- NETTY

精尽 Netty 源码解析 —— Channel（一）之简介

1. 概述

在前面的文章中，我们已经不断看到 Netty Channel 的身影，例如：

- 在《精尽 Netty 源码分析 —— 启动（一）之服务端》中，我们看了服务端 NioServerSocketChannel **对象创建**的过程。
- 在《精尽 Netty 源码分析 —— 启动（二）之客户端》中，我们看了客户端 NioSocketChannel **对象创建**的过程。

但是，考虑到本小节的后续文章，我们还是需要这样一篇文章，整体性的再看一次 Channel 的面貌。

2. Channel

`io.netty.channel.Channel` ，实现 AttributeMap、ChannelOutboundInvoker、Comparable 接口，Netty Channel 接口。

在《精尽 Netty 源码分析 —— Netty 简介（一）之项目结构》中，我们对 Channel 的组件定义如下：

> Channel 是 Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、
> ite 之外，还包括了 Netty 框架相关的一些功能，如获取该
> oop 。
>
> 中，作为核心类的 Socket ，它对程序员来说并不是那么友
> 还是稍微高了点。而 Netty 的 Channel 则提供的一系列的
> 直接与 Socket 进行操作的复杂性。而相对于原生 NIO 的
> Channel 具有如下优势(摘自《Netty权威指南(第二版)》)
>
> 层，采用 Facade 模式进行统一封装，将网络 I/O 操作、网
> 其他操作封装起来，统一对外提供。
> 义尽量大而全，为 SocketChannel 和
> nnel 提供统一的视图，由不同子类实现不同的功能，公共
> 实现，最大程度地实现功能和接口的重用。

文章目录

> - 具体实现采用聚合而非包含的方式，将相关的功能类聚合在 Channel 中，由 Channel 统一负责和调度，功能实现更加灵活。

2.1 基础查询

```
/**
 * Returns the globally unique identifier of this {@link Channel}.
 *
 * Channel 的编号
 */
ChannelId id();

/**
 * Return the {@link EventLoop} this {@link Channel} was registered to.
 *
 * Channel 注册到的 EventLoop
 */
EventLoop eventLoop();

/**
 * Returns the parent of this channel.
 *
 * 父 Channel 对象
 *
 * @return the parent channel.
 *         {@code null} if this channel does not have a parent channel.
 */
Channel parent();

/**
n of this channel.

use-only</em> object that provides unsafe operations.

nk ChannelPipeline}.

于处理 Inbound 和 Outbound 事件的处理

/**
```

文章目录

```
     * Return the assigned {@link ByteBufAllocator} which will be used to allocate {@link ByteBuf}s.
     *
     * ByteBuf 分配器
     */
    ByteBufAllocator alloc();

    /**
     * Returns the local address where this channel is bound to.  The returned
     * {@link SocketAddress} is supposed to be down-cast into more concrete
     * type such as {@link InetSocketAddress} to retrieve the detailed
     * information.
     *
     * 本地地址
     *
     * @return the local address of this channel.
     *         {@code null} if this channel is not bound.
     */
    SocketAddress localAddress();
    /**
     * Returns the remote address where this channel is connected to.  The
     * returned {@link SocketAddress} is supposed to be down-cast into more
     * concrete type such as {@link InetSocketAddress} to retrieve the detailed
     * information.
     *
     * 远端地址
     *
     * @return the remote address of this channel.
     *         {@code null} if this channel is not connected.
     *         If this channel is not connected but it can receive messages
     *         from arbitrary remote addresses (e.g. {@link DatagramChannel},
     *         use {@link DatagramPacket#recipient()} to determine
     *         the origination of the received message as this method will
     *         return {@code null}.
     */
                                                                    ();
```

