13 Netty服务如何接收新的连接

更新时间：2020-07-28 09:31:33

> 合理安排时间，就等于节约时间。——培根

前言

你好，我是彤哥。

上一节，我们一起学习了服务启动过程的源码剖析，简单回顾一下，服务启动的时候会创建 ServerSocketChannel，并将之与 ChannelPipeline、EventLoop、Selector、Java 原生 Channel 进行绑定，同时通过 Java 原生 Channel 绑定到一个本地地址。

那么，如果有新连接进来，服务是如何接收（或接受，Accept）的呢？

本节，我们就来深入学习服务接收连接的源码剖析。

问题

经过上一节的学习，我们知道，服务启动的时候会往 NioServerSocketChannel 对应的 ChannelPipeline 后面添加一个叫做 ServerBootstrapAcceptor 的 ChannelHandler，那么，今天的问题是：

1. ServerBootstrapAcceptor 的作用是什么？
2. 接收连接的过程是否也要跟 Java 原生 Channel 打交道？
3. Selector 又是在哪里使用到的？

带着这几个问题进入今天的探索吧。

调试技巧

上一节，有我们自己编写的 main () 方法为基础，我们把第一个断点打在 main () 方法中，跟着断点一步一步调试即可剖析出整个服务启动过程的源码，本节的主题为服务接收新连接，断点该打在哪里呢？又该如何调试呢？

针对这个问题，我给出我的解决方案，有更好方案的同学，也可留言或者私聊我贡献给大家。

我们知道，Netty 中将 ChannelHandler 分成 inbound 和 outbound 两种类型，既然是接收新连接，那肯定是 inbound，所以，我们先从 ChannelInboundHandler 这个接口出发，分析它有哪些方法，简单瞄一眼，大致发现有三种可能跟接收新连接有关系的：

- channelRegistered ()/channelUnregistered ()，服务启动的时候有看到 Register 相关的调用
- channelActive ()/channelInactive ()，服务启动绑定完地址的时候有看到 Active 相关的调用
- channelRead ()/channelReadComplete ()，暂未看到在哪里使用的

简单分析一下，上面两种类型在服务启动的时候都有见过它们的源码，所以，不太可能是在接收新连接的时候调用的，因此，只剩下 channelRead ()/channelReadComplete () 这两个方法是有极大可能是在接收新连接的时候调用的，根据名称也能大概猜测是这样的。

所以，我们只需要找到一个 ChannelInboundHandler，把断点打在它的 channelRead () 方法中，就能大概率的跟踪到接收新连接的过程。

通过上一节的分析，我们知道，服务启动完成后，只有一个 Channel，也就是 NioServerSocketChannel，而它对应的 ChannelPipeline 中是有四个 Handler 的，即 head<=>LoggingHandler<=>ServerBootstrapAcceptor<=>tail，所以，我们只要把断点打在这四个 Handler 中的任意一个的 channelRead () 方法中即可。

> 确切地说应该是 ChannelHandlerContext，本系列课程，不严格区分 ChannelHandler 和 ChannelHandlerContext 的语义，大家根据上下文很容易理解即可。但是，要记住 ChannelPipeline 中包含的是 ChannelHandlerContext 的双向链表，而 ChannelHandlerContext 中才是具体的 ChannelHandler。

