我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro

文章目录

（二）之客户端

常使用 Netty 主要使用 NIO 部分，所以本文也只分享 Netty NIO

们在《精尽 Netty 源码分析 —— 调试环境搭建》搭建的

```
("ssl") != null;
("host", "127.0.0.1");
em.getProperty("port", "8007"));
em.getProperty("size", "256"));

 Exception {
```

```
    // 配置 SSL
    final SslContext sslCtx;
    if (SSL) {
        sslCtx = SslContextBuilder.forClient()
            .trustManager(InsecureTrustManagerFactory.INSTANCE).build();
    } else {
        sslCtx = null;
    }

    // Configure the client.
    // 创建一个 EventLoopGroup 对象
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        // 创建 Bootstrap 对象
        Bootstrap b = new Bootstrap();
        b.group(group) // 设置使用的 EventLoopGroup
         .channel(NioSocketChannel.class) // 设置要被实例化的为 NioSocketChannel 类
         .option(ChannelOption.TCP_NODELAY, true) // 设置 NioSocketChannel 的可选项
         .handler(new ChannelInitializer<SocketChannel>() { // 设置 NioSocketChannel 的处理器
             @Override
```

```
            public void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline p = ch.pipeline();
                if (sslCtx != null) {
                    p.addLast(sslCtx.newHandler(ch.alloc(), HOST, PORT));
                }
                //p.addLast(new LoggingHandler(LogLevel.INFO));
                p.addLast(new EchoClientHandler());
            }
        });
```

文章目录

端
T).sync();
d.
ate all threads.

ap 抽象类，用于 Client 的启动器实现类。

3.1 构造方法

```
/**
 * 默认地址解析器对象
 */
private static final AddressResolverGroup<?> DEFAULT_RESOLVER = DefaultAddressResolverGroup.INSTANCE;

/**
 * 启动类配置对象
 */
private final BootstrapConfig config = new BootstrapConfig(this);
/**
 * 地址解析器对象
 */
@SuppressWarnings("unchecked")
private volatile AddressResolverGroup<SocketAddress> resolver = (AddressResolverGroup<SocketAddress>)
/**
 * 连接地址
 */
private volatile SocketAddress remoteAddress;
```

```
public Bootstrap() { }

private Bootstrap(Bootstrap bootstrap) {
    super(bootstrap);
    resolver = bootstrap.resolver;
    remoteAddress = bootstrap.remoteAddress;
}
```

文章目录

用 DEFAULT_RESOLVER 即可。

，设置 resolver 属性。代码如下：

```
 resolver) {
ddress>) (resolver == null ? DEFAULT_RESOLVER : resol
```

。代码如下：

```
 resolver) {
ddress>) (resolver == null ? DEFAULT_RESOLVER : resol
```

```
public Bootstrap remoteAddress(SocketAddress remoteAddress) {
    this.remoteAddress = remoteAddress;
    return this;
}

public Bootstrap remoteAddress(String inetHost, int inetPort) {
    remoteAddress = InetSocketAddress.createUnresolved(inetHost, inetPort);
    return this;
}

public Bootstrap remoteAddress(InetAddress inetHost, int inetPort) {
    remoteAddress = new InetSocketAddress(inetHost, inetPort);
    return this;
}
```

3.4 validate

#validate() 方法，校验配置是否正确。代码如下：

```
@Override
public Bootstrap validate() {
    // 父类校验
    super.validate();
    // handler 非空
    if (config.handler() == null) {
        throw new IllegalStateException("handler not set");
    }
```

文章目录

法进行校验。

o 构造方法，克隆一个 Bootstrap 对象。差别在于，下面的方

#connect(...) 方法，连接服务端，即启动客户端。代码如下：

```
public ChannelFuture connect() {
    // 校验必要参数
    validate();
    SocketAddress remoteAddress = this.remoteAddress;
    if (remoteAddress == null) {
        throw new IllegalStateException("remoteAddress not set");
    }
    // 解析远程地址，并进行连接
    return doResolveAndConnect(remoteAddress, config.localAddress());
}

public ChannelFuture connect(String inetHost, int inetPort) {
    return connect(InetSocketAddress.createUnresolved(inetHost, inetPort));
}

public ChannelFuture connect(InetAddress inetHost, int inetPort) {
    return connect(new InetSocketAddress(inetHost, inetPort));
```

```
}

public ChannelFuture connect(SocketAddress remoteAddress) {
    // 校验必要参数
    validate();
    if (remoteAddress == null) {
        throw new NullPointerException("remoteAddress");
    }
    // 解析远程地址，并进行连接
```