文章目录


```
                                                                    arent() 、 #config() 、 #localAddress() 、 #remoteAddress() 方法。
                                                                    #eventLoop() 、 #unsafe() 、 #pipeline() 、 #alloc() 方法。
```

```
                                                                    the {@link Channel} is open and may get active later
```

```
                                                                    , 不可用
```

```
    /**
```

```
     * Returns {@code true} if the {@link Channel} is registered with an {@link EventLoop}.
     *
     * Channel 是否注册
     *
     * true 表示 Channel 已注册到 EventLoop 上
     * false 表示 Channel 未注册到 EventLoop 上
     */
    boolean isRegistered();

    /**
     * Return {@code true} if the {@link Channel} is active and so connected.
     *
     * Channel 是否激活
     *
     * 对于服务端 ServerSocketChannel ，true 表示 Channel 已经绑定到端口上，可提供服务
     * 对于客户端 SocketChannel ，true 表示 Channel 连接到远程服务器
     */
    boolean isActive();

    /**
     * Returns {@code true} if and only if the I/O thread will perform the
     * requested write operation immediately.  Any write requests made when
     * this method returns {@code false} are queued until the I/O thread is
     * ready to process the queued write requests.
     *
     * Channel 是否可写
     *
     * 当 Channel 的写缓存区 outbound 非 null 且可写时，返回 true
     */
    boolean isWritable();
    /**
     * 获得距离不可写还有多少字节数
     *
     * Get how many bytes can be written until {@link #isWritable()} returns {@code false}.
     s be non-negative. If {@link #isWritable()} is {@code false} then 0.

    );


    e drained from underlying buffers until {@link #isWritable()} returns {@co
     s be non-negative. If {@link #isWritable()} is {@code true} then 0.
```

文章目录

有**两**种情况：

nel 、或者客户端发起连接(connect)的 Channel 。

```
D -> ACTIVE -> CLOSE -> INACTIVE -> UNREGISTERED
```

Channel 。

```
REGISTERED -> ACTIVE -> CLOSE -> INACTIVE -> UNREGISTERED
```

一个**异常关闭**的 Channel 状态转移不符合上面的。

2.3 IO 操作

```
@Override
Channel read();

@Override
Channel flush();
```

- 这两个方法，继承自 ChannelOutboundInvoker 接口。实际还有如下几个：

```
ChannelFuture bind(SocketAddress localAddress);
ChannelFuture connect(SocketAddress remoteAddress);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress);
ChannelFuture disconnect();
ChannelFuture close();
ChannelFuture deregister();
ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise);
ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise);
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise prom
ChannelFuture disconnect(ChannelPromise promise);
ChannelFuture close(ChannelPromise promise);
ChannelFuture deregister(ChannelPromise promise);
ChannelOutboundInvoker read();
ChannelFuture write(Object msg);
ChannelFuture write(Object msg, ChannelPromise promise);
ChannelOutboundInvoker flush();
ChannelFuture writeAndFlush(Object msg, ChannelPromise promise);
sh(Object msg);
```

文章目录

重写 ChannelOutboundInvoker 这两个接口的原因是：将返回值从
成 Channel 。

lush() 方法，其它方法的返回值的类型都是 ChannelFuture ，这表明这些操作是**异步**

```
elFuture} which will be notified when this
method always returns the same future instance.
```