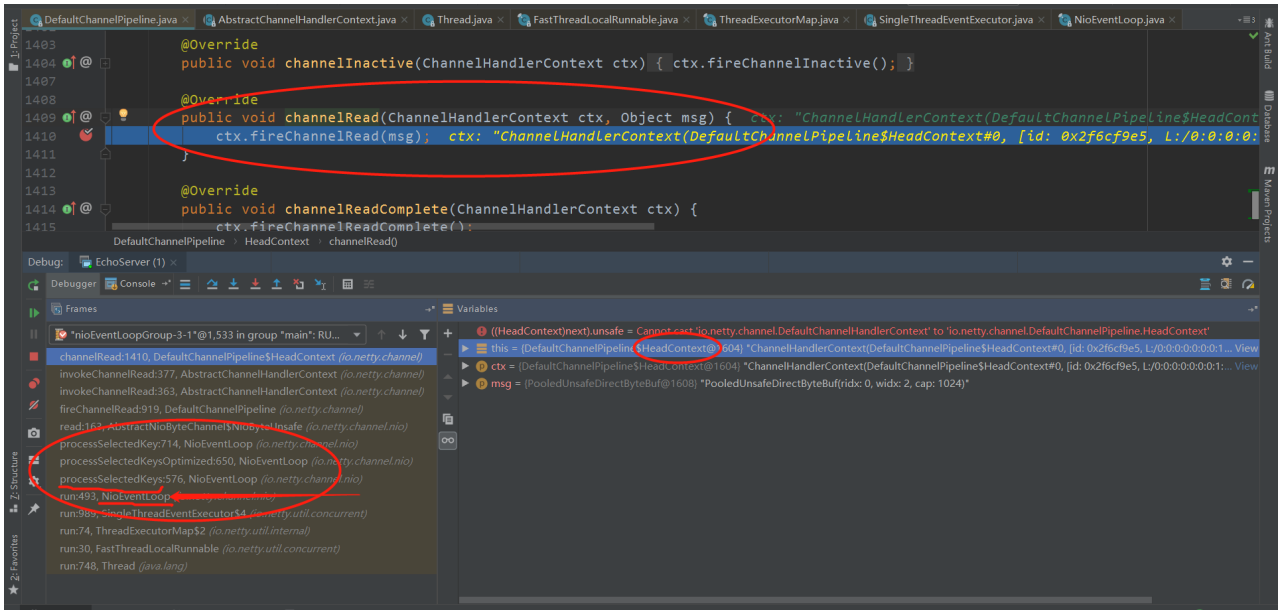
本节，我们就把断点打在 HeadContext 的 channelRead () 方法中。

> HeadContext，即是一个 ChannelHandlerContext，又是一个 ChannelInboundHandler，同时也是 ChannelOutboundHandler。

服务接收新连接过程

OK，在启动服务之前，请先将上一节的所有断点取消掉，待服务启动完成后，再在 HeadContext 的 channelRead () 方法中打一个断点，然后，进入今天的源码剖析阶段。

打开 XSHELL，输入 `telnet localhost 8007`，回车，可以看到，服务端成功停在了 HeadContext 的 channelRead () 方法的断点处：



好了，下面要讲我的独家秘技了 —— **观察线程栈**。观察线程栈，会发现一些非常熟悉的字眼，比如“NioEventLoop”、“run”、“processSelectedKeys”，如果你看不出这么明显的字眼，说明前面的 NIO 部分没有好好学习哦。

> 为了照顾大部分同学，我们再回顾下 NIO 编程的大致过程：
>
> - 先是启动 ServerSocketChannel，并注册 Accept 事件；
> - 轮询调用 Selector 的 select () 方法，select () 方法还有两个兄弟 ——selectNow ()，select (timeout)；
> - 调用 Selector 的 selectedKeys () 方法，拿到轮询到的 SelectionKey；
> - 遍历这些 selectedKeys，处理它们感兴趣到的事件；
> - 如果是 Accept 事件，则从 SelectionKey 中取出 ServerSocketChannel，并 accept () 出来一个 SocketChannel；
> - 把这个 SocketChannel 也注册到 Selector 上，并注册 Read 事件；

所以，这里我们大胆猜测，Netty 底层接收新连接跟 Java 原生 Channel 是一致的，而且它的 select 是在 NioEventLoop 的 run () 方法中，并在获取到 SelectedKeys 之后，调用了 processSelectedKeys () 方法处理这些 SelectionKey。

