3

第 3 篇

Netty 核心篇

6

第 6 章

揭开 BootStrap 的神秘面纱

课程目标

1、深入了解 Netty 的运行机制。

2、掌握 NioEventLoop、Pipeline、ByteBuf 的核心原理。

3、掌握 Netty 常见的调优方案。

内容定位

1、希望深入了解 Netty 源码的人群。

2、未来可能参与中间件开发的人群。

6.1 客户端 BootStrap

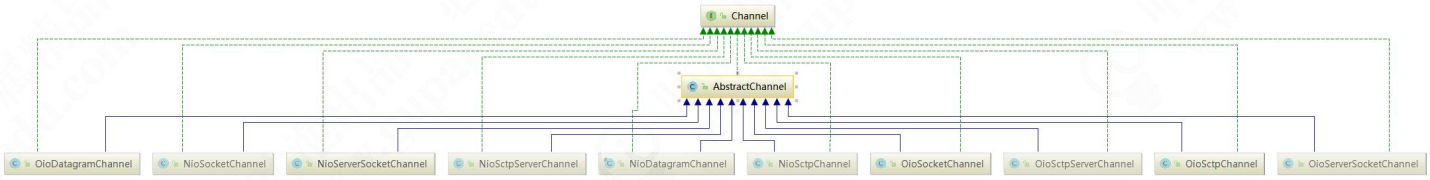
6.1.1 Channel 简介

在 Netty 中，Channel 是一个 Socket 的抽象，它为用户提供了关于 Socket 状态(是否是连接还是断开)以及对 Socket 的读写等操作。每当 Netty 建立了一个连接后，都创建一个对应的 Channel 实例。

除了 TCP 协议以外，Netty 还支持很多其他的连接协议，并且每种协议还有 NIO(非阻塞 IO)和 OIO(Old-IO，即传统的阻塞 IO)版本的区别。不同协议不同的阻塞类型的连接都有不同的 Channel 类型与之对应下面是一些常用的 Channel 类型：

类名	**解释**
NioSocketChannel	异步非阻塞的客户端 TCP Socket 连接。
NioServerSocketChannel	异步非阻塞的服务器端 TCP Socket 连接。
NioDatagramChannel	异步非阻塞的 UDP 连接。
NioSctpChannel	异步的客户端 Sctp（Stream Control Transmission Protocol，流控制传输协议）连接。
NioSctpServerChannel	异步的 Sctp 服务器端连接。
OioSocketChannel	同步阻塞的客户端 TCP Socket 连接。
OioServerSocketChannel	同步阻塞的服务器端 TCP Socket 连接。
OioDatagramChannel	同步阻塞的 UDP 连接。
OioSctpChannel	同步的 Sctp 服务器端连接。
OioSctpServerChannel	同步的客户端 TCP Socket 连接。

下面我们来看一下 Channel 的总体类图：



6.1.2 NioSocketChannel 的创建

Bootstrap 是 Netty 提供的一个便利的工厂类，我们可以通过它来完成 Netty 的客户端或服务器端的 Netty 初始化。

下面我先来看一个例子，从客户端和服务器端分别分析一下 Netty 的程序是如何启动的。首先，让我们从客户端的代码片段开始：

```
public class ChatClient {
    public ChatClient connect(int port,String host,final String nickName){
     EventLoopGroup group = new NioEventLoopGroup();
     try {
         Bootstrap bootstrap = new Bootstrap();
         bootstrap.group(group)
         .channel(NioSocketChannel.class)
         .option(ChannelOption.SO_KEEPALIVE, true)
         .handler(new ChannelInitializer<SocketChannel>() {
          @Override
          protected void initChannel(SocketChannel ch) throws Exception {
              ...
          }
         });
         //发起同步连接操作
         ChannelFuture channelFuture = bootstrap.connect(host, port).sync();
         channelFuture.channel().closeFuture().sync();
     } catch (InterruptedException e) {
         e.printStackTrace();
     }finally{
         //关闭，释放线程资源
         group.shutdownGracefully();
     }
     return this;
    }

    public static void main(String[] args) {
     new ChatClient().connect(8080, "localhost","Tom 老师");
    }
}
```

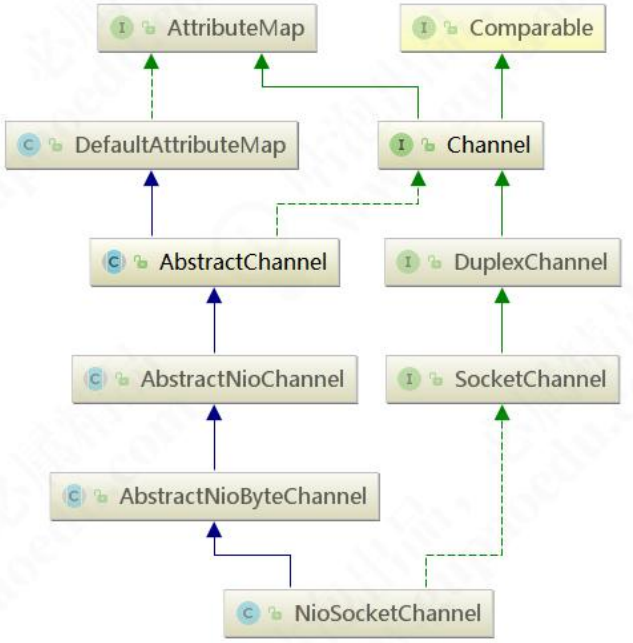
从上面的客户端代码虽然简单，但是却展示了 Netty 客户端初始化时所需的所有内容：