文章目录

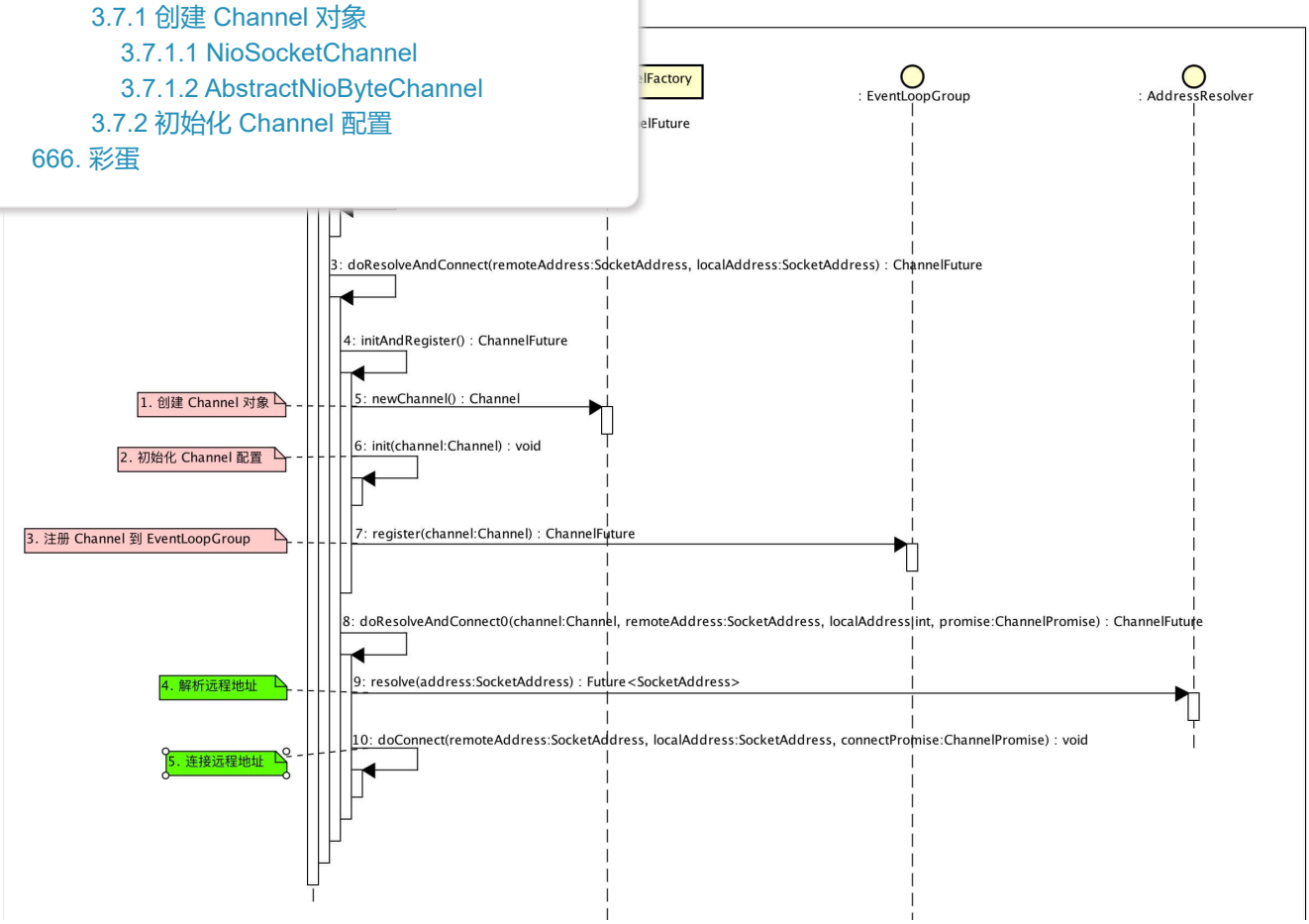
fig.localAddress());