至于，我们的猜测对不对呢？打开 NioEventLoop 的 run () 方法看一看：

```java
// io.netty.channel.nio.NioEventLoop#run
@Override
protected void run() {
    int selectCnt = 0;
    // 死循环，还记得NIO中的死循环吗？
    for (;;) {
        try {
            int strategy;
            try {
                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()) {
                                // key1，select()相关的方法
                                strategy = select(curDeadlineNanos);
                            }
                        } finally {
                            nextWakeupNanos.lazySet(AWAKE);
                        }
                    default:
                }
            } catch (IOException e) {
                rebuildSelector0();
                selectCnt = 0;
                handleLoopException(e);
                continue;
            }

            selectCnt++;
            cancelledKeys = 0;
            needsToSelectAgain = false;
            final int ioRatio = this.ioRatio;
            boolean ranTasks;
            if (ioRatio == 100) {
                try {
                    if (strategy > 0) {
                        // key2，处理SelectionKey
                        processSelectedKeys();
                    }
                } finally {
                    ranTasks = runAllTasks();
                }
            } else if (strategy > 0) {
                final long ioStartTime = System.nanoTime();
                try {
                    // key2，处理SelectionKey
                    processSelectedKeys();
                } finally {
                    final long ioTime = System.nanoTime() - ioStartTime;
                    ranTasks = runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
                }
            } else {
                ranTasks = runAllTasks(0);
            }
     // 省略其他代码
        }
}
```

从这个方法中，我们可以发现一些关键信息：

- 有个死循环，跟 NIO 编程一模一样；
- 找到了一个看起来像是 select () 的调用 select (curDeadlineNanos)；
- 找到了一个看起来像是处理 SelectionKey 的方法 processSelectedKeys ()；

至于是不是呢，让我们先来看看 `select(curDeadlineNanos)` 方法：

```
// io.netty.channel.nio.NioEventLoop#select
private int select(long deadlineNanos) throws IOException {
    if (deadlineNanos == NONE) {
        // 无限期阻塞
        return selector.select();
    }
    long timeoutMillis = deadlineToDelayNanos(deadlineNanos + 995000L) / 1000000L;
    // 不阻塞的selectNow()和阻塞一段时间的select(timeout)
    return timeoutMillis <= 0 ? selector.selectNow() : selector.select(timeoutMillis);
}
```

可以看到，这里把 select () 的三兄弟都考虑在内了，根据传进来的超时时间判断调用哪个 select () 方法。

再看看 `processSelectedKeys()` 方法：

```
// io.netty.channel.nio.NioEventLoop#processSelectedKeys
private void processSelectedKeys() {
    if (selectedKeys != null) {
        // 我们这里有新连接进来，肯定不为空
        processSelectedKeysOptimized();
    } else {
        processSelectedKeysPlain(selector.selectedKeys());
    }
}
private void processSelectedKeysOptimized() {
    // 遍历SelectionKey
    for (int i = 0; i < selectedKeys.size; ++i) {
        final SelectionKey k = selectedKeys.keys[i];
        // 优化GC
        selectedKeys.keys[i] = null;
        // 取出SelectionKey中的附件，还记得附件是什么吗？
        // 上一节，服务启动的时候把Selector、Java原生Channel、Netty的Channel绑定在一起了
        // 其中，Netty的Channel是通过attachment绑定到SelectionKey中的
        // 所以，针对新连接建立的过程，这里取出来的就是Netty中的NioServerSocketChannel
        final Object a = k.attachment();

        if (a instanceof AbstractNioChannel) {
            // 处理之
            processSelectedKey(k, (AbstractNioChannel) a);
        } else {
            @SuppressWarnings("unchecked")
            NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
            processSelectedKey(k, task);
        }

        if (needsToSelectAgain) {
            selectedKeys.reset(i + 1);
                selectAgain();
                i = -1;
            }
        }
    }
}
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();

    // 省略异常处理等其他代码

    try {
        int readyOps = k.readyOps();

        // 如果是Connect事件
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);
            unsafe.finishConnect();
        }
        // 如果是Write事件
        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            ch.unsafe().forceFlush();
        }
        // 如果是Read事件或者Accept事件
        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
            unsafe.read();
        }
    } catch (CancelledKeyException ignored) {
        unsafe.close(unsafe.voidPromise());
    }
}
```