1、EventLoopGroup：不论是服务器端还是客户端，都必须指定 EventLoopGroup。在这个例子中，指定了 NioEventLoopGroup，表示一个 NIO 的 EventLoopGroup。

2、ChannelType: 指定 Channel 的类型。 因为是客户端，因此使用了 NioSocketChannel。

3、Handler: 设置处理数据的 Handler。

下面我们继续深入代码，看一下客户端通过 Bootstrap 启动后，都做了哪些工作？我们看一下 NioSocketChannel 的类层次结构如下:



回到我们在客户端连接代码的初始化 Bootstrap 中调用了一个 channel()方法，传入的参数是 NioSocketChannel.class,

在这个方法中其实就是初始化了一个 ReflectiveChannelFactory 的对象：

```
public B channel(Class<? extends C> channelClass) {
    if (channelClass == null) {
        throw new NullPointerException("channelClass");
    }
    return channelFactory(new ReflectiveChannelFactory<C>(channelClass));
}
```

而 ReflectiveChannelFactory 实现了 ChannelFactory 接口，它提供了唯一的方法，即 newChannel()方法，

ChannelFactory，顾名思义，就是创建 Channel 的工厂类。进入到 ReflectiveChannelFactory 的 newChannel()方法中，

我们看到其实现代码如下:

```
public T newChannel() {
    // 删除了 try...catch 块
    return clazz.newInstance();
}
```

根据上面代码的提示，我们就可以得出：

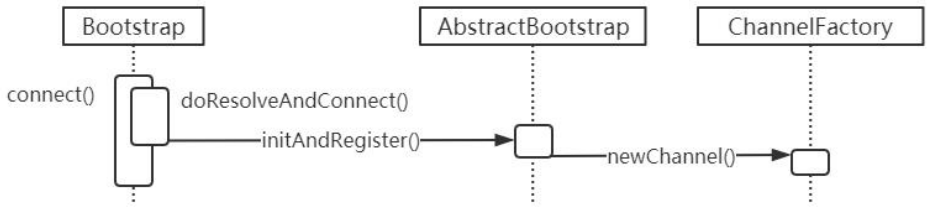
1、Bootstrap 中的 ChannelFactory 实现类是 ReflectiveChannelFactory。

2、通过 channel()方法创建的 Channel 具体类型是 NioSocketChannel。

Channel 的实例化过程其实就是调用 ChannelFactory 的 newChannel()方法，而实例化的 Channel 具体类型又是和初始化 Bootstrap 时传入的 channel()方法的参数相关。因此对于客户端的 Bootstrap 而言，创建的 Channel 实例就是 NioSocketChannel。

6.1.3 客户端 Channel 的初始化

前面我们已经知道了如何设置一个 Channel 的类型，并且了解到 Channel 是通过 ChannelFactory 的 newChannel()方法来实例化的，那么 ChannelFactory 的 newChannel()方法在哪里调用呢?继续跟踪，我们发现其调用链如下:



在 AbstractBootstrap 的 initAndRegister()中调用了 ChannelFactory()的 newChannel()来创建一个 NioSocketChannel 的实例，其源码如下：

```
final ChannelFuture initAndRegister() {
    // 去掉非关键代码
    Channel channel = channelFactory.newChannel();
    init(channel);

    ChannelFuture regFuture = config().group().register(channel);
    // 去掉非关键代码
    return regFuture;
}
```

在 newChannel()方法中，利用反射机制调用类对象的 newInstance()方法来创建一个新的 Channel 实例，相当于调用 NioSocketChannel 的默认构造器。NioSocketChannel 的默认构造器代码如下:

```
public NioSocketChannel() {
    this(DEFAULT_SELECTOR_PROVIDER);
}
```

这里的代码比较关键 ,我们看到 ,在这个构造器中会调用 newSocket()来打开一个新的 Java NIO 的 SocketChannel:

```
private static SocketChannel newSocket(SelectorProvider provider) {
       // 删除了 try...catch 块
           return provider.openSocketChannel();
```

```
}
```

接着会调用父类，即 AbstractNioByteChannel 的构造器：

```
AbstractNioByteChannel(Channel parent, SelectableChannel ch)
```

并传入参数 parent 为 null，ch 为刚才调用 newSocket()创建的 Java NIO 的 SocketChannel 对象，因此新创建的 NioSocketChannel 对象中 parent 暂时是 null。

```
protected AbstractNioByteChannel(Channel parent, SelectableChannel ch) {
    super(parent, ch, SelectionKey.OP_READ);
}
```