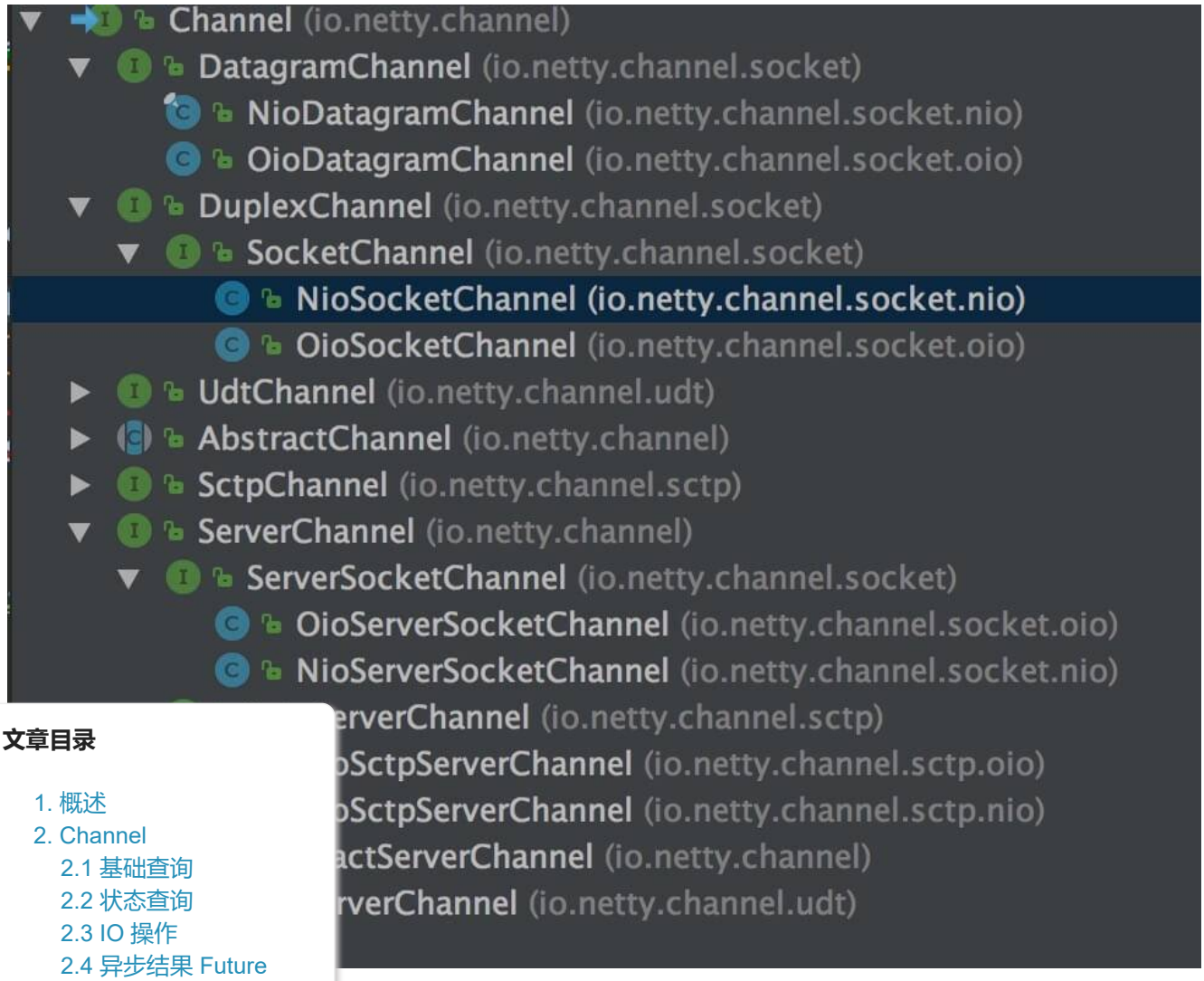
- 除了自定义的 `#closeFuture()` 方法，也从 ChannelOutboundInvoker 接口继承了几个接口方法：

```
ChannelPromise newPromise();
ChannelProgressivePromise newProgressivePromise();

ChannelFuture newSucceededFuture();
ChannelFuture newFailedFuture(Throwable cause);

ChannelPromise voidPromise();
```