可以看到，这里的写法也是跟我们的 NIO 编程保持一致的，针对不同的事件使用不同的逻辑进行处理，不同的是，Netty 中具体的处理逻辑交给了 Channel 的 unsafe 来处理，对于接收新连接的过程，这里的 unsafe 无疑就是 NioServerSocketChannel 中的 unsafe 了。

```
// io.netty.channel.nio.AbstractNioMessageChannel.NioMessageUnsafe#read
@Override
public void read() {
    assert eventLoop().inEventLoop();
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();
    allocHandle.reset(config);

    boolean closed = false;
    Throwable exception = null;
    try {
        try {
            do {
                // key1，读取消息
                int localRead = doReadMessages(readBuf);
                if (localRead == 0) {
                    break;
                }
                if (localRead < 0) {
                    closed = true;
                    break;
                }

                allocHandle.incMessagesRead(localRead);
            } while (allocHandle.continueReading());
        } catch (Throwable t) {
            exception = t;
        }

        int size = readBuf.size();
        for (int i = 0; i < size; i ++) {
            readPending = false;
            // key2，触发channelRead()
            pipeline.fireChannelRead(readBuf.get(i));
        }
        readBuf.clear();
        allocHandle.readComplete();
        pipeline.fireChannelReadComplete();

        if (exception != null) {
            closed = closeOnReadError(exception);
            pipeline.fireExceptionCaught(exception);
        }

        if (closed) {
            inputShutdown = true;
            if (isOpen()) {
                close(voidPromise());
            }
        }
    } finally {
        if (!readPending && !config.isAutoRead()) {
            removeReadOp();
        }
    }
}
}
```

从这个方法中我们可以捕捉到两个关键方法：

- doReadMessages (readBuf)，读取消息，如何读取？读取的内容是什么？
- pipeline.fireChannelRead (readBuf.get (i))，触发 ChannelHandler 的 channelRead () 方法，最终由谁处理？

我们先来看第一个方法 `doReadMessages(readBuf)`：

```
// io.netty.channel.socket.nio.NioServerSocketChannel#doReadMessages
@Override
protected int doReadMessages(List<Object> buf) throws Exception {
    // key1，看起来跟Java原生Channel有关系
    SocketChannel ch = SocketUtils.accept(javaChannel());

    try {
        if (ch != null) {
            // key2，构造了一个Netty的NioSocketChannel，并把Java原生SocketChannel传入
            buf.add(new NioSocketChannel(this, ch));
            return 1;
        }
    } catch (Throwable t) {
        logger.warn("Failed to create a new channel from an accepted socket.", t);

        try {
            ch.close();
        } catch (Throwable t2) {
            logger.warn("Failed to close a socket.", t2);
        }
    }

    return 0;
}
// io.netty.util.internal.SocketUtils#accept
public static SocketChannel accept(final ServerSocketChannel serverSocketChannel) throws IOException {
    try {
        return AccessController.doPrivileged(new PrivilegedExceptionAction<SocketChannel>() {
            @Override
            public SocketChannel run() throws IOException {
                // 调用Java原生的aacept()方法创建一个SocketChannel
                return serverSocketChannel.accept();
            }
        });
    } catch (PrivilegedActionException e) {
        throw (IOException) e.getCause();
    }
}
```

可以，Netty 最终还是调用的 Java 原生的 SeverSocketChannel 的 accept () 方法来创建一个 SocketChannel，并把这个 SocketChannel 绑定到 Netty 自己的 NioSocketChannel 中，还记得上一节创建 NioServerSocketChannel 的过程吗？

NioSocketChannel 的创建过程也是一样的，我就不贴源码了，简单再回顾一下：

- 将 Java 原生 Channel 配置成非阻塞 `ch.configureBlocking(false)`；
- 设置感兴趣的事件为 Read 事件，NioServerSocketChannel 感兴趣的事件为 Accept 事件；
- 分配 id；
- 创建 unsafe；
- 创建 ChannelPipeline；

好了，到这里 SocketChannel 就创建好了，但是它的 pipeline 中还是只有 head 和 tail 两个 Handler，还无法处理消息，那么，ChannelPipeline 的初始化在哪里呢？