Address, SocketAddress localAddress) {

ress");

alAddress);

务端，启动客户端。如果需要**同步**，则需要调用



- 主要有 5 个步骤，下面我们来拆解代码，看看和我们在 《精尽 Netty 源码分析 —— NIO 基础（五）之示例》 的 NioClient 的代码，是**怎么对应**的。
- 相比 `#bind(...)` 方法的流程，主要是**绿色**的 2 个步骤。

3.6.1 doResolveAndConnect

`#doResolveAndConnect(final SocketAddress remoteAddress, final SocketAddress localAddress)` 方法，代码如下：


```
al SocketAddress remoteAddress, final SocketAddress l
异步的过程，所以返回一个 ChannelFuture 对象。
gister();
();
remoteAddress, localAddress, channel.newPromise());
ys fulfilled already, but just in case it's not.
ise = new PendingRegistrationPromise(channel);
reListener() {
nnelFuture future) throws Exception {
and do a null check so we only need one volatile read
e();
entLoop failed so fail the ChannelPromise directly to
once we try to access the EventLoop of the Channel.
26:                     promise.setFailure(cause);
27:                 } else {
28:                     // Registration was successful, so set the correct executor to use.
29:                     // See https://github.com/netty/netty/issues/2586
30:                     promise.registered();
31:
32:                     // 解析远程地址，并进行连接
33:                     doResolveAndConnect0(channel, remoteAddress, localAddress, promise);
34:                 }
35:             }
36:
37:         });
38:         return promise;
39:     }
40: }
```

- 第 3 行：调用 `#initAndRegister()` 方法，初始化并注册一个 Channel 对象。因为注册是**异步**的过程，所以返回一个 ChannelFuture 对象。详细解析，见 「3.7 initAndRegister」 。
  - 第 6 至 10 行：若执行失败，直接进行返回 `regFuture` 对象。

- 第 9 至 37 行：因为注册是**异步**的过程，有可能已完成，有可能未完成。所以实现代码分成了【第 12 行】和【第 13 至 37 行】分别处理已完成和未完成的情况。
  - **核心**在【第 12 行】或者【第 33 行】的代码，调用 `#doResolveAndConnect0(final Channel channel, SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise)` 方法，解析远程地址，并进行连接。
  - 如果**异步**注册对应的 ChanelFuture 未完成，则调用 `ChannelFuture#addListener(ChannelFutureListener)` 方法，添加监听器，在**注册**完成后，进行回调执行 `#doResolveAndConnect0(...)` 方法的逻辑。详细解析，见「3.6.2 doResolveAndConnect0」。

所以总结来说，**resolve 和 connect 的逻辑，执行在 register 的逻辑之后。**

「3.7 initAndRegister」的内容在回过头来 `...)` 方法的执行，在

进行连接。代码如下：

```
nal Channel channel, SocketAddress remoteAddress,
nal SocketAddress localAddress, final ChannelPromise

ventLoop();
resolver = this.resolver.getResolver(eventLoop);

ss) || resolver.isResolved(remoteAddress)) {
t to do with the specified remote address or it's res
ress, promise);
```

```
13:         // 解析远程地址
14:         final Future<SocketAddress> resolveFuture = resolver.resolve(remoteAddress);
15:
16:         if (resolveFuture.isDone()) {
17:             // 解析远程地址失败，关闭 Channel ，并回调通知 promise 异常
18:             final Throwable resolveFailureCause = resolveFuture.cause();
19:             if (resolveFailureCause != null) {
20:                 // Failed to resolve immediately
21:                 channel.close();
22:                 promise.setFailure(resolveFailureCause);
23:             } else {
24:                 // Succeeded to resolve immediately; cached? (or did a blocking lookup)
25:                 // 连接远程地址
26:                 doConnect(resolveFuture.getNow(), localAddress, promise);
27:             }
28:             return promise;
29:         }
30:
31:         // Wait until the name resolution is finished.
32:         resolveFuture.addListener(new FutureListener<SocketAddress>() {
```

```
33:             @Override
34:             public void operationComplete(Future<SocketAddress> future) throws Exception {
35:                 // 解析远程地址失败，关闭 Channel ，并回调通知 promise 异常
36:                 if (future.cause() != null) {
37:                     channel.close();
38:                     promise.setFailure(future.cause());
39:                 // 解析远程地址成功，连接远程地址
40:                 } else {
                        doConnect(future.getNow(), localAddress, promise);
```