接着会继续调用父类 AbstractNioChannel 的构造器，并传入实际参数 readInterestOp = SelectionKey.OP_READ：

```
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp) {
        super(parent);
        this.ch = ch;
        this.readInterestOp = readInterestOp;
    // 省略...catch 块
        // 设置 Java NIO SocketChannel 为非阻塞的
        ch.configureBlocking(false);
}
```

然后继续调用父类 AbstractChannel 的构造器:

```
protected AbstractChannel(Channel parent) {
        this.parent = parent;
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
}
```

至此，NioSocketChannel 就初始化就完成了，我们可以稍微总结一下 NioSocketChannel 初始化所做的工作内容:

1、调用 NioSocketChannel.newSocket(DEFAULT_SELECTOR_PROVIDER)打开一个新的 Java NIOSocketChannel。

2、AbstractChannel(Channel parent)中需要初始化的属性:

id：每个 Channel 都拥有一个唯一的 id。

parent：属性置为 null。

unsafe :通过 newUnsafe()实例化一个 unsafe 对象，它的类型是 AbstractNioByteChannel.NioByteUnsafe 内部类。

pipeline：是 new DefaultChannelPipeline(this)新创建的实例。

3、AbstractNioChannel 中的属性:

ch：赋值为 Java SocketChannel，即 NioSocketChannel 的 newSocket()方法返回的 Java NIO SocketChannel。

readInterestOp：赋值为 SelectionKey.OP_READ

ch：被配置为非阻塞，即调用 ch.configureBlocking(false)。

4、NioSocketChannel 中的属性:

config = new NioSocketChannelConfig(this, socket.socket())

6.1.4 Unsafe 字段的初始化

我们简单地提到了，在实例化 NioSocketChannel 的过程中，会在父类 AbstractChannel 的构造方法中调用 newUnsafe()来获取一个 unsafe 实例。那么 unsafe 是怎么初始化的呢？它的作用是什么？

其实 unsafe 特别关键，它封装了对 Java 底层 Socket 的操作，因此实际上是沟通 Netty 上层和 Java 底层的重要的桥梁。那么我们下面看一下 Unsafe 接口所提供的方法吧：

```
interface Unsafe {
       RecvByteBufAllocator.Handle recvBufAllocHandle();
       SocketAddress localAddress();
       SocketAddress remoteAddress();
       void register(EventLoop eventLoop, ChannelPromise promise);
       void bind(SocketAddress localAddress, ChannelPromise promise);
       void connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise);
       void disconnect(ChannelPromise promise);
       void close(ChannelPromise promise);
       void closeForcibly();
       void deregister(ChannelPromise promise);
       void beginRead();
       void write(Object msg, ChannelPromise promise);
       void flush();

       ChannelPromise voidPromise();
       ChannelOutboundBuffer outboundBuffer();
   }
}
```

从源码中可以看出，这些方法其实都是对应到相关的 Java 底层的 Socket 的操作。

继续回到 AbstractChannel 的构造方法中，在这里调用了 newUnsafe()获取一个新的 unsafe 对象,而 newUnsafe()方法在 NioSocketChannel 中被重写了。来看代码:

```
protected AbstractNioUnsafe newUnsafe() {
    return new NioSocketChannelUnsafe();
}
```

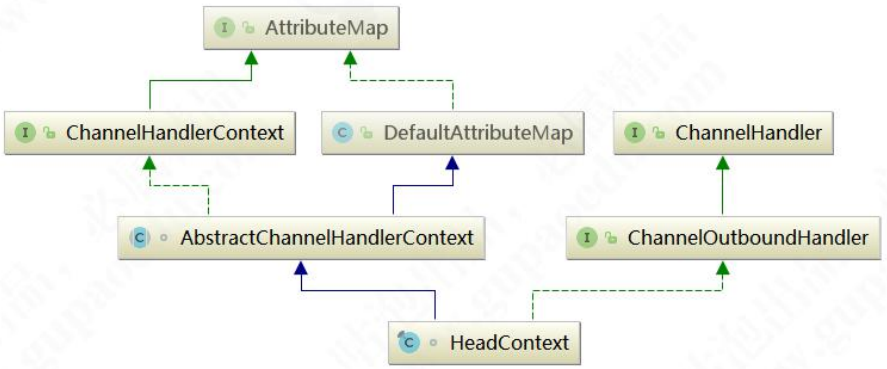
NioSocketChannel 的 newUnsafe()方法会返回一个 NioSocketChannelUnsafe 实例。从这里我们就可以确定了，在实例化的 NioSocketChannel 中的 unsafe 字段，其实是一个 NioSocketChannelUnsafe 的实例。

