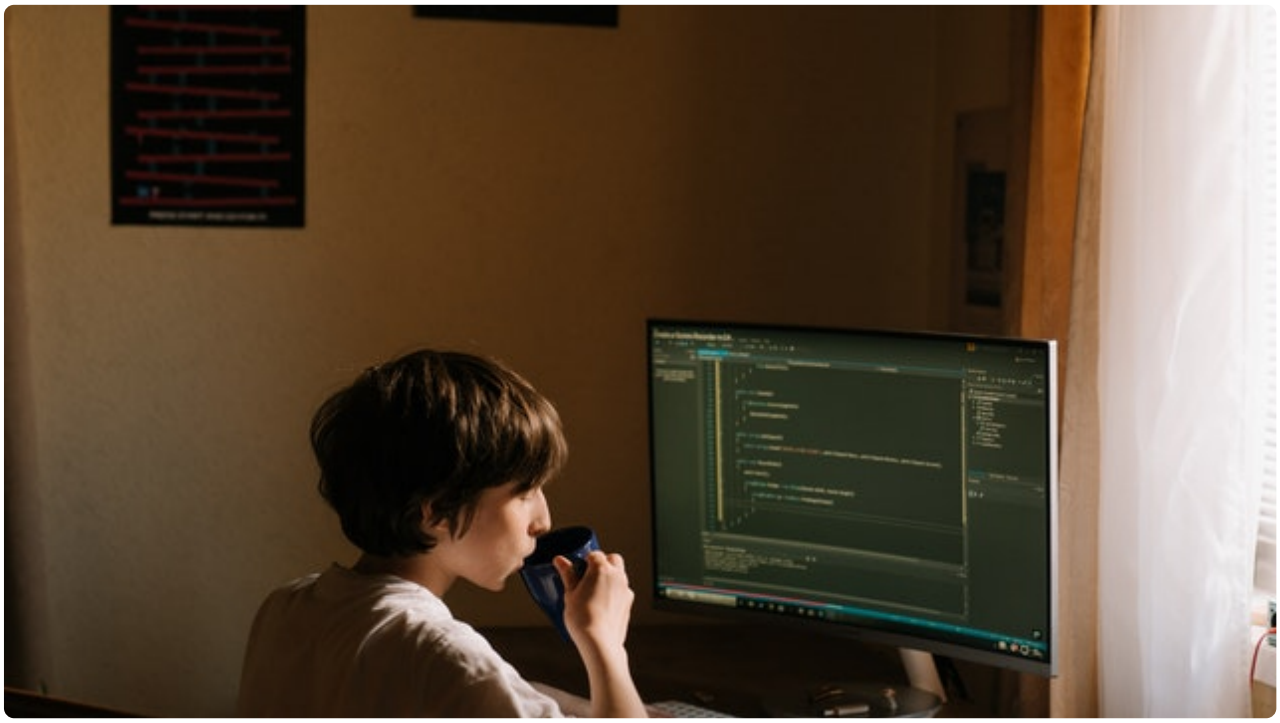
12 Netty服务启动的时候都做了什么

更新时间：2020-08-12 09:38:41



> 一个人追求的目标越高，他的才力就发展得越快，对社会就越有益。——高尔基

前言

你好，我是彤哥。

前面的章节，我们一起学习了 NIO 以及 Netty 的基本知识，通过前面的学习，相信你对 Netty 编程有了一定的了解。

不过，学习一个框架，还是得从源码的角度来深刻领悟它，所以，从本节开始，我将从源码的角度来剖析 Netty。

源码剖析我准备分成两个部分来讲解，第一部分从数据流向的角度剖析源码，这部分源码具有整体性，关联性比较强；第二部分从 Netty 核心知识的角度剖析源码，这部分源码比较散乱，关联性不强，比如，FastThreadLocal，它比 JDK 自带的快在哪里，等等。中间我会穿插着讲一些源码调试的技巧，Java 方面的高阶知识等等。

本节，我们将从服务启动的过程来看看服务启动的时候 Netty 都做了些什么。

好了，让我们一起进入今天的学习吧。

源码剖析的原则

源码剖析有一个非常重要的原则 —— **针对性原则**，当然，这是我起的名字，意思为一定要有一个明确的目标，针对这个特定的目标死磕到底，跟这个目标无关的内容只要看懂大概逻辑就可以了，不能太深陷，否则容易迷失，特别是开源框架的源码，因为要考虑很多兼容性的问题，会有很多奇奇怪怪的代码，针对这些代码，我们可以略微一瞥，或者直接跳过，如果遇到你感兴趣的问题，但又是跟本次目标无关的，可以先记下来，等本次目标完成了，再把记录的问题归类，重新设置新的目标。

