接内存（Direct Buffer）
   - 堆内存：分配在JVM堆上，受GC管理，适合小数据或频繁创建/销毁的场景。
   - 直接内存：通过`ByteBuffer.allocateDirect()`分配，避免JVM堆与操作系统间的数据复制，提升I/O性能，但需手动释放。
   - 池化（Pooled）与非池化（Unpooled）：
     - 池化内存（`PooledByteBufAllocator`）通过预分配内存块并复用，减少频繁分配/回收的开销。
     - 非池化内存（`UnpooledByteBufAllocator`）每次分配新内存，适合临时使用场景。
2. 内存规格划分
   Netty将内存划分为四类以适应不同需求：
   - Tiny（<512B）：用于小对象（如协议头）。
   - Small（<8KB）：适用于常见数据包。
   - Normal（8KB-16MB）：通过`PoolChunk`管理，按页（8KB）分配。
   - Huge（>16MB）：直接分配非池化内存，避免大块内存浪费。

内存分配与回收机制

1. 分层内存池结构
   - PoolArena：管理内存池，分为`HeapArena`（堆内存）和`DirectArena`（直接内存）。
   - PoolChunk：以16MB为单元，拆分为多个8KB的页（Page），按需分配内存块。
   - PoolSubpage：将8KB的页进一步拆分为更小的块（如512B），提升小内存利用率。
   - PoolThreadCache：线程本地缓存，减少多线程竞争，优先从本地缓存分配内存。
2. 引用计数与释放
   - 引用计数（Reference Counting）：每个`ByteBuf`维护计数器，通过`retain()`增加、`release()`减少引用，归零时触发释放。
   - 自动回收：池化内存释放后回归内存池；非池化堆内存由GC回收，直接内存需显式释放。
   - 泄漏检测：通过`ResourceLeakDetector`监控未释放的`ByteBuf`，支持`PARANOID`模式记录泄漏堆栈。

零拷贝优化

1. 零拷贝技术
   - CompositeByteBuf：逻辑合并多个`ByteBuf`，避免物理复制（如合并协议头和数据体）。

- FileRegion：通过`FileChannel.transferTo()`直接传输文件，减少内核态与用户态的数据拷贝。
- Direct Buffer复用：网络传输直接使用堆外内存，省去堆内到堆外的复制步骤。

背景问题解决方案

1. 设置流量控制机制

高低水位（WriteBufferWaterMark）：

```
channel.config().setWriteBufferHighWaterMark(64 * 1024); // 设置高水位
if (channel.isWritable()) {
    channel.writeAndFlush(data); // 可写时发送
} else {
    // 丢弃或者存储至内存或redis适时重发,第一阶段建议直接丢弃
}
```

2. 连接channel关闭,对于长时间处于弱网的连接进行关闭

记录channel超过高水位的次数,如果长时间处于弱网,关闭连接.

3. 强制释放策略：

在exceptionCaught和channelInactive事件中释放未发送的积压数据：

```
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    ChannelOutboundBuffer buffer = ctx.channel().unsafe().outboundBuffer();
    buffer.failFlushed(cause, true);  // 释放积压数据并关闭连接
}
```