6.1.5 Pipeline 的初始化

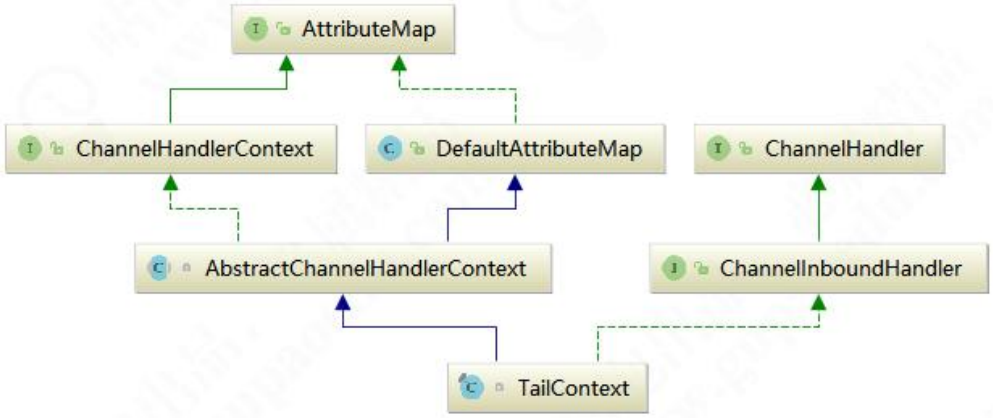
上面我们分析了 NioSocketChannel 的大体初始化过程，但是我们漏掉了一个关键的部分，即 ChannelPipeline 的初始化。在 Pipeline 的注释说明中写到"Each channel has its own pipeline and it is created automatically when a new channel is created."，我们知道,在实例化一个 Channel 时，必然都要实例化一个 ChannelPipeline。而我们确实在 AbstractChannel 的构造器看到了 pipeline 字段被初始化为 DefaultChannelPipeline 的实例。那么我们就来看一下，DefaultChannelPipeline 构造器做了哪些工作。

```
protected DefaultChannelPipeline(Channel channel) {
    this.channel = ObjectUtil.checkNotNull(channel, "channel");
    succeededFuture = new SucceededChannelFuture(channel, null);
    voidPromise =  new VoidChannelPromise(channel, true);

    tail = new TailContext(this);
    head = new HeadContext(this);

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

DefaultChannelPipeline 的构造器需要传入一个 channel，而这个 channel 其实就是我们实例化的 NioSocketChannel，DefaultChannelPipeline 会将这个 NioSocketChannel 对象保存在 channel 字段中。DefaultChannelPipeline 中还有两个特殊的字段，即 head 和 tail，这两个字段是双向链表的头和尾。其实在 DefaultChannelPipeline 中，维护了一个以 AbstractChannelHandlerContext 为节点元素的双向链表，这个链表是 Netty 实现 Pipeline 机制的关键。关于 DefaultChannelPipeline 中的双向链表以及它所起的作用 ,本节我们暂不讲解 ,后续再做深入分析。先看看 HeadContext 的类继承层次结构如下所示：



TailContext 的继承层次结构如下所示:



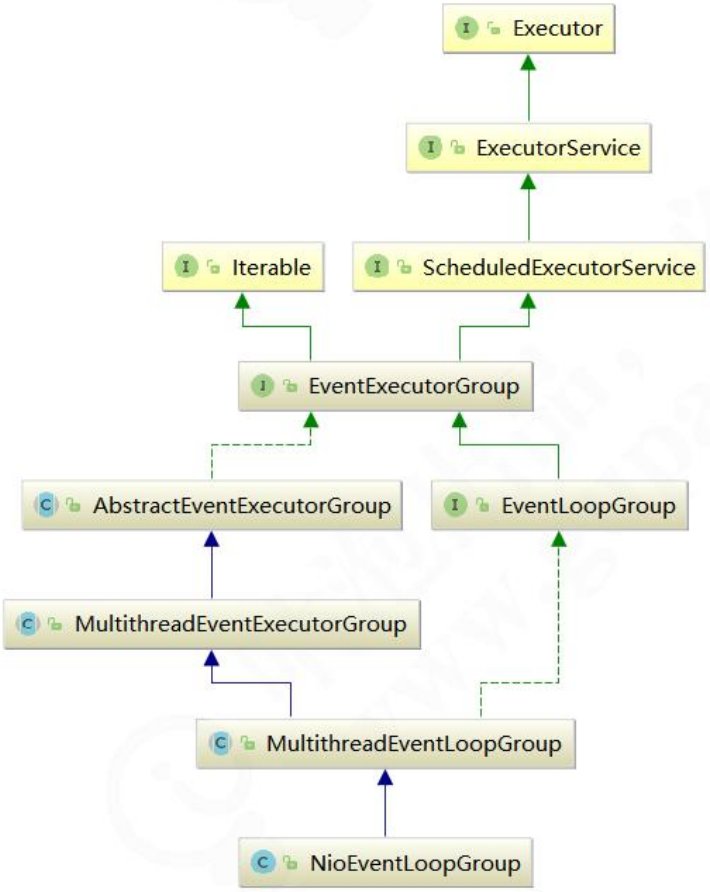
我们可以看到，链表中 head 是一个 ChannelOutboundHandler，而 tail 则是一个 ChannelInboundHandler。接着看 HeadContext 的构造器:

