- NETTY

精尽 Netty 源码解析 —— Channel（二）之 accept 操作

1. 概述

文章目录

cept)客户端连接的过程。简单来说：

询是否有新的客户端连接接入。
nnel 为 Netty NioSocketChannel 对象。
oop ， 将客户端的 NioSocketChannel 注册到其上。并且，
户端是否有数据写入。

> 老艿艿：有点不知道怎么取标题好，直接用方法名吧。

在 NioEventLoop 的 `#processSelectedKey(SelectionKey k, AbstractNioChannel ch)` 方法中，我们会看到这样一段代码：

```
// SelectionKey.OP_READ 或 SelectionKey.OP_ACCEPT 就绪
// readyOps == 0 是对 JDK Bug 的处理，防止空的死循环
// Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead
// to a spin loop
if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
    unsafe.read();
}
```

- 当 `(readyOps & SelectionKey.OP_ACCEPT) != 0` 时，这就是服务端 NioServerSocketChannel 的 boss EventLoop 线程**轮询到**有新的客户端连接接入。
- 然后，调用 `NioMessageUnsafe#read()` 方法，“读取”(😈 这个抽象很灵性)新的客户端连接连入。

`NioMessageUnsafe#read()` 方法，代码如下：

```
1: private final class NioMessageUnsafe extends AbstractNioUnsafe {
2:
3:     /**
```

```
4:      * 新读取的客户端连接数组
5:      */
6:     private final List<Object> readBuf = new ArrayList<Object>();
7:
8:     @SuppressWarnings("Duplicates")
9:     @Override
10:    public void read() {
11:        assert eventLoop().inEventLoop();
12:        final ChannelConfig config = config();
13:        final ChannelPipeline pipeline = pipeline();
14:        // 获得 RecvByteBufAllocator.Handle 对象
15:        final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();
16:        // 重置 RecvByteBufAllocator.Handle 对象
17:        allocHandle.reset(config);
18:
19:        boolean closed = false;
```

文章目录


```
                    uf 中
                    ages(readBuf);
                    结束
32:                     closed = true; // 标记关闭
33:                     break;
34:                 }
35:
36:                 // 读取消息数量 + localRead
37:                 allocHandle.incMessagesRead(localRead);
38:             } while (allocHandle.continueReading()); // 循环判断是否继续读取
39:         } catch (Throwable t) {
40:             // 记录异常
41:             exception = t;
42:         }
43:
44:         // 循环 readBuf 数组，触发 Channel read 事件到 pipeline 中。
45:         int size = readBuf.size();
46:         for (int i = 0; i < size; i ++) {
47:             // TODO 芋艿
48:             readPending = false;
49:             // 在内部，会通过 ServerBootstrapAcceptor ，将客户端的 Netty NioSocketChannel 注册到
50:             pipeline.fireChannelRead(readBuf.get(i));
51:         }
52:         // 清空 readBuf 数组
53:         readBuf.clear();
54:         // 读取完成
55:         allocHandle.readComplete();
56:         // 触发 Channel readComplete 事件到 pipeline 中。
57:         pipeline.fireChannelReadComplete();
```

```
58:
59:             // 发生异常
60:             if (exception != null) {
61:                 // 判断是否要关闭 TODO 芋艿
62:                 closed = closeOnReadError(exception);
63:
64:                 // 触发 exceptionCaught 事件到 pipeline 中。
65:                 pipeline.fireExceptionCaught(exception);
66:             }
67:
68:             if (closed) {
69:                 // TODO 芋艿
70:                 inputShutdown = true;
71:                 // TODO 芋艿
72:                 if (isOpen()) {
73:                     close(voidPromise());
```

文章目录


```
 which was not processed yet.

) or ChannelHandlerContext.read() in channelRead(...
) or ChannelHandlerContext.read() in channelReadComp

tty/issues/2254

Read()) {
86:             }
87:         }
88:     }
89: }
```

- 😈 NioMessageUnsafe 只有一个 `#read()` 方法，而该方法，“读取”新的客户端连接连入。
- 第 15 行：调用 `Unsafe#recvBufAllocHandle()` 方法，获得 获得 RecvByteBufAllocator.Handle 对象。默认情况下，返回的是 AdaptiveRecvByteBufAllocator.HandleImpl 对象。关于它的内容，我们放在 ByteBuf 相关的文章，详细解析。
  - 第 17 行：调用 `DefaultMaxMessagesRecvByteBufAllocator.MaxMessageHandle#reset(ChannelConfig)` 方法，重置 RecvByteBufAllocator.Handle 对象。代码如下：

```
@Override
public void reset(ChannelConfig config) {
    this.config = config; // 重置 ChannelConfig 对象
    maxMessagePerRead = maxMessagesPerRead(); // 重置 maxMessagePerRead 属性
    totalMessages = totalBytesRead = 0; // 重置 totalMessages 和 totalBytesRead 属性
}
```