> 何为明确的目标？目标一定是可量化的，不能是含糊的，比如，我要看懂 Netty 所有源码，这就是含糊的目标，比如，我要看懂 Netty 服务启动相关的源码，那就相对明确一些，当然，初期能有这么个目标已经不错了。随着研究的深入，会不断发现新的问题，这些新的问题又可以成为新的目标，只有这样才能不断进步。

为了让我们的主题不跑偏，针对源码剖析的部分，每节开头，我将设置几个问题，大家可以带着这些问题来学习。

案例回顾

在正式学习今天的内容之前，我们先来回顾一下之前讲过的 EchoServer：

```
public final class EchoServer {

    static final int PORT = Integer.parseInt(System.getProperty("port", "8007"));

    public static void main(String[] args) throws Exception {
        // 1. 声明线程池
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        EchoServerHandler echoServerHandler = new EchoServerHandler();
        try {
            // 2. 服务端引导器
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            // 3. 设置线程池
            serverBootstrap.group(bossGroup, workerGroup)
                    // 4. 设置ServerSocketChannel的类型
                    .channel(NioServerSocketChannel.class)
                    // 5. 设置参数
                    .option(ChannelOption.SO_BACKLOG, 100)
                    // 6. 设置ServerSocketChannel对应的Handler，只能设置一个
                    .handler(new LoggingHandler(LogLevel.INFO))
                    // 7. 设置SocketChannel对应的Handler
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline p = ch.pipeline();
                            // 可以添加多个子Handler
                            p.addLast(new LoggingHandler(LogLevel.INFO));
                            p.addLast(echoServerHandler);
                        }
                    });

            // 8. 绑定端口
            ChannelFuture f = serverBootstrap.bind(PORT).sync();
            // 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();
        }
    }
}
```

我们后面的源码分析也会基于这个案例来讲解，请记住这里的序号，我们后面还会使用到它们。

问题

通过前面使用案例的学习，我们知道，Netty 启动的时候主要是下面这行代码：

```
ChannelFuture f = serverBootstrap.bind(PORT).sync();
```

这里主要有两个方法，一个是 bind ()，一个是 sync ()，sync () 属于 ChannelFuture 的范畴，今天暂不讨论，所以，今天我们只讨论 bind () 相关的问题：

1. Netty 的 Channel 跟 Java 原生的 Channel 是否有某种关系？
2. bind () 是否调用了 Java 底层的 Socket 相关的操作？
3. Netty 服务启动之后 ChannelPipeline 里面长什么样？

好了，让我们带着这几个问题探索吧。

服务启动过程剖析

让我们将断点打在 `ChannelFuture f = serverBootstrap.bind(PORT).sync();` 这行，以 Debug 模式启动程序，程序会停在此处，按 F7 或者 Shirft+F7 进入 bind () 方法内部：

```
public ChannelFuture bind(int inetPort) {
    return bind(new InetSocketAddress(inetPort));
}
```

可以看到，我们只传进来了一个端口，而使用 `InetSocketAddress` 类构造了一个地址，默认的，会生成一个 `0.0.0.0:8007` 的地址。

接着往下走，会来到一个叫 `doBind()` 的方法，一般地，在开源框架中带 `doXxx` 开头的方法都是干正事的方法。

```
private ChannelFuture doBind(final SocketAddress localAddress) {
    // key1，初始化并注册什么呢？
    final ChannelFuture regFuture = initAndRegister();
    final Channel channel = regFuture.channel();
    // 注册失败
    if (regFuture.cause() != null) {
        return regFuture;
    }
    // 已完成，上面未失败，所以这里是已完成且成功了
    if (regFuture.isDone()) {
        ChannelPromise promise = channel.newPromise();
        // key2，绑定什么呢？取决于initAndRegister()中异步执行的快慢，所以不一定到这里，这里可以打一个断点
        doBind0(regFuture, channel, localAddress, promise);
        return promise;
    } else {
        // 未完成
        final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);
        // 设置回调等待完成，回调内部处理注册的结果
        regFuture.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                Throwable cause = future.cause();
                // 注册失败
                if (cause != null) {
                    promise.setFailure(cause);
                } else {
                    // 注册成功
                    promise.registered();
                    // key2，绑定什么呢？取决于initAndRegister()中异步执行的快慢，所以不一定到这里，这里可以打一个断点
                    doBind0(regFuture, channel, localAddress, promise);
                }
            }
        });
        return promise;
    }
}
```