- 通过这些接口方法，可创建或获得和该 Channel 相关的 Future / Promise 对象。

2.5 类图

Channel 的子接口和实现类如下图：

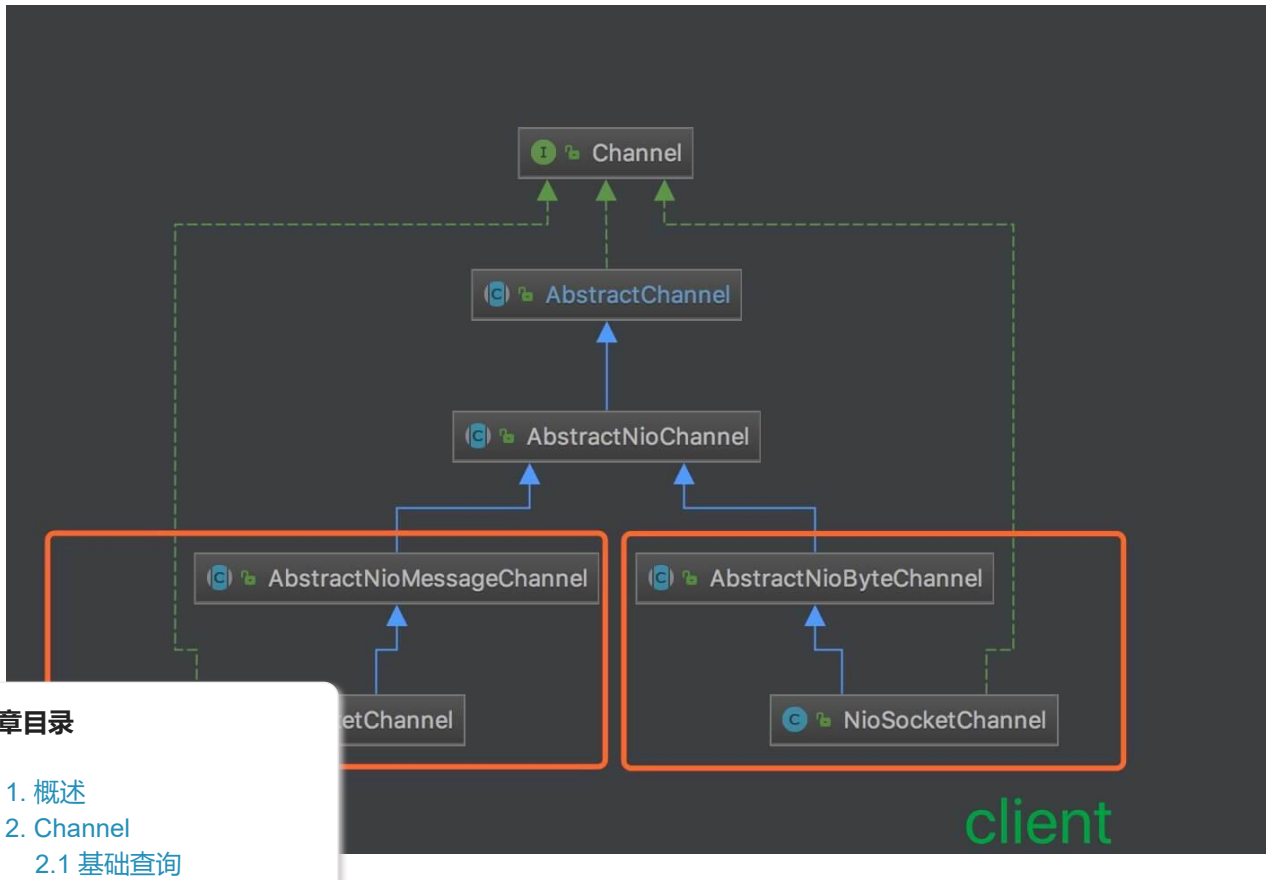


Embedded 四种 Channel 实现类。说明如下：

文章目录

Name	Package	Description
NIO	`io.netty.channel.socket.nio`	Uses the `java.nio.channels` package as a foundation and so uses a selector-based approach.
OIO	`io.netty.channel.socket.oio`	Uses the `java.net package` as a foundation and so uses blocking streams.
Local	`io.netty.channel.local`	A local transport that can be used to communicate in the VM via pipes.
Embedded	`io.netty.channel.embedded`	Embedded transport, which allows using `ChannelHandlers` without a real network based Transport. This can be quite useful for testing your `ChannelHandler` implementations.