    - 注意，AdaptiveRecvByteBufAllocator.HandleImpl 继承 DefaultMaxMessagesRecvByteBufAllocator.MaxMessageHandle 抽象类。
- 第 22 至 42 行：**while 循环** “读取”新的客户端连接连入。

- 第 25 行：调用 `NioServerSocketChannel#doReadMessages(List<Object> buf)` 方法，读取客户端的连接到 `readBuf` 中。详细解析，胖友先跳到 「3. AbstractNioMessageChannel#doReadMessages」 中，看完记得回到此处。
- 第 25 至 29 行：无可读取的客户端的连接，结束循环。
- 第 30 至 34 行：读取出错，**标记关闭服务端**，并结束循环。目前我们看到 `NioServerSocketChannel#doReadMessages(List<Object> buf)` 方法的实现，返回的结果只会存在 0 和 1 ，也就是说不会出现这种情况。笔者又去翻了别的实现类，例如 `NioDatagramChannel#doReadMessages(List<Object> buf)` 方法，在发生异常时，会返回 -1 。
- 第 37 行：调用 `AdaptiveRecvByteBufAllocator.HandleImpl#incMessagesRead(int amt)` 方法，读取消息(客户端)数量 + `localRead` 。代码如下：

```
@Override
public final void incMessagesRead(int amt) {
    totalMessages += amt;
}
```


考虑到**抽象**的需要，所以统一使用“消息”的说法。

dleImpl#incMessagesRead(int

续，读取(接受)新的客户端连接。代码如下：

va

oplier);

MessageHandle.java

```
@Override
public boolean continueReading(UncheckedBooleanSupplier maybeMoreDataSupplier) {
    return config.isAutoRead() &&
           (!respectMaybeMoreData || maybeMoreDataSupplier.get()) &&
           totalMessages < maxMessagePerRead &&
           totalBytesRead > 0; // <1>
}
```

  - 因为 `<1>` 处，此时 `totalBytesRead` 等于 0 ，所以会返回 **false** 。因此，循环会结束。也因此，对于 NioServerSocketChannel 来说，**每次只接受一个新的客户端连接。**😈 当然，因为服务端 NioServerSocketChannel 对 `Selectionkey.OP_ACCEPT` 事件感兴趣，所以**后续的新的客户端连接还是会被接受的。**
- 第 39 至 42 行：读取过程中发生异常，记录该异常到 `exception` 中，同时结束循环。
- 第 44 至 51 行：循环 `readBuf` 数组，触发 Channel read 事件到 pipeline 中。
  - 第 48 行：TODO 芋艿 细节
  - 第 50 行：调用 `ChannelPipeline#fireChannelRead(Object msg)` 方法，触发 Channel read 事件到 pipeline 中。
    - **注意**，传入的方法参数是新接受的客户端 NioSocketChannel 连接。
    - 在内部，会通过 ServerBootstrapAcceptor ，将客户端的 Netty NioSocketChannel 注册到 EventLoop 上。详细解析，胖友先跳到 「4. ServerBootstrapAcceptor」 中，看完记得回到此处。
- 第 53 行：清空 `readBuf` 数组。
- 第 55 行：调用 `RecvByteBufAllocator.Handle#readComplete()` 方法，读取完成。暂无重要的逻辑，不详细解析。
- 第 57 行：调用 `ChannelPipeline#fireChannelReadComplete()` 方法，触发 Channel readComplete 事件到 pipeline 中。
  - *如果有需要，胖友可以自定义处理器，处理该事件。一般情况下，不需要。*

- 如果没有自定义 ChannelHandler 进行处理，最终会被 pipeline 中的尾节点 TailContext 所处理。代码如下：

```
// TailContext.java
@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
    onUnhandledInboundChannelReadComplete();
}

// DefaultChannelPipeline.java
protected void onUnhandledInboundChannelReadComplete() {
}
```

  - 具体的调用是**空方法**。
- 第 60 至 66 行： `exception` 非空，说明在接受连接过程中发生异常。
  - 第 62 行：TODO 芋艿 细节
  - 第 65 行： 调用 `ChannelPipeline#fireExceptionCaught(Throwable)` 方法，触发 exceptionCaught 事件到