> 关键方法，我前面加上一个 `key` 表示。

`doBind()` 主要干了两件事：

- initAndRegister ()，初始化并注册什么呢？
- doBind0 ()，到底绑定的是什么？

让我们继续跟进到 `initAndRegister()` 方法中：

```
final ChannelFuture initAndRegister() {
    Channel channel = null;
    try {
        // key1，通过ChannelFactory创建一个Channel，使用反射的形式创建一个Channel
        // 反射的类为我们第4步中设置的NioServerSocketChannel.class
        channel = channelFactory.newChannel();
        // key2，初始化Channel，干了些什么？
        init(channel);
    } catch (Throwable t) {
        // 异常处理，可以不看
        if (channel != null) {
            channel.unsafe().closeForcibly();
            return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);
        }
        return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);
    }
    // key3，注册Channel到哪里？这里有个group()，难道是EventLoopGroup？
    ChannelFuture regFuture = config().group().register(channel);
    // 失败了，关闭Channel
    if (regFuture.cause() != null) {
        if (channel.isRegistered()) {
            channel.close();
        } else {
            channel.unsafe().closeForcibly();
        }
    }

    return regFuture;
}
```

`initAndRegister` 主要干了三个事：

channelFactory.newChannel ()，通过反射的形式创建 Channel，而且是无参构造方法，new 的时候做了哪些事儿？

`init(channel)`，初始化 Channel 的什么？

`register(channel)`，注册 Channel 到哪里？

因为我们这里使用的是 NioServerSocketChannel，所以，直接查看它的无参构造方法即可：

```
// 1. 无参构造方法
public NioServerSocketChannel() {
    this(newSocket(DEFAULT_SELECTOR_PROVIDER));
}
// 1.1 使用Java底层的SelectorProvider创建一个Java原生的ServerSocketChannel
// windows平台下使用的是WindowsSelectorProvider，因为ServerSocketChannel是跟操作系统交互的，所以是平台相关的，每个平台下都不一样
private static ServerSocketChannel newSocket(SelectorProvider provider) {
    try {
        // key，创建Java原生ServerSocketChannel
        return provider.openServerSocketChannel();
    } catch (IOException e) {
        throw new ChannelException(
            "Failed to open a server socket.", e);
    }
}
// 1.2 有参构造方法，参数是Java原生的ServerSocketChannel
public NioServerSocketChannel(ServerSocketChannel channel) {
    // 调用父类的构造方法，注意parent参数为null
    // key，感兴趣的事件为Accept事件
    super(null, channel, SelectionKey.OP_ACCEPT);
    // 创建ChannelConfig
    config = new NioServerSocketChannelConfig(this, javaChannel().socket());
}
```

```
// 1.2.1 调用父类构造方法
protected AbstractNioMessageChannel(Channel parent, SelectableChannel ch, int readInterestOp) {
    super(parent, ch, readInterestOp);
}
// 1.2.1.1 调用父类父类的构造方法
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp) {
    // 调用父类的构造方法，parent为null
    super(parent);
    // ch为Java原生的Channel
    this.ch = ch;
    // 感兴趣的事件，这里为Accept事件
    this.readInterestOp = readInterestOp;
    try {
        // 将channel设置为非阻塞（是不是跟NIO案例中的用法一样？！）
        ch.configureBlocking(false);
    } catch (IOException e) {
        try {
            ch.close();
        } catch (IOException e2) {
            logger.warn(
                "Failed to close a partially initialized socket.", e2);
        }

        throw new ChannelException("Failed to enter non-blocking mode.", e);
    }
}
//  1.2.1.1.1 调用父类父类父类的构造方法
protected AbstractChannel(Channel parent) {
    // 此时parent为null
    this.parent = parent;
    // 赋予一个id
    id = newId();
    // 赋值了一个unsafe，非Java的Unsafe，而是Netty自己的Unsafe
    unsafe = newUnsafe();
    // 创建ChannelPipeline
    pipeline = newChannelPipeline();
}
// 1.2.1.1.1.1 创建默认的ChannelPipeline
protected DefaultChannelPipeline(Channel channel) {
    this.channel = ObjectUtil.checkNotNull(channel, "channel");
    succeededFuture = new SucceededChannelFuture(channel, null);
    voidPromise =  new VoidChannelPromise(channel, true);
    // ChannelPipeline中默认有两个节点，head和tail，且是双向链表
    tail = new TailContext(this);
    head = new HeadContext(this);

    head.next = tail;
    tail.prev = head;
}
```