- 本系列仅分享 NIO 部分，所以裁剪类图如下：



文章目录

`y.channel.Channel` 内部，和 Channel 的操作**紧密结合**，下文我们将看到。

告诉我们，**无需且不必要**在我们使用 Netty 的代码中，**不能直接**调用 Unsafe 相关的方法。

```
ns that should <em>never</em> be called from user-code.

rovided to implement the actual transport, and must be invoked from an I/O

*    <li>{@link #localAddress()}</li>
```

```
 *   <li>{@link #remoteAddress()}</li>
 *   <li>{@link #closeForcibly()}</li>
 *   <li>{@link #register(EventLoop, ChannelPromise)}</li>
 *   <li>{@link #deregister(ChannelPromise)}</li>
 *   <li>{@link #voidPromise()}</li>
 * </ul>
 */
```

😈 当然，对于我们想要了解 Netty 内部实现的胖友，那必须开扒它的代码实现落。因为它和 Channel 密切相关，所以我们也对它的接口做下分类。

3.1 基础查询

```
/**
 * Return the assigned {@link RecvByteBufAllocator.Handle} which will be used to allocate {@link ByteB
 * receiving data.
 *
 * ByteBuf 分配器的处理器
 */
RecvByteBufAllocator.Handle recvBufAllocHandle();

/**
 * Return the {@link SocketAddress} to which is bound local or
 * {@code null} if none.
 *
 * 本地地址
 */
SocketAddress localAddress();

/**
 * Return the {@link SocketAddress} to which is bound remote or
 * {@code null} if none is bound yet.
 *
```



```
();
```

```
ntLoop, ChannelPromise promise);
alAddress, ChannelPromise promise);
emoteAddress, SocketAddress localAddress, ChannelPromise promise);
ise promise);
omise);

void deregister(ChannelPromise promise);
```

```
void beginRead();
void write(Object msg, ChannelPromise promise);
void flush();

/**
 * Returns the {@link ChannelOutboundBuffer} of the {@link Channel} where the pending write requests a
 */
ChannelOutboundBuffer outboundBuffer();
```