事件。详细解析，见「4.3 exceptionCaught」。

一般情况下，不需要。

l#doReadMessages

端的连接到方法参数 `buf` 中。它是一个**抽象**方法，定义在

```
/**
 * Read messages into the given array and return the amount which was read.
 */
protected abstract int doReadMessages(List<Object> buf) throws Exception;
```

- 返回值为读取到的数量。

NioServerSocketChannel 对该方法的实现代码如下：

```
 1: @Override
 2: protected int doReadMessages(List<Object> buf) throws Exception {
 3:     // 接受客户端连接
 4:     SocketChannel ch = SocketUtils.accept(javaChannel());
 5:
 6:     try {
 7:         // 创建 Netty NioSocketChannel 对象
 8:         if (ch != null) {
 9:             buf.add(new NioSocketChannel(this, ch));
10:             return 1;
11:         }
12:     } catch (Throwable t) {
13:         logger.warn("Failed to create a new channel from an accepted socket.", t);
14:         // 发生异常，关闭客户端的 SocketChannel 连接
15:         try {
16:             ch.close();
```

```
 17:         } catch (Throwable t2) {
 18:             logger.warn("Failed to close a socket.", t2);
 19:         }
 20:     }
 21:
 22:     return 0;
 23: }

 @Override
protected ServerSocketChannel javaChannel() {
    return (ServerSocketChannel) super.javaChannel();
}
```

- 第 4 行：调用 `SocketUtils#accept(ServerSocketChannel serverSocketChannel)` 方法，接受客户端连接。代码如下：


```
ocketChannel serverSocketChannel) throws IOExcepti

 PrivilegedExceptionAction<SocketChannel>() {

Exception {
t(); // <1>

    }
}
```

  - 重点是看 `<1>` 处，调用 `ServerSocketChannel#accept()` 方法，接受客户端连接。
- 第 9 行：基于客户端的 NIO ServerSocket ，创建 Netty NioSocketChannel 对象。整个过程，就是《精尽 Netty 源码分析 —— 启动（二）之客户端》的「3.7.1 创建 Channel 对象」小节。
  - 第 10 行：返回 1 ，表示成功接受了 1 个新的客户端连接。
- 第 12 至 20 行：发生异常，关闭客户端的 SocketChannel 连接，并打印**告警**日志。
  - 第 22 行：返回 0 ，表示成功接受 0 个新的客户端连接。

4. ServerBootstrapAcceptor

ServerBootstrapAcceptor ，继承 ChannelInboundHandlerAdapter 类，服务器接收器(acceptor)，负责将接受的客户端的 NioSocketChannel 注册到 EventLoop 中。

另外，从继承的是 ChannelInboundHandlerAdapter 类，可以看出它是 Inbound 事件处理器。

4.1 构造方法

在服务端的启动过程中，我们看到 ServerBootstrapAcceptor 注册到服务端的 NioServerSocketChannel 的 pipeline 的尾部，代码如下：

```
// 记录当前的属性
final EventLoopGroup currentChildGroup = childGroup;
final ChannelHandler currentChildHandler = childHandler;
```

```
final Entry<ChannelOption<?>, Object>[] currentChildOptions;
final Entry<AttributeKey<?>, Object>[] currentChildAttrs;
synchronized (childOptions) {
    currentChildOptions = childOptions.entrySet().toArray(newOptionArray(0));
}
synchronized (childAttrs) {
    currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(0));
}

// 添加 ChannelInitializer 对象到 pipeline 中，用于后续初始化 ChannelHandler 到 pipeline 中。
p.addLast(new ChannelInitializer<Channel>() {

    @Override
    public void initChannel(final Channel ch) throws Exception {
        final ChannelPipeline pipeline = ch.pipeline();

文章目录

1. 概述
2. NioMessageUnsafe#read
3. AbstractNioMessageChannel#doReadMessages
4. ServerBootstrapAcceptor
   4.1 构造方法
   4.2 channelRead
   4.3 exceptionCaught
666. 彩蛋

                                                                    。

                                                                    中。
                                                                    ithub.com/lightningMan/netty/commit/4638df20628a8987

                                                                    apAcceptor(
                        ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildA
            }
        });
    }

});
```