到这里 NioServerSocketChannel 的创建过程就完毕了，我们简单总结一下：

1. Netty 的 ServerSocketChannel 会与 Java 原生的 ServerSocketChannel 绑定在一起；
2. 会注册 Accept 事件；
3. 会为每一个 Channel 分配一个 id；
4. 会为每一个 Channel 创建一个叫作 unsafe 的东西；
5. 会为每一个 Channel 分配一个 ChannelPipeline；
6. ChannelPipeline 中默认存在一个双向链表 `head<=>tail`；

好了，再来看 `init(channel)` 方法：

```
// io.netty.bootstrap.ServerBootstrap#init
@Override
void init(Channel channel) {
    // 将第5步中设置到ServerBootstrap中的Option设置到Channel的Config中
    setChannelOptions(channel, options0().entrySet().toArray(EMPTY_OPTION_ARRAY), logger);
    // 设置一些属性到Channel中，用法与Option一样
    setAttributes(channel, attrs0().entrySet().toArray(EMPTY_ATTRIBUTE_ARRAY));
    // 从Channel中取出ChannelPipeline，上面创建的
    ChannelPipeline p = channel.pipeline();

    // 子Channel的配置，子Channel也就是SocketChannel
    final EventLoopGroup currentChildGroup = childGroup;
    final ChannelHandler currentChildHandler = childHandler;
    final Entry<ChannelOption<?>, Object>[] currentChildOptions =
        childOptions.entrySet().toArray(EMPTY_OPTION_ARRAY);
    final Entry<AttributeKey<?>, Object>[] currentChildAttrs = childAttrs.entrySet().toArray(EMPTY_ATTRIBUTE_ARRAY);

    // 将ServerBootstrap中的Handler设置到ChannelPipeline的最后
    // ChannelInitializer的实现原理？
    p.addLast(new ChannelInitializer<Channel>() {
        @Override
        public void initChannel(final Channel ch) {
            final ChannelPipeline pipeline = ch.pipeline();
            // 第6步设置的Handler
            ChannelHandler handler = config.handler();
            if (handler != null) {
                pipeline.addLast(handler);
            }

            // 同时，又向ChannelPipeline的最后添加了一个叫作ServerBootstrapAcceptor的Handler
            // 这是什么写法？
            ch.eventLoop().execute(new Runnable() {
                @Override
                public void run() {
                    // 把子Channel相关的参数传到这个Handler里面，那它是干什么的呢？
                    pipeline.addLast(new ServerBootstrapAcceptor(
                        ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
                }
            });
        }
    });
}
```

`init(channel)` 方法整体来说还是比较简单的，就是把 ServerBootstrap 中的配置设置到 Channel 中，不过依然有几处我们现在可能还不太理解的地方：

- ChannelInitializer 的实现原理是什么？
- `ch.eventLoop().execute()` 这是什么写法？
- ServerBootstrapAcceptor 是干什么？

这三个问题，我们留到后面的章节中再解答。

好了，我们再来看看 `initAndRegister()` 方法的最后一个关键步骤，`ChannelFuture regFuture = config().group().register(channel);` 注册 Channel 到什么地方？

查看源码，可以发现，这里的 group 就是我们的 bossGroup，所以这里就是调用 bossGroup 的 `register(channel)` 方法。

```
@Override
public ChannelFuture register(Channel channel) {
    return next().register(channel);
}
```

这里会调用 next() 方法选择出来一个 EventLoop 来注册 Channel，里面实际上使用的是一个叫做 EventExecutorChooser 的东西来选择，它实际上又有两种实现方式 —— PowerOfTwoEventExecutorChooser 和 GenericEventExecutorChooser，本质上就是从 EventExecutor 数组中选择一个 EventExecutor，我们这里就是 NioEventLoop，那么，它们有什么区别呢？有兴趣的可以点开它们的源码看看，我简单地提一下，本质都是按数组长度取余数 ，不过，2 的 N 次方的形式更高效。