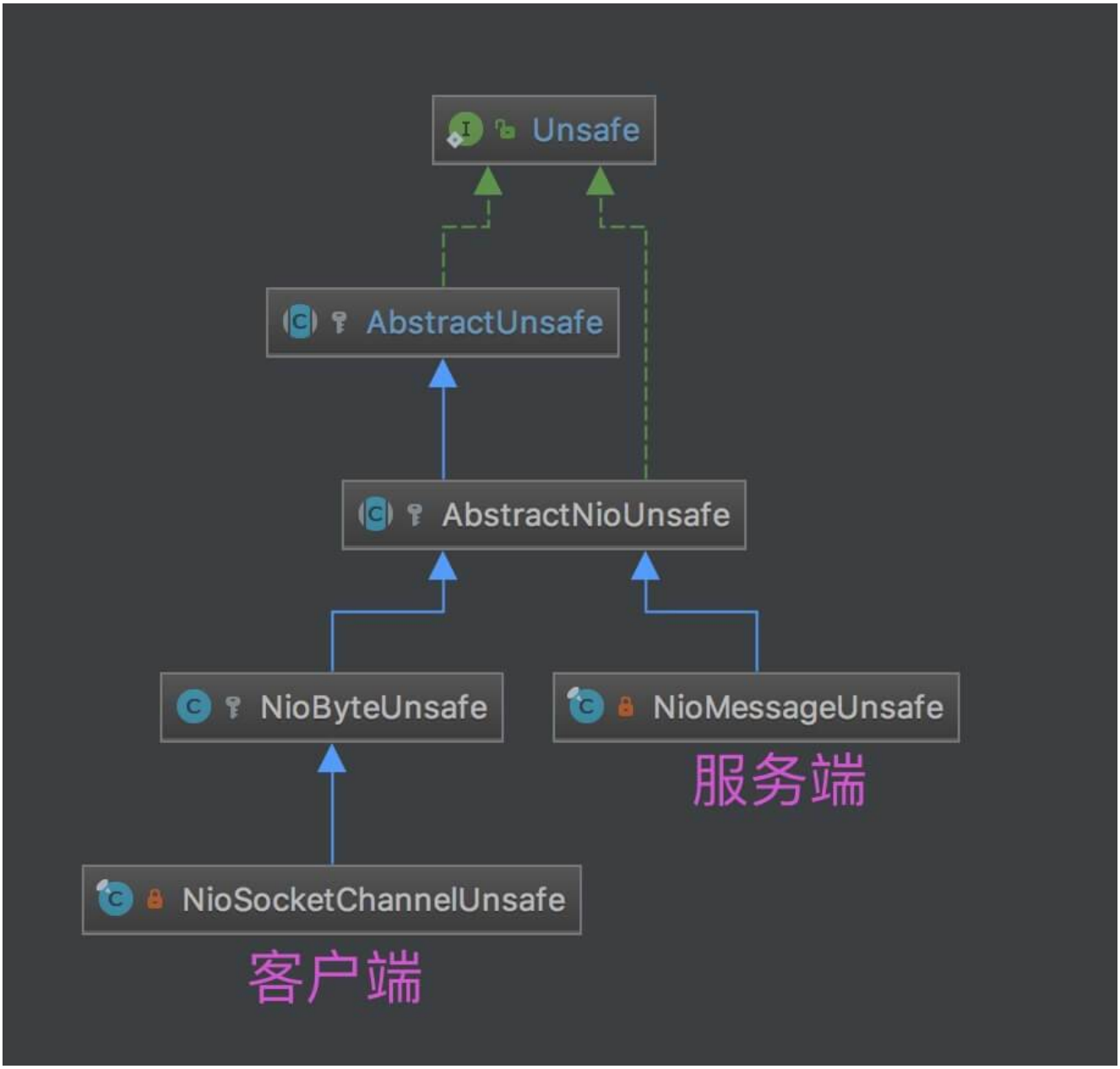
3.4 异步结果 Future

```
/**
 * Return a special ChannelPromise which can be reused and passed to the operations in {@link Unsafe}.
 * It will never be notified of a success or error and so is only a placeholder for operations
 * that take a {@link ChannelPromise} as argument but for which you not want to get notified.
 */
ChannelPromise voidPromise();
```

3.5 类图

Unsafe 的子接口和实现类如下图:

文章目录



nnel 相关的 Unsafe 的子接口和实现类部分。

Unsafe 来说，类名中包含 Byte 是属于客户端的，Message 是属于服务端的。

文章目录

实现 Serializable、Comparable 接口，Channel 编号接口。代码如下：

```
extends Serializable, Comparable<ChannelId> {

 globally non-unique string representation of the {@link ChannelId}.

 globally unique string representation of the {@link ChannelId}.
     *
```

```
     * 全局唯一
     */
    String asLongText();

}
```

- #asShortText() 方法，返回的编号，短，但是全局非唯一。
- #asLongText() 方法，返回的编号，长，但是全局唯一。

ChanelId 的**默认**实现类为 io.netty.channel.DefaultChannelId ，我们主要看看它是如何生成 Channel 的**两种**编号的。代码如下：

```
@Override
public String asShortText() {
    String shortValue = this.shortValue;
    if (shortValue == null) {
        this.shortValue = shortValue = ByteBufUtil.hexDump(data, data.length - RANDOM_LEN, RANDOM_LEN)
    }
    return shortValue;
}

@Override
public String asLongText() {
    String longValue = this.longValue;
    if (longValue == null) {
        this.longValue = longValue = newLongValue();
    }
    return longValue;
}
```