- 即 `<1>` 处。也是在此处，创建了 ServerBootstrapAcceptor 对象。代码如下：

```
private final EventLoopGroup childGroup;
private final ChannelHandler childHandler;
private final Entry<ChannelOption<?>, Object>[] childOptions;
private final Entry<AttributeKey<?>, Object>[] childAttrs;
/**
 * 自动恢复接受客户端连接的任务
 */
private final Runnable enableAutoReadTask;

ServerBootstrapAcceptor(
        final Channel channel, EventLoopGroup childGroup, ChannelHandler childHandler,
        Entry<ChannelOption<?>, Object>[] childOptions, Entry<AttributeKey<?>, Object>[] childAttr
    this.childGroup = childGroup;
    this.childHandler = childHandler;
    this.childOptions = childOptions;
    this.childAttrs = childAttrs;
```

```
    // Task which is scheduled to re-enable auto-read.
    // It's important to create this Runnable before we try to submit it as otherwise the URLClass
    // not be able to load the class because of the file limit it already reached.
    //
    // See https://github.com/netty/netty/issues/1328
    enableAutoReadTask = new Runnable() { // <2>
        @Override
        public void run() {
            channel.config().setAutoRead(true);
        }
    };
}
```

- `enableAutoReadTask` 属性，自动恢复接受客户端连接的任务，在 `<2>` 处初始化。具体的使用，我们在「4.3 exceptionCaught」中，详细解析。

文章目录

g) 方法，将接受的客户端的 NioSocketChannel 注册到

```
ctx, Object msg) {
端的 NioSocketChannel

6:      final Channel child = (Channel) msg;
7:      // 添加 NioSocketChannel 的处理器
8:      child.pipeline().addLast(childHandler);
9:      // 设置 NioSocketChannel 的配置项
10:     setChannelOptions(child, childOptions, logger);
11:     // 设置 NioSocketChannel 的属性
12:     for (Entry<AttributeKey<?>, Object> e: childAttrs) {
13:         child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
14:     }
15:
16:     try {
17:         // 注册客户端的 NioSocketChannel 到 work EventLoop 中。
18:         childGroup.register(child).addListener(new ChannelFutureListener() {
19:
20:             @Override
21:             public void operationComplete(ChannelFuture future) throws Exception {
22:                 // 注册失败，关闭客户端的 NioSocketChannel
23:                 if (!future.isSuccess()) {
24:                     forceClose(child, future.cause());
25:                 }
26:             }
27:
28:         });
29:     } catch (Throwable t) {
30:         // 发生异常，强制关闭客户端的 NioSocketChannel
31:         forceClose(child, t);
```

```
32:     }
33: }
```

- 为了方便描述，我们统一认为接受的客户端连接为 NioSocketChannel 对象。
- 第 6 行：接受的客户端的 NioSocketChannel 对象。
  - 第 8 行：调用 `ChannelPipeline#addLast(childHandler)` 方法，将配置的子 Channel 的处理器，添加到 NioSocketChannel 中。
  - 第 10 至 14 行：设置 NioSocketChannel 的配置项、属性。
- 第 17 至 28 行：调用 `EventLoopGroup#register(Channel channel)` 方法，将客户端的 NioSocketChannel 对象，从 worker EventLoopGroup 中选择一个 EventLoop ，注册到其上。
  - 后续的逻辑，就和《精尽 Netty 源码分析 —— 启动（一）之服务端》的注册逻辑**基本一致**(虽然说，文章写的是 NioServerSocketChannel 的注册逻辑)。
  - 在注册完成之后，该 worker EventLoop 就会开始轮询该客户端是否有数据写入。

`forceClose(Channel child, Throwable t)` 方法，

```
Throwable t) {

ed channel: {}", child, t);
```

方法，强制关闭客户端的 NioSocketChannel 。

`nel child, Throwable t)` 方法，强制关闭客户端的

4.3 exceptionCaught

`#exceptionCaught(ChannelHandlerContext ctx, Throwable cause)` 方法，当捕获到异常时，**暂停 1 秒**，不再接受新的客户端连接；而后，再恢复接受新的客户端连接。代码如下：

```
 1: @Override
 2: public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
 3:     final ChannelConfig config = ctx.channel().config();
 4:     if (config.isAutoRead()) {
 5:         // 关闭接受新的客户端连接
 6:         // stop accept new connections for 1 second to allow the channel to recover
 7:         // See https://github.com/netty/netty/issues/1328
 8:         config.setAutoRead(false);
 9:         // 发起 1 秒的延迟任务，恢复重启开启接受新的客户端连接
10:         ctx.channel().eventLoop().schedule(enableAutoReadTask, 1, TimeUnit.SECONDS);
11:     }
12:
13:     // 继续传播 exceptionCaught 给下一个节点
14:     // still let the exceptionCaught event flow through the pipeline to give the user
15:     // a chance to do something with it
16:     ctx.fireExceptionCaught(cause);
17: }
```