最后，来到了 EventLoop 的 register(channel) 方法：

```
// io.netty.channel.SingleThreadEventLoop#register(io.netty.channel.Channel)
@Override
public ChannelFuture register(Channel channel) {
    return register(new DefaultChannelPromise(channel, this));
}

@Override
public ChannelFuture register(final ChannelPromise promise) {
    ObjectUtil.checkNotNull(promise, "promise");
    // key，调用的是channel的unsafe的register()方法
    promise.channel().unsafe().register(this, promise);
    return promise;
}
```

可以看到，先创建了一个叫做 ChannelPromise 的东西，它是 ChannelFuture 的子类，暂时先把它当作 ChannelFuture 来看待。最后，又调回了 Channel 的 Unsafe 的 register () 方法，这里第一个参数是 this，也就是 NioEventLoop，第二个参数是刚创建的 ChannelPromise。

```
// io.netty.channel.AbstractChannel.AbstractUnsafe#register
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
    // 各种检查，可以跳过
    ObjectUtil.checkNotNull(eventLoop, "eventLoop");
    if (isRegistered()) {
        promise.setFailure(new IllegalStateException("registered to an event loop already"));
        return;
    }
    if (!isCompatible(eventLoop)) {
        promise.setFailure(
            new IllegalStateException("incompatible event loop type: " + eventLoop.getClass().getName()));
        return;
    }
    // key1，将上面传进来的EventLoop绑定到Channel上
    AbstractChannel.this.eventLoop = eventLoop;
    // 判断当前线程是否跟EventLoop线程是同一个
    if (eventLoop.inEventLoop()) {
        // key2，调用register0
        register0(promise);
    } else {
        try {
            eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                    // key2，调用register0，实际走的是这里，所以这里需要打个断点
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            // 异常处理，可以跳过
            logger.warn(
                "Force-closing a channel whose registration task was not accepted by an event loop: {}",
                AbstractChannel.this, t);
            closeForcibly();
            closeFuture.setClosed();
            safeSetFailure(promise, t);
        }
    }
}
```

这个方法主要干了两件事：

- 把 EventLoop 与 Channel 绑定在一起；
- 调用 register0 () 方法；

这里又出现了 eventLoop.execute () 这种写法，先忽略它，专注于主要逻辑。

接着，跟踪到 `register0()` 方法中：

```
// io.netty.channel.AbstractChannel.AbstractUnsafe#register0
private void register0(ChannelPromise promise) {
    try {
        // 判断检查，可以跳过
        if (!promise.setUncancellable() || !ensureOpen(promise)) {
            return;
        }
        boolean firstRegistration = neverRegistered;
        // key1，调用doRegister()方法
        doRegister();
        neverRegistered = false;
        registered = true;

        // key2，调用invokeHandlerAddedIfNeeded()
        // 触发添加Handler的回调，其中pineline.addLast(ChannelInitializer)的处理就是在这一步完成的
        // 这一步之后pipeline里面应该是head<=>LoggineHandler<=>tail
        // 而ServerBootstrapAcceptor还没有加入到pipeline中，
        // 因为它设置了使用EventLoop的线程执行，当前线程就是EventLoop的线程
        // 所以，添加ServerBootstrapAcceptor会在当前任务执行完毕才会执行
        pipeline.invokeHandlerAddedIfNeeded();

        safeSetSuccess(promise);
        // 调用ChannelPineline的fireChannelRegistered()，实际是调用的各个ChannelHandler的channelRegistered()方法
        pipeline.fireChannelRegistered();
        // Channel是否已经激活，此时还未绑定到具体的地址，所以还未激活
        if (isActive()) {
            if (firstRegistration) {
                pipeline.fireChannelActive();
            } else if (config().isAutoRead()) {
                beginRead();
            }
        }
    } catch (Throwable t) {
        // 异常处理，可以跳过
        closeForcibly();
        closeFuture.setClosed();
        safeSetFailure(promise, t);
    }
}
```

这里有两个个非常重要的方法：

- doRegister ()，一看就是干正事的方法
- pipeline.invokeHandlerAddedIfNeeded ()，触发添加 Handler 的回调，其中 pineline.addLast (ChannelInitializer) 的处理就是在这一步完成的，有兴趣的同学可以跟踪看一下，这一块我们本节不详细展开