```
HeadContext(DefaultChannelPipeline pipeline) {
    super(pipeline, null, HEAD_NAME, false, true);
    unsafe = pipeline.channel().unsafe();
    setAddComplete();
}
```

它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = false，outbound = true。而 TailContext 的构造器与 HeadContext 的相反，它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = true，outbound = false。即 header 是一个 OutBoundHandler，而 tail 是一个 InBoundHandler，关于这一特征，大家要特别注意，先记住，后续我们分析到 Netty 的 Pipeline 时，我们会反复用到 inbound 和 outbound 这两个属性。

6.1.6 EventLoop 的初始化

回到最开始的 ChatClient 用户代码中，我们在一开始就实例化了一个 NioEventLoopGroup 对象，因此我们就从它的构造器中追踪一下 EventLoop 的初始化过程。首先来看一下 NioEventLoopGroup 的类继承层次：



NioEventLoop 有几个重载的构造器，不过内容都没有太大的区别，最终都是调用的父类 MultithreadEventLoopGroup 的构造器:

```
protected MultithreadEventLoopGroup(int nThreads, Executor executor, Object... args) {
    super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads, executor, args);
}
```

其中有个有意思的地方是，如果我们传入的线程数 nThreads 是 0，那么 Netty 会为我们设置默认的线程数 DEFAULT_EVENT_LOOP_THREADS，而这个默认的线程数是怎么确定的呢?

其实很简单，在静态代码块中，会首先确定 DEFAULT_EVENT_LOOP_THREADS 的值:

```
static {
    DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
            SystemPropertyUtil.getInt("io.netty.eventLoopThreads",Runtime.getRuntime().availableProcessors() * 2));
}
```

Netty 首先会从系统属性中获取"io.netty.eventLoopThreads"的值，如果我们没有设置的话，那么就返回默认值:即处理器核心数 * 2。回到 MultithreadEventLoopGroup 构造器中会继续调用父类 MultithreadEventExecutorGroup 的构造器:

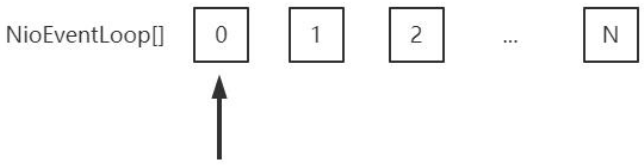
```
protected MultithreadEventExecutorGroup(int nThreads, Executor executor,
                                        EventExecutorChooserFactory chooserFactory, Object... args) {
```

```
      // 去掉了参数检查，异常处理等代码
       children = new EventExecutor[nThreads];

       for (int i = 0; i < nThreads; i ++) {
       // 去掉了 try...catch...finally 代码块
            children[i] = newChild(executor, args);
       }
       chooser = chooserFactory.newChooser(children);
       // 去掉了包装处理的代码
}
```

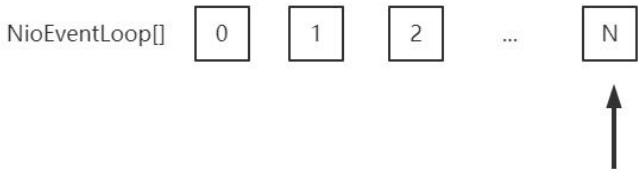
我们可以继续跟踪到 newChooser 方法里面看看其实现逻辑，具体代码如下：

```
public final class DefaultEventExecutorChooserFactory implements EventExecutorChooserFactory {

    //去掉了定义全局变量的代码

    public EventExecutorChooser newChooser(EventExecutor[] executors) {
        if (isPowerOfTwo(executors.length)) {
            return new PowerOfTwoEventExecutorChooser(executors);
        } else {
            return new GenericEventExecutorChooser(executors);
        }
    }

    private static boolean isPowerOfTwo(int val) {
        return (val & -val) == val;
    }

    private static final class PowerOfTwoEventExecutorChooser implements EventExecutorChooser {
        private final AtomicInteger idx = new AtomicInteger();
        private final EventExecutor[] executors;

        PowerOfTwoEventExecutorChooser(EventExecutor[] executors) {
            this.executors = executors;
        }

        public EventExecutor next() {
            return executors[idx.getAndIncrement() & executors.length - 1];
        }
    }

    private static final class GenericEventExecutorChooser implements EventExecutorChooser {
        private final AtomicInteger idx = new AtomicInteger();
        private final EventExecutor[] executors;

        GenericEventExecutorChooser(EventExecutor[] executors) {
            this.executors = executors;
        }