文章目录

是**异步**的过程，所以返回一个 Future 对象。

点，所以暂时省略。😈 老艿艿猜测胖友应该也暂时不感兴

有可能未完成。所以实现代码分成了【第 16 至 29 行】和

oConnect(...) 方法，连接远程地址。

ddListener(FutureListener) 方法，添加监听器，在

的逻辑。详细解析，见 见 「3.13.3 doConnect」 。

辑之后。

【第 16 至 30 行】的条件，即无需走**异步**的流程。

辑。代码如下：

```
1: private static void doConnect(final SocketAddress remoteAddress, final SocketAddress localAddress,
2:
3:     // This method is invoked before channelRegistered() is triggered.  Give user handlers a chanc
4:     // the pipeline in its channelRegistered() implementation.
5:     final Channel channel = connectPromise.channel();
6:     channel.eventLoop().execute(new Runnable() {
7:
8:         @Override
```

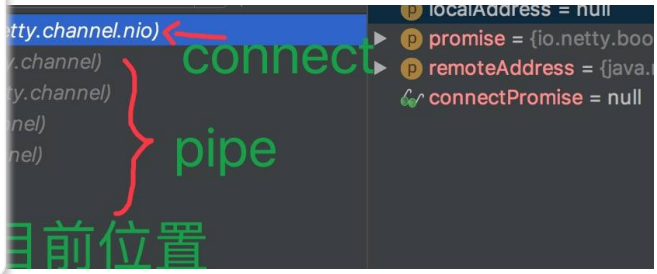
文章目录


```
, connectPromise);

, localAddress, connectPromise);

lFutureListener.CLOSE_ON_FAILURE);
```

辑。但是，实际上当前线程已经是 EventLoop 所在的线程
注释。感叹句，Netty 虽然代码量非常庞大且复杂，但是英文
描述，也非常健全。
行 Channel 连接远程地址的逻辑。后续的方法栈调用如下



- 还是老样子，我们先省略掉 pipeline 的内部实现代码，从 `AbstractNioUnsafe#connect(final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise)` 方法，继续向下分享。

`AbstractNioUnsafe#connect(final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise)` 方法，执行 Channel 连接远程地址的逻辑。代码如下：

```
1: @Override
2: public final void connect(final SocketAddress remoteAddress, final SocketAddress localAddress, fin
3:     if (!promise.setUncancellable() || !ensureOpen(promise)) {
4:         return;
5:     }
6:
7:     try {
8:         // 目前有正在连接远程地址的 ChannelPromise ，则直接抛出异常，禁止同时发起多个连接。
9:         if (connectPromise != null) {
10:            // Already a connect in process.
11:            throw new ConnectionPendingException();
12:        }
13:
```

```java
14:         // 记录 Channel 是否激活
15:         boolean wasActive = isActive();
16: 
17:         // 执行连接远程地址
18:         if (doConnect(remoteAddress, localAddress)) {
19:             fulfillConnectPromise(promise, wasActive);
20:         } else {
21:             // 记录 connectPromise
22:             connectPromise = promise;
...
ldress;
...
听连接远程地址超时。若连接超时，则回调通知 connectPromise
...
().getConnectTimeoutMillis(); // 默认 30 * 1000 毫秒
...
oop().schedule(new Runnable() {
...
tPromise = AbstractNioChannel.this.connectPromise;
on cause = new ConnectTimeoutException("connection ti
 null && connectPromise.tryFailure(cause)) {
());
...
Unit.MILLISECONDS);
...
tureListener() {
...
(ChannelFuture future) throws Exception {
 {
48:                     if (connectTimeoutFuture != null) {
49:                         connectTimeoutFuture.cancel(false);
50:                     }
51:                     // 置空 connectPromise
52:                     connectPromise = null;
53:                     close(voidPromise());
54:                 }
55:             }
56:         });
57:     }
58:     } catch (Throwable t) {
59:         // 回调通知 promise 发生异常
60:         promise.tryFailure(annotateConnectException(t, remoteAddress));
61:         closeIfClosed();
62:     }
63: }
```