先来看 `doRegister()` 方法：

```
// io.netty.channel.nio.AbstractNioChannel#doRegister
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            // key，将EventLoop中的Selector与Java原生的Channel绑定在一起，并返回这个SelectionKey
            // 注意，第三个参数是this，代表的是当前这个Netty中的Channel，我们这里就是NioServerSocketChannel
            // 它作为Selection的attachment绑定到SelectionKey上，与JavaChannel和Selector是同一个级别的
            selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);
            return;
        } catch (CancelledKeyException e) {
            // 异常处理，可以跳过
            if (!selected) {
                eventLoop().selectNow();
                selected = true;
            } else {
                throw e;
            }
        }
    }
}
```

这里其实就一行关键代码，将 Selector 与 Java 原生 Channel 绑定在一起，并将当前 Netty 的 Channel 通过 attachment 的形式绑定到 SelectionKey 上，到这里，你可能会有疑问：为什么要把 Netty 的 Channel 当作附件放到 SelectionKey 中呢？后面你会知道的，相信我。

所以，整个注册的过程主要就干了三个事：

1. 把 Channel 绑定到一个 EventLoop 上；
2. 把 Java 原生 Channel、Netty 的 Channel、Selector 绑定到 SelectionKey 中；
3. 触发 Register 相关的事件；

至此，`initAndRegister()` 方法内部就分析完成了，我们再来看看另一个重要方法 `doBind0()`：

```
// 1. io.netty.bootstrap.AbstractBootstrap#doBind0
private static void doBind0(
    final ChannelFuture regFuture, final Channel channel,
    final SocketAddress localAddress, final ChannelPromise promise) {

    // 异步执行
    channel.eventLoop().execute(new Runnable() {
        @Override
        public void run() {
            if (regFuture.isSuccess()) {
                // key，调用Channel的bind()方法，因为在线程池里面，所以这里要打一个断点
                channel.bind(localAddress, promise).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            } else {
                promise.setFailure(regFuture.cause());
            }
        }
    });
}
// 2. 调用Channel的bind()方法 io.netty.channel.AbstractChannel#bind(java.net.SocketAddress, io.netty.channel.ChannelPromise)
@Override
public ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {
    // 调用的是pipeline的bind()方法
    return pipeline.bind(localAddress, promise);
}
// 3. 调用的是pipeline的bind()方法io.netty.channel.DefaultChannelPipeline#bind(java.net.SocketAddress, io.netty.channel.ChannelPromise)
@Override
public final ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {
    // 从尾开始调用，也就是outbound
    return tail.bind(localAddress, promise);
```

```
}
// 4. 此时pipeline中的Handler为head<=>LoggingHandler<=>ServerBootstrapAcceptor<=>tail，出站的pineple实际为tail=>LoggingHandler=>head，下面我只贴主要代码
// 5. io.netty.handler.logging.LoggingHandler#bind
@Override
public void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) throws Exception {
    if (logger.isEnabled(internalLevel)) {
        logger.log(internalLevel, format(ctx, "BIND", localAddress));
    }
    ctx.bind(localAddress, promise);
}
// 6. io.netty.channel.DefaultChannelPipeline.HeadContext#bind
@Override
public void bind(
    ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) {
    // 最后调用的是HeadContext这个Handler中unsafe的bind()方法
    unsafe.bind(localAddress, promise);
}
// 7. io.netty.channel.AbstractChannel.AbstractUnsafe#bind
@Override
public final void bind(final SocketAddress localAddress, final ChannelPromise promise) {
    // 省略其它代码

    boolean wasActive = isActive();
    try {
        // key，绑定地址
        doBind(localAddress);
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        closeIfClosed();
        return;
    }
 // 成功激活，调用pipeline.fireChannelActive()方法
    if (!wasActive && isActive()) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                pipeline.fireChannelActive();
            }
        });
    }
    // 设置promise为成功状态
    safeSetSuccess(promise);
}
// 8. 绕了一圈，最后又回到了NioServerChannel的doBind()方法 io.netty.channel.socket.nio.NioServerSocketChannel#doBind
@SuppressJava6Requirement(reason = "Usage guarded by java version check")
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    // 根据不同的JDK版本调用不同的方法
    if (PlatformDependent.javaVersion() >= 7) {
        // 我使用的JDK8版本，所以走到这里了
        javaChannel().bind(localAddress, config.getBacklog());
    } else {
        javaChannel().socket().bind(localAddress, config.getBacklog());
    }
}
```