- 第 8 行：调用 `ChannelConfig#setAutoRead(false)` 方法，关闭接受新的客户端连接。代码如下：

```
// DefaultChannelConfig.java
/**
 * {@link #autoRead} 的原子更新器
 */
private static final AtomicIntegerFieldUpdater<DefaultChannelConfig> AUTOREAD_UPDATER = AtomicInte
/**
 * 是否开启自动读取的开关
 *
 * 1 - 开启
 * 0 - 关闭
 */
@SuppressWarnings("FieldMayBeFinal")
private volatile int autoRead = 1;

@Override
public ChannelConfig setAutoRead(boolean autoRead) {
    boolean oldAutoRead = AUTOREAD_UPDATER.getAndSet(this, autoRead ? 1 : 0) == 1;
```

文章目录

- `autoRead` 字段，是否开启自动读取的开关。😈 笔者原本以为是个 `boolean` 类型，是不是胖友也是。其中，1 表示开启，0 表示关闭。
  - `AUTOREAD_UPDATER` 静态变量，对 `autoRead` 字段的原子更新器。
- `<1>` 处，使用 `AUTOREAD_UPDATER` 更新 `autoRead` 字段，并获得更新前的值。为什么需要获取更新前的值呢？在后续的 `<2.1>` 和 `<2.2>` 中，当 `autoRead` 有变化时候，才进行后续的逻辑。
- 😈 下面的逻辑，我们按照 `channel` 的类型为 NioServerSocketChannel 来分享。
- `<2.1>` 处， `autoRead && !oldAutoRead` 返回 `true` ，意味着恢复重启开启接受新的客户端连接。所以调用 `NioServerSocketChannel#read()` 方法，后续的逻辑，就是《精尽 Netty 源码分析 —— 启动（一）之服务端》的「3.13.3 beginRead」的逻辑。
- `<2.2>` 处， `!autoRead && oldAutoRead` 返回 `false` ，意味着关闭接受新的客户端连接。所以调用 `#autoReadCleared()` 方法，移除对 `SelectionKey.OP_ACCEPT` 事件的感兴趣。

```
// NioServerSocketChannel.java

@Override
protected void autoReadCleared() {
    clearReadPending();
}
```

- 在方法内部，会调用 `#clearReadPending()` 方法，代码如下：

```
protected final void clearReadPending() {
    if (isRegistered()) {
        EventLoop eventLoop = eventLoop();
```

```
        if (eventLoop.inEventLoop()) {
            clearReadPending0();
        } else {
            eventLoop.execute(clearReadPendingRunnable);
        }
    } else {
        // Best effort if we are not registered yet clear readPending. This happens durir
        // NB: We only set the boolean field instead of calling clearReadPending0(), beca
        // not set yet so it would produce an assertion failure.
        readPending = false;
    }
}

private final Runnable clearReadPendingRunnable = new Runnable() {
    @Override
    public void run() {
```

文章目录


```
adOp();
```

- 最终的结果，是在 EventLoop 的线程中，调用 `AbstractNioUnsafe#clearReadPending0()` 方法，移除对“**读**”事件的感兴趣(对于 NioServerSocketChannel 的 “**读**“事件就是 `SelectionKey.OP_ACCEPT`)。代码如下：

```
// AbstractNioUnsafe.java

protected final void removeReadOp() {
    SelectionKey key = selectionKey();
    // 忽略，如果 SelectionKey 不合法，例如已经取消
    // Check first if the key is still valid as it may be canceled as part of the de
    // from the EventLoop
    // See https://github.com/netty/netty/issues/2104
    if (!key.isValid()) {
        return;
    }
    // 移除对“读”事件的感兴趣。
    int interestOps = key.interestOps();
    if ((interestOps & readInterestOp) != 0) {
        // only remove readInterestOp if needed
        key.interestOps(interestOps & ~readInterestOp);
    }
}
```

- 通过取反求并，后调用 `SelectionKey#interestOps(interestOps)` 方法，**仅**移除对“读”事件的感兴趣。
- 😈 整个过程的调用链，有丢丢长，胖友可以回看，或者多多调试。

- 第 10 行：调用 `EventLoop#schedule(Runnable command, long delay, TimeUnit unit)` 方法，发起 1 秒的延迟任务，恢复重启开启接受新的客户端连接。该定时任务会调用 `ChannelConfig#setAutoRead(true)` 方法，即对应 `<2.1>` 情况。
- 第 16 行：调用 `ChannelHandlerContext#fireExceptionCaught(cause)` 方法，继续传播 exceptionCaught 给下一个节点。具体的原因，可看英文注释。

666. 彩蛋

推荐阅读文章：

- 闪电侠 《netty 源码分析之新连接接入全解析》
- 占小狼 《Netty 源码分析之 accept 过程》

文章目录