        public EventExecutor next() {
            return executors[Math.abs(idx.getAndIncrement() % executors.length)];
        }
    }
}
```

上面的代码逻辑主要表达的意思是：即如果 nThreads 是 2 的幂，则使用 PowerOfTwoEventExecutorChooser，否则使用 GenericEventExecutorChooser。这两个 Chooser 都重写 next()方法。next()方法的主要功能就是讲数组索引循环位移，如下图所示：



当索引移动最后一个位置时，再调用 next()方法就会将索引位置重新指向 0。



这个运算逻辑其实很简单，就是每次让索引自增后和数组长度取模：idx.getAndIncrement() % executors.length。但是就连一个非常简单的数组索引运算，Netty 都帮我们做了优化。因为在计算机底层，&与比%运算效率更高。

好了，分析到这里我们应该已经非常清楚 MultithreadEventExecutorGroup 中的处理逻辑，简单做一个总结：

1、创建一个大小为 nThreads 的 SingleThreadEventExecutor 数组。

2、根据 nThreads 的大小，创建不同的 Chooser，即如果 nThreads 是 2 的幂，则使用 PowerOfTwoEventExecutorChooser，反之使用 GenericEventExecutorChooser。不论使用哪个 Chooser，它们的功能都是一样的，即从 children 数组中选出一个合适的 EventExecutor 实例。

3、调用 newChhild()方法初始化 children 数组。

根据上面的代码，我们也能知道：MultithreadEventExecutorGroup 内部维护了一个 EventExecutor 数组，而 Netty 的 EventLoopGroup 的实现机制其实就建立在 MultithreadEventExecutorGroup 之上。每当 Netty 需要一个 EventLoop 时，会调用 next()方法获取一个可用的 EventLoop。

上面代码的最后一部分是 newChild()方法，这个是一个抽象方法，它的任务是实例化 EventLoop 对象。我们跟踪一下它的代码。可以发现。这个方法在 NioEventLoopGroup 类中有实现，其内容很简单：

```
protected EventLoop newChild(Executor executor, Object... args) throws Exception {
    return new NioEventLoop(this, executor, (SelectorProvider) args[0],
                      ((SelectStrategyFactory) args[1]).newSelectStrategy(), (RejectedExecutionHandler) args[2]);
}
```

其实逻辑很简单就是实例化一个 NioEventLoop 对象，然后返回 NioEventLoop 对象。

最后总结一下整个 EventLoopGroup 的初始化过程：

1、EventLoopGroup(其实是 MultithreadEventExecutorGroup)内部维护一个类型为 EventExecutor children 数组,其大小是 nThreads,这样就构成了一个线程池。

2、如果我们在实例化 NioEventLoopGroup 时，如果指定线程池大小，则 nThreads 就是指定的值，反之是处理器核心数 * 2。

3、MultithreadEventExecutorGroup 中会调用 newChild()抽象方法来初始化 children 数组。

4、抽象方法 newChild()是在 NioEventLoopGroup 中实现的，它返回一个 NioEventLoop 实例。

5、NioEventLoop 属性赋值：

provider：在 NioEventLoopGroup 构造器中通过 SelectorProvider.provider()获取一个 SelectorProvider。

selector：在 NioEventLoop 构造器中通过调用通过 provider.openSelector()方法获取一个 selector 对象。

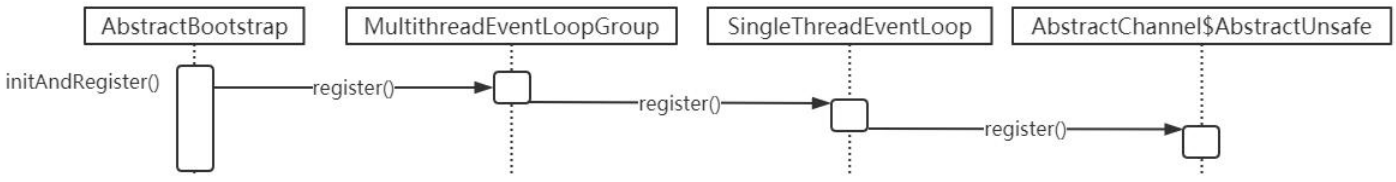
6.1.7 Channel 注册到 Selector

在前面的分析中，我们提到 Channel 会在 Bootstrap 的 initAndRegister()中进行初始化，但是这个方法还会将初始化好的 Channe 注册到 NioEventLoop 的 selector 中。接下来我们来分析一下 Channel 注册的过程。

再回顾一下 AbstractBootstrap 的 initAndRegister()方法：

```
final ChannelFuture initAndRegister() {
    // 删除了非关键代码
    Channelchannel = channelFactory.newChannel();
    init(channel);

    ChannelFuture regFuture = config().group().register(channel);

    return regFuture;
}
```

当 Channel 初始化后，紧接着会调用 group().register()方法来向 selector 注册 Channel。我们继续跟踪的话，会发现其调用链如下：



通过跟踪调用链，最终我们发现是调用到了 unsafe 的 register 方法，那么接下来我们就仔细看一下

AbstractChannel$AbstractUnsafe.register()方法中到底做了什么?

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
          // 省略了条件判断和错误处理的代码
          AbstractChannel.this.eventLoop = eventLoop;
          register0(promise);
}
```