可以看到，`doBind0()` 最后也是通过 Java 原生 Channel 的 bind () 方法来实现的。

最后，我们来总结一下整个服务启动的过程，服务启动主要是通过两个主要的大方法来完成的：

1. initAndRegister ()，初始化并注册什么呢？
    - channelFactory.newChannel()

        - 通过反射创建一个 NioServerSocketChannel
        - 将 Java 原生 Channel 绑定到 NettyChannel 中
        - 注册 Accept 事件
        - 为 Channel 分配 id
        - 为 Channel 创建 unsafe
        - 为 Channel 创建 ChannelPipeline（默认是 head<=>tail 的双向链表）
    - init(channel)
        - 把 ServerBootstrap 中的配置设置到 Channel 中
        - 添加 ServerBootstrapAcceptor 这个 Handler
    - register(channel)
        - 把 Channel 绑定到一个 EventLoop 上
        - 把 Java 原生 Channel、Netty 的 Channel、Selector 绑定到 SelectionKey 中
        - 触发 Register 相关的事件
2. doBind0 ()，到底绑定的是什么？
    - 通过 Java 原生 Channel 绑定到一个本地地址上

好了，经过本节的学习，我想你一定会发现很多新的问题，比如：

1. ChannelInitializer 是怎么实现的？
2. ch.eventLoop ().execute () 这种写法是在干什么？
3. ServerBootstrapAcceptor 是干什么的？
4. Netty 使用的还是 Java 原生的 Channel，那么，Selector 在哪里用的？

等等。

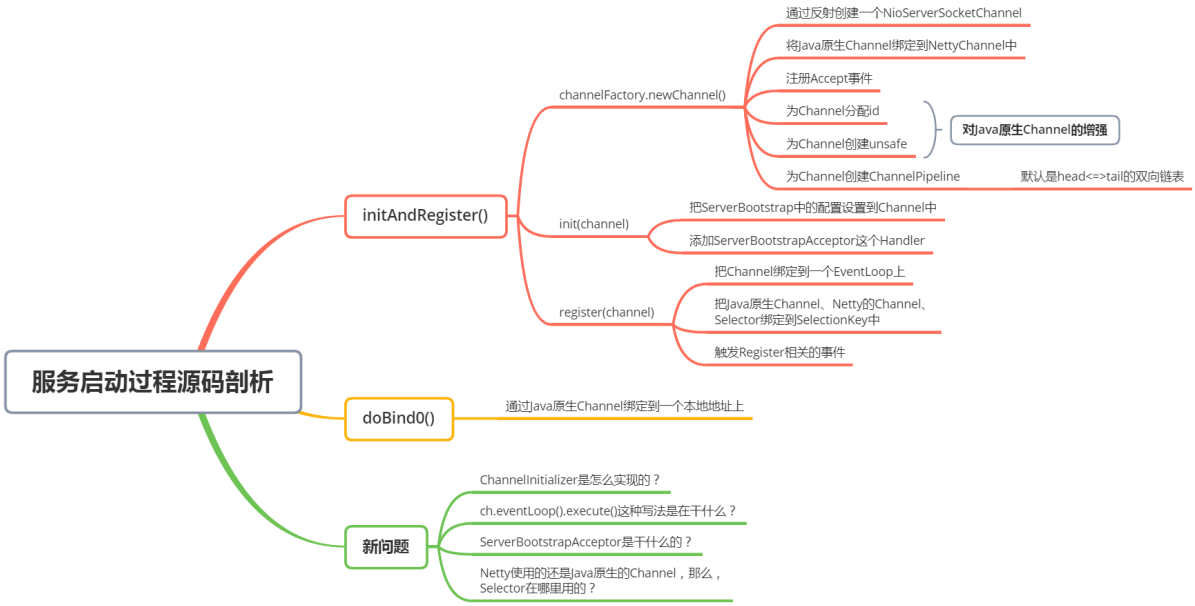
能发现问题是好事，有些问题会在后面的章节中涉及，有些问题需要你自己解决，我相信你一定可以做到的。

后记

本节，我们一起学习了 Netty 服务启动的过程，通过源码的阅读，可以看到，Netty 源码的调用链也是非常长的，在源码阅读的过程中，肯定会发现一些新的问题，随着我们源码阅读的不断深入，这些问题我们也会一一攻破。

下一节，我们将一起学习 Netty 中连接建立过程的源码剖析，敬请期待。

思维导图



}

11 Netty是如何支持常见的编解码方式的

13 Netty服务如何接收新的连接