文章目录

- 第 8 至 12 行：目前有正在连接远程地址的 ChannelPromise ，则直接抛出异常，禁止同时发起多个连接。 `connectPromise` 变量，定义在 AbstractNioChannel 类中，代码如下：

```
/**
 * 目前正在连接远程地址的 ChannelPromise 对象。
 *
 * The future of the current connection attempt.  If not null, subsequent
 * connection attempts will fail.
 */
private ChannelPromise connectPromise;
```


激活。NioSocketChannel 对该方法的实现代码如下：

此时，一般返回的是 `false` 。

dress, SocketAddress localAddress) 方法，执行连接

```
 remoteAddress, SocketAddress localAddress) throws E
```

功

onnect(javaChannel(), remoteAddress);

NECT)事件。

```
13:         if (!connected) {
14:             selectionKey().interestOps(SelectionKey.OP_CONNECT);
15:         }
16:         // 标记执行是否成功
17:         success = true;
18:         // 返回是否连接完成
19:         return connected;
20:     } finally {
21:         // 执行失败，则关闭 Channel
22:         if (!success) {
23:             doClose();
24:         }
25:     }
26: }
```

- 第 3 至 6 行：若 `localAddress` 非空，则调用 `#doBind0(SocketAddress)` 方法，绑定本地地址。一般情况下，NIO Client 是不需要绑定本地地址的。默认情况下，系统会随机分配一个可用的本地地址，进行绑定。
- 第 11 行：调用 `SocketUtils#connect(SocketChannel socketChannel, SocketAddress remoteAddress)` 方法，Java 原生 NIO SocketChannel 连接 远程地址，并返回是否连接完成(成功)。代码如

下：

```
public static boolean connect(final SocketChannel socketChannel, final SocketAddress remoteAd
    try {
        return AccessController.doPrivileged(new PrivilegedExceptionAction<Boolean>() {
            @Override
            public Boolean run() throws IOException {
                return socketChannel.connect(remoteAddress);
```

文章目录

作包在 AccessController 中呢？我们来看下 SocketUtils

```
ges enabled. This is necessary for applications
ink SocketPermission} to their application. By
ns can proceed even if some code in the calling
```

解。

键字，或者阅读 《Java 安全模型介绍》 。

= false)时，我们可以看到，调用

事件(SelectionKey.OP_CONNECT)为感兴趣的事件。也

应的 Selector 将会轮询到该事件，可以进一步处理。

，调用 #doClose() 方法，关闭 Channel 。

nnel socketChannel, SocketAddress remoteAddress) 方法的结果为 false ，所以不会执行【第 19 行】代码的 #fulfillConnectPromise(ChannelPromise promise, boolean wasActive) 方法，而是执行【第 20 至 57 行】的代码逻辑。

- 第 22 行：记录 connectPromise 。
- 第 24 行：记录 requestedRemoteAddress 。 requestedRemoteAddress 变量，在 AbstractNioChannel 类中定义，代码如下：

```
/**
 * 正在连接的远程地址
 */
private SocketAddress requestedRemoteAddress;
```

- 第 26 至 40 行：调用 EventLoop#schedule(Runnable command, long delay, TimeUnit unit) 方法，发起定时任务 connectTimeoutFuture ，监听连接远程地址**是否超时**。若连接超时，则回调通知 connectPromise 超时异常。 connectPromise 变量，在 AbstractNioChannel 类中定义，代码如下：