- 对于 #asShortText() 方法，仅使用最后 4 字节的随机数字，并转换成 16 进制的数字字符串。也因此，短，但是全

文章目录

通过调用 #newLongValue() 方法生成。代码如下：

```
ue() {
ew StringBuilder(2 * data.length + 5); // + 5 的原因是有 5 个 '-'

d(buf, i, MACHINE_ID.length); // MAC 地址。
d(buf, i, PROCESS_ID_LEN); // 进程 ID 。4 字节。
d(buf, i, SEQUENCE_LEN); // 32 位数字，顺序增长。4 字节。
d(buf, i, TIMESTAMP_LEN); // 时间戳。8 字节。
d(buf, i, RANDOM_LEN); // 32 位数字，随机。4 字节。
th;
0, buf.length() - 1);


Field(StringBuilder buf, int i, int length) {
l.hexDump(data, i, length));
```

```
        return i;
    }
```

- 具体的生成规则，见代码。最终也是 16 进制的数字。也因此，长，但是全局唯一。

5. ChannelConfig

`io.netty.channel.ChannelConfig` ，Channel 配置接口。代码如下：

```
Map<ChannelOption<?>, Object> getOptions();
<T> T getOption(ChannelOption<T> option);
boolean setOptions(Map<ChannelOption<?>, ?> options);
<T> boolean setOption(ChannelOption<T> option, T value);

int getConnectTimeoutMillis();
ChannelConfig setConnectTimeoutMillis(int connectTimeoutMillis);

@Deprecated
int getMaxMessagesPerRead();
@Deprecated
ChannelConfig setMaxMessagesPerRead(int maxMessagesPerRead);

int getWriteSpinCount();
ChannelConfig setWriteSpinCount(int writeSpinCount);

ByteBufAllocator getAllocator();
ChannelConfig setAllocator(ByteBufAllocator allocator);

<T extends RecvByteBufAllocator> T getRecvByteBufAllocator();
ChannelConfig setRecvByteBufAllocator(RecvByteBufAllocator allocator);

boolean isAutoRead();
oolean autoRead);


oolean autoClose);

Mark();
HighWaterMark(int writeBufferHighWaterMark);

ark();
LowWaterMark(int writeBufferLowWaterMark);

sageSizeEstimator();
Estimator(MessageSizeEstimator estimator);

teBufferWaterMark();
WaterMark(WriteBufferWaterMark writeBufferWaterMark);
```