首先，将 eventLoop 赋值给 Channel 的 eventLoop 属性，而我们知道这个 eventLoop 对象其实是 MultithreadEventLoopGroup 的 next()方法获取的，根据我们前面的分析，我们可以确定 next()方法返回的 eventLoop 对象是 NioEventLoop 实例。register()方法接着调用了 register0()方法：

```
private void register0(ChannelPromise promise) {
          // 省略了非关键代码
               boolean firstRegistration = neverRegistered;
               doRegister();
               neverRegistered = false;
               registered = true;

               pipeline.invokeHandlerAddedIfNeeded();

               safeSetSuccess(promise);
               pipeline.fireChannelRegistered();

               if (isActive()) {
                   if (firstRegistration) {
                       pipeline.fireChannelActive();
                   }
               }
}
```

register0()方法又调用了 AbstractNioChannel 的 doRegister()方法：

```
protected void doRegister() throws Exception {
    // 省略了错误处理的代码
    selectionKey = javaChannel().register(eventLoop().selector, 0, this);
}
```

看到 javaChannel()这个方法在前面我们已经知道了，它返回的是一个 Java NIO 的 SocketChannel 对象，这里我们将这个 SocketChannel 注册到与 eventLoop 关联的 selector 上了。

我们总结一下 Channel 的注册过程：

1、首先在 AbstractBootstrap 的 initAndRegister()方法中，通过 group().register(channel)，调用 MultithreadEventLoopGroup 的 register()方法。

2、在 MultithreadEventLoopGroup 的 register()中，调用 next()方法获取一个可用的 SingleThreadEventLoop，然后调用它的 register()方法。

3、在 SingleThreadEventLoop 的 register()方法中，调用 channel.unsafe().register(this, promise)方法来获取

channel 的 unsafe()底层操作对象，然后调用 unsafe 的 register()。

4、在 AbstractUnsafe 的 register()方法中，调用 register0()方法注册 Channel 对象。

5、在 AbstractUnsafe 的 register0()方法中，调用 AbstractNioChannel 的 doRegister()方法。

6、AbstractNioChannel 的 doRegister()方法通过 javaChannel().register(eventLoop().selector, 0, this)将 Channel 对应的 Java NIO 的 SocketChannel 注册到一个 eventLoop 的 selector 中，并且将当前 Channel 作为 attachment 与 SocketChannel 关联。

总的来说，Channel 注册过程所做的工作就是将 Channel 与对应的 EventLoop 关联，因此这也体现了，在 Netty 中，每个 Channel 都会关联一个特定的 EventLoop，并且这个 Channel 中的所有 IO 操作都是在这个 EventLoop 中执行的；当关联好 Channel 和 EventLoop 后，会继续调用底层 Java NIO 的 SocketChannel 对象的 register()方法，将底层 Java NIO 的 SocketChannel 注册到指定的 selector 中。通过这两步，就完成了 Netty 对 Channel 的注册过程。

6.1.8 Handler 的添加过程

Netty 有一个强大和灵活之处就是基于 Pipeline 的自定义 handler 机制。基于此，我们可以像添加插件一样自由组合各种各样的 handler 来完成业务逻辑。例如我们需要处理 HTTP 数据，那么就可以在 pipeline 前添加一个针对 HTTP 编、解码的 Handler，然后接着添加我们自己的业务逻辑的 handler，这样网络上的数据流就向通过一个管道一样，从不同的 handler 中流过并进行编、解码，最终在到达我们自定义的 handler 中。

说到这里，有些小伙伴肯定会好奇，既然这个 pipeline 机制是这么的强大，那么它是怎么实现的呢？在此我还不打算详细讲解，本节内容中，我们从简单的入手，先体验一下自定义的 handler 是如何以及何时添加到 ChannelPipeline 中的。首先我们看一下如下的用户代码片段：

```
// 此处省略 N 句代码
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
       ChannelPipeline pipeline = ch.pipeline();
       pipeline.addLast(new StringDecoder());
       pipeline.addLast(new StringEncoder());
       pipeline.addLast(new ChatClientHandler(nickName));
    }
```

```
    });
```

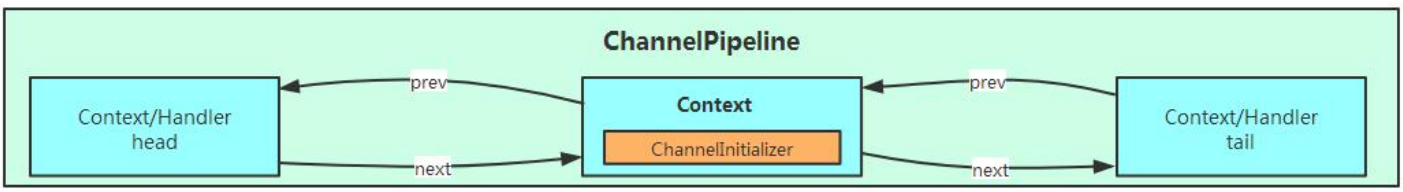
这个代码片段就是实现了 handler 的添加功能。我们看到，Bootstrap 的 handler()方法接收一个 ChannelHandler，而我们传的参数是一个派生于抽象类 ChannelInitializer 的匿名类，它当然也实现了 ChannelHandler 接口。我们来看一下，ChannelInitializer 类内到底有什么玄机：

```
public abstract class ChannelInitializer<C extends Channel> extends ChannelInboundHandlerAdapter {
    private static final InternalLogger logger = InternalLoggerFactory.getInstance(ChannelInitializer.class);
    private final ConcurrentMap<ChannelHandlerContext, Boolean> initMap = PlatformDependent.newConcurrentHashMap();

    protected abstract void initChannel(C ch) throws Exception;

    @Override
    @SuppressWarnings("unchecked")
    public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        if (initChannel(ctx)) {
            ctx.pipeline().fireChannelRegistered();
        } else {
            ctx.fireChannelRegistered();
        }
    }
// 这个方法在 channelRegistered 中被调用
    private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
        initChannel((C) ctx.channel());
        remove(ctx);
        return false;
    }
    // 省略...
}
```