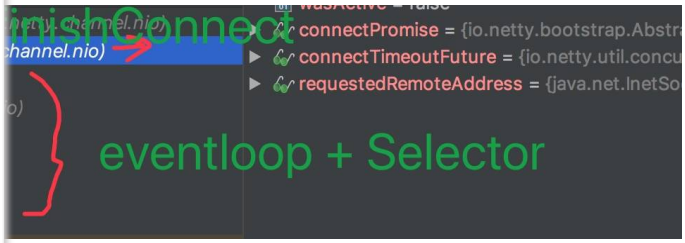
```
/**
 * 连接超时监听 ScheduledFuture 对象。
```

```
     */
    private ScheduledFuture<?> connectTimeoutFuture;
```

- 第 42 至 57 行：调用 `ChannelPromise#addListener(ChannelFutureListener)` 方法，添加监听器，监听连接远程地址**是否取消**。若取消，则取消 `connectTimeoutFuture` 任务，并置空 `connectPromise` 。这样，客户端 Channel 可以发起下一次连接。

3.6.4 finishConnect

答案在 `AbstractNioUnsafe#finishConnect()` 方法中。而事件时，进行触发。调用栈如下图：



实现代码，从 `AbstractNioUnsafe#finishConnect()` 方法，

端的连接。代码如下：

```
nt loop only if the connection attempt was
```

```
12:         doFinishConnect();
13:         // 通知 connectPromise 连接完成
14:         fulfillConnectPromise(connectPromise, wasActive);
15:     } catch (Throwable t) {
16:         // 通知 connectPromise 连接异常
17:         fulfillConnectPromise(connectPromise, annotateConnectException(t, requestedRemoteAddress))
18:     } finally {
19:         // 取消 connectTimeoutFuture 任务
20:         // Check for null as the connectTimeoutFuture is only created if a connectTimeoutMillis >
21:         // See https://github.com/netty/netty/issues/1770
22:         if (connectTimeoutFuture != null) {
23:             connectTimeoutFuture.cancel(false);
24:         }
25:         // 置空 connectPromise
26:         connectPromise = null;
27:     }
28: }
```

- 第 6 行：判断是否在 EventLoop 的线程中。

- 第 10 行：调用 `#isActive()` 方法，获得 Channel 是否激活。笔者调试时，此时返回 `false` ，因为连接还没完成。
- 第 12 行：调用 `#doFinishConnect()` 方法，执行完成连接的逻辑。详细解析，见「3.6.4.1 doFinishConnect」。
- 第 14 行：执行完成连接**成功**，调用 `#fulfillConnectPromise(ChannelPromise promise, boolean wasActive)` 方法，通知 `connectPromise` 连接完成。详细解析，见「3.6.4.2 fulfillConnectPromise 成功」。
- 第 15 至 17 行：执行完成连接**异常**，调用 `#fulfillConnectPromise(ChannelPromise promise, Throwable cause)` 方法，通知 `connectPromise` 连接异常。详细解析，见「3.6.4.3 fulfillConnectPromise 异常」。
- 第 18 至 27 行：执行完成连接**结束**，取消 `connectTimeoutFuture` 任务，并置空 `connectPromise` 。

文章目录

连接的逻辑。代码如下：

```
) {
```

shConnect() 方法，完成连接。😈 美滋滋。

romise promise, Throwable cause) 方法，通知

```
mise promise, boolean wasActive) {

omise has been notified already.
```

```
 8:     // Get the state as trySuccess() may trigger an ChannelFutureListener that will close the Chan
 9:     // We still need to ensure we call fireChannelActive() in this case.
10:     boolean active = isActive();
11:
12:     // 回调通知 promise 执行成功
13:     // trySuccess() will return false if a user cancelled the connection attempt.
14:     boolean promiseSet = promise.trySuccess();
15:
16:     // 若 Channel 是新激活的，触发通知 Channel 已激活的事件。
17:     // Regardless if the connection attempt was cancelled, channelActive() event should be trigger
18:     // because what happened is what happened.
19:     if (!wasActive && active) {
20:         pipeline().fireChannelActive();
21:     }
22:
23:     // If a user cancelled the connection attempt, close the channel, which is followed by channel
24:     // TODO 芋艿
25:     if (!promiseSet) {
26:         close(voidPromise());
```

```
27:     }
28: }
```