tion<T> option, T value) 方法时，会调用相应的 #setXXX(...) 方法。代码如

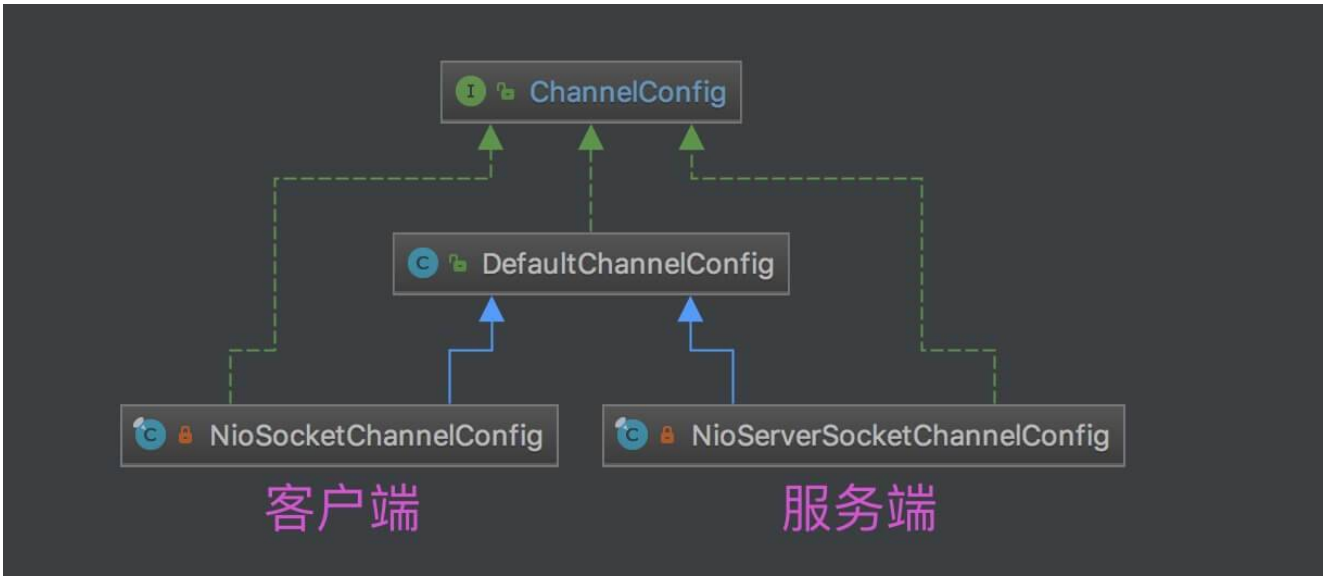
文章目录

```
// DefaultChannelConfig.java

@Override
@SuppressWarnings("deprecation")
public <T> boolean setOption(ChannelOption<T> option, T value) {
    validate(option, value);

    if (option == CONNECT_TIMEOUT_MILLIS) {
        setConnectTimeoutMillis((Integer) value);
    } else if (option == MAX_MESSAGES_PER_READ) {
        setMaxMessagesPerRead((Integer) value);
    } else if (option == WRITE_SPIN_COUNT) {
        setWriteSpinCount((Integer) value);
    } else if (option == ALLOCATOR) {
        setAllocator((ByteBufAllocator) value);
    } else if (option == RCVBUF_ALLOCATOR) {
        setRecvByteBufAllocator((RecvByteBufAllocator) value);
    } else if (option == AUTO_READ) {
        setAutoRead((Boolean) value);
    } else if (option == AUTO_CLOSE) {
        setAutoClose((Boolean) value);
    } else if (option == WRITE_BUFFER_HIGH_WATER_MARK) {
        setWriteBufferHighWaterMark((Integer) value);
    } else if (option == WRITE_BUFFER_LOW_WATER_MARK) {
        setWriteBufferLowWaterMark((Integer) value);
    } else if (option == WRITE_BUFFER_WATER_MARK) {
        setWriteBufferWaterMark((WriteBufferWaterMark) value);
    } else if (option == MESSAGE_SIZE_ESTIMATOR) {
        setMessageSizeEstimator((MessageSizeEstimator) value);
    } else if (option == SINGLE_EVENTEXECUTOR_PER_GROUP) {
        setPinEventExecutorPerGroup((Boolean) value);
    } else {
        return false;
```

文章目录

etty.channel.ChannelOption 很多，胖友可以看下《Netty：option 和 childOption
]配置项。

如下图：



- 已经经过裁剪，仅保留 NIO Channel 相关的 ChannelConfig 的子接口和实现类部分。

666. 彩蛋

正如文头所说，在前面的文章中，我们已经不断看到 Netty Channel 的身影，例如：

- 在《精尽 Netty 源码分析 —— 启动（一）之服务端》中，我们看了服务端 NioServerSocketChannel **bind** 的过程。
- 在《精尽 Netty 源码分析 —— 启动（二）之客户端》中，我们看了客户端 NioSocketChannel **connect** 的过程。

在后续的文章中，我们会分享 Netty NIO Channel 的其他操作，😈 一篇一个操作。

推荐阅读文章：

- Hypercube《自顶向下深入分析 Netty（六）–Channel总述》

文章目录