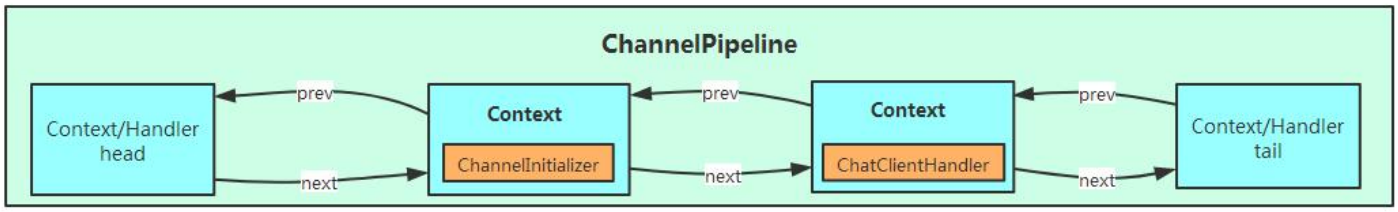
ChannelInitializer 是一个抽象类，它有一个抽象的方法 initChannel()，我们看到的匿名类正是实现了这个方法，并在这个方法中添加的自定义的 handler 的。那么 initChannel()是哪里被调用的呢?其实是在 ChannelInitializer 的 channelRegistered()方法中。

接下来关注一下 channelRegistered()方法。从上面的源码中，我们可以看到，在 channelRegistered()方法中，会调用 initChannel()方法，将自定义的 handler 添加到 ChannelPipeline 中，然后调用 ctx.pipeline().remove(this)方法将自己从 ChannelPipeline 中删除。上面的分析过程，如下图片所示：

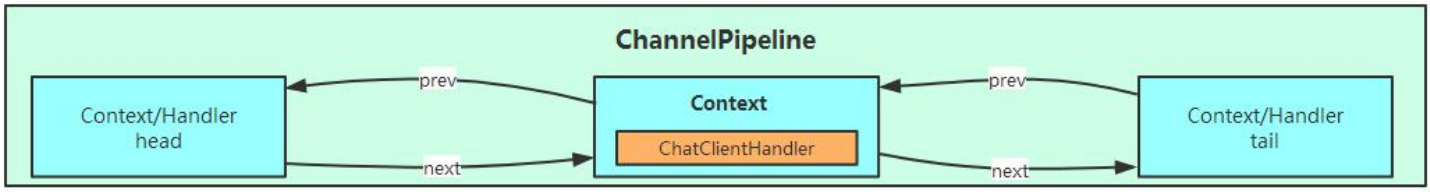
一开始，ChannelPipeline 中只有三个 handler，分别是：head、tail 和我们添加的 ChannelInitializer。



接着 initChannel()方法调用后，添加了自定义的 handler：



最后将 ChannelInitializer 删除：



分析到这里，我们已经简单了解了自定义的 handler 是如何添加到 ChannelPipeline 中的，之后的章节我们再进行深入的探讨。

6.1.9 客户端发起连接请求

经过上面的各种分析后，我们大致已经了解 Netty 客户端初始化时，所做的工作，那么接下来我们就直奔主题，分析一下客户端是如何发起 TCP 连接的？

首先，客户端通过调用 Bootstrap 的 connect()方法进行连接。在 connect()方法中，会进行一些参数检查后，最终调用的是 doConnect()方法，其代码实现如下：

```
private static void doConnect(
        final SocketAddress remoteAddress,
        final SocketAddress localAddress,
        final ChannelPromise connectPromise) {
        final Channel channel = connectPromise.channel();
        channel.eventLoop().execute(new Runnable() {
            @Override
            public void run() {
                if (localAddress == null) {
                    channel.connect(remoteAddress, connectPromise);
                } else {
                    channel.connect(remoteAddress, localAddress, connectPromise);
                }
                connectPromise.addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            }
       });
}
```

在 doConnect()方法中，会 eventLoop 线程中调用 Channel 的 connect()方法，而这个 Channel 的具体类型实际就是 NioSocketChannel，前面已经分析过了。继续跟踪到 channel.connect()方法中，我们发现它调用的是

DefaultChannelPipeline 的 connect()方法，pipeline 的 connect()方法代码如下：

```
public final ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {
       return tail.connect(remoteAddress, promise);
}
```

tail 我们已经分析过，是一个 TailContext 的实例，而 TailContext 又是 AbstractChannelHandlerContext 的子类，并且没有实现 connect()方法，因此这里调用的其实是 AbstractChannelHandlerContext 的 