- 第 10 行：调用 `#isActive()` 方法，获得 Channel 是否激活。笔者调试时，此时返回 `true` ，因为连接已经完成。
- 第 14 行：回调通知 `promise` 执行成功。此处的通知，对应回调的是我们添加到 `#connect(...)` 方法返回的 ChannelFuture 的 ChannelFutureListener 的监听器。示例代码如下：

文章目录


```
tener(new ChannelFutureListener() { // 回调的就是我!

e future) throws Exception {
```

```
= true ，因此，Channel 可以认为是新激活的，满足【第

nelActive() 方法，触发 Channel 激活的事件。【重
也就说，会调用到 AbstractUnsafe#beginRead() 方
时，设置的 readInterestOp =
地，客户端可以读取服务端发送来的数据。
见《精尽 Netty 源码分析 —— 启动（一）之服务端》的

romiseSet

hrowable cause) 方法，通知 connectPromise 连接异常。
```

```
e promise, Throwable cause) {

        // Closed via cancellation and the promise has been notified already.
        return;
    }

    // 回调通知 promise 发生异常
    // Use tryFailure() instead of setFailure() to avoid the race against cancel().
    promise.tryFailure(cause);
    // 关闭
    closeIfClosed();
}
```

- 比较简单，已经添加中文注释，胖友自己查看。

3.7 initAndRegister

Bootstrap 继承 AbstractBootstrap 抽象类，所以 `#initAndRegister()` 方法的流程上是一致的。所以和 ServerBootstrap 的差别在于：

1. 创建的 Channel 对象不同。

2. 初始化 Channel 配置的代码实现不同。

3.7.1 创建 Channel 对象

考虑到本文的内容，我们以 NioSocketChannel 的创建过程作为示例。创建 NioSocketChannel 对象的流程，和 NioServerSocketChannel 基本是一致的，所以流程图我们就不提供了，直接开始撸源码。

3.7.1.1 NioSocketChannel


```
ECTOR_PROVIDER = SelectorProvider.provider();

) {
```

torProvider 实现类。
实现类，也会对应一个 ChannelConfig 实现类。例如，
。
rovider) 方法，创建 NIO 的 ServerSocketChannel 对象。

```
rProvider provider) {

 open {@link SocketChannel} and so remove condition
 which is called by each SocketChannel.open() other

         *  See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.
         */
        return provider.openSocketChannel();
    } catch (IOException e) {
        throw new ChannelException("Failed to open a socket.", e);
    }
}
```

- 😈 是不是很熟悉这样的代码，效果和 `SocketChannel#open()` 方法创建 SocketChannel 对象是一致。

- `#NioSocketChannel(SocketChannel channel)` 构造方法，代码如下：

```
public NioSocketChannel(SocketChannel socket) {
    this(null, socket);
}

public NioSocketChannel(Channel parent, SocketChannel socket) {
    super(parent, socket);
    config = new NioSocketChannelConfig(this, socket.socket());
}
```

文章目录

，见「3.7.1.2 AbstractNioByteChannel」。

对象。

```
SelectableChannel ch) {
```

，和 NioServerSocketChannel 是一致的。

代码如下：

```
    final Map<ChannelOption<?>, Object> options = options0();
    synchronized (options) {
        setChannelOptions(channel, options, logger);
    }

    // 初始化 Channel 的属性集合
    final Map<AttributeKey<?>, Object> attrs = attrs0();
    synchronized (attrs) {
        for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
            channel.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
        }
    }
}
```

- 比较简单，已经添加中文注释，胖友自己查看。

666. 彩蛋

撸完 Netty 服务端启动之后，再撸 Netty 客户端启动之后，出奇的顺手。美滋滋。

另外，也推荐如下和 Netty 客户端启动相关的文章，以加深理解：

- 杨武兵 《Netty 源码分析系列 —— Bootstrap》
- 永顺 《Netty 源码分析之 一 揭开 Bootstrap 神秘的红盖头 (客户端)》

文章目录