这就轮到 `pipeline.fireChannelRead(readBuf.get(i))` 这行代码来发挥作用了，这里的 pipeline 实际上是 NioServerSocketChannel 对应的 ChannelPipeline，通过上一节的分析我们知道，服务启动完成后这个 ChannelPipeline 中的双向链表为 `head<=>LoggingHandler<=>ServerBootstrapAcceptor<=>tail`，很显然，只有 ServerBootstrapAcceptor 这个 ChannelHandler 会往子 ChannelPipeline 中添加子 ChannelHandler，所以，我们直接看 ServerBootstrapAcceptor 的 channelRead () 方法即可。

```
// io.netty.bootstrap.ServerBootstrap.ServerBootstrapAcceptor#channelRead
@Override
@SuppressWarnings("unchecked")
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    // 这里的msg就是上面的readBuf.get(i)，也就是NioSocketChannel，也就是子Channel
    final Channel child = (Channel) msg;
    // 添加子ChannelHandler，这里同样也是以ChannelInitializer的形式添加的
    child.pipeline().addLast(childHandler);
    // 设置子Channel的配置等信息
    setChannelOptions(child, childOptions, logger);
    setAttributes(child, childAttrs);

    try {
        // 将子Channel注册到workerGroup中的一个EventLoop上
        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);
    }
}
```

这里的代码跟上一节中 `initAndRegister()` 方法中的逻辑是完全一样的，我们就不再赘述了。

好了，今天的源码剖析基本就讲完了，让我们来总结一下服务接收新连接的过程：

1. Netty 中轮询的方法是写在 NioEventLooop 中的；
2. Netty 底层也是通过 Java 原生 ServerSocketChannel 来接收新连接的；
3. Netty 将接收到的 SocketChannel 包装成了 NioSocketChannel，并给它分配 ChannelPipeline 等元素；
4. Netty 中通过 ServerBootstrapAcceptor 这个 ChannelHandler 来初始化 NioSocketChannel 的配置并将其注册到 EventLoop 中；
5. 注册到 EventLoop 中同样也会与这个 EventLoop 中的 Selector 绑定，Netty 的 NioSocketChannel 同样地也是以附件的形式绑定在 SelectionKey 中；

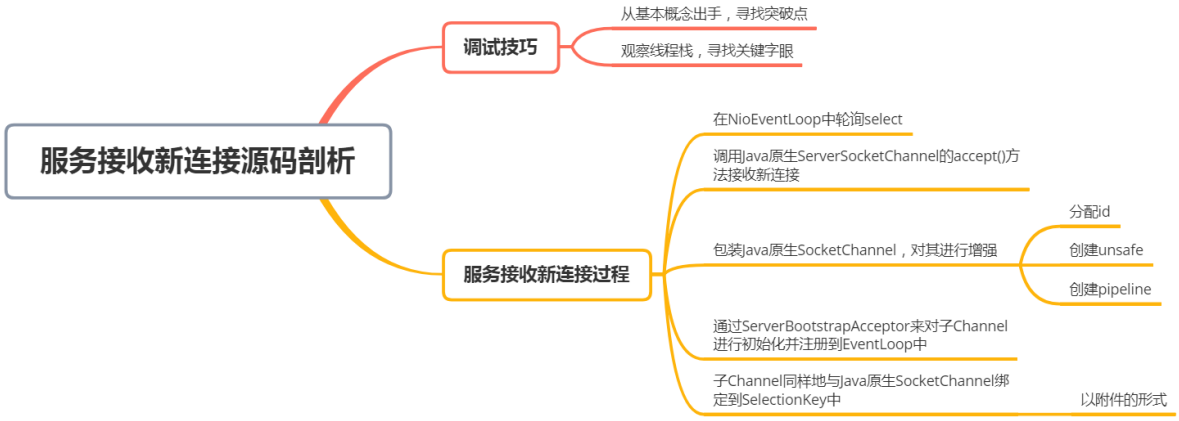
至此，一个 SocketChannel 才算真正建立完成，也就可以接收消息了。

后记

本节，我们一起学习了 Netty 中服务接收新连接过程的源码剖析，相对于上一节，你可能会发现，本节的内容似乎要简单不少，对的，万事开头难，源码剖析也是一样，一开始入门会比较困难，一旦入门了，越往后越简单。

通过本节的剖析，我们知道了 Netty 中接收新连接的过程跟 Java 原生 NIO 是同出一辙，或者说是对 Java 原生 NIO 的增强，那么，既然连接已经就绪，如何接收消息呢？且听下回分解。

思维导图



}

12 Netty服务启动的时候都做了什么

14 Netty服务如何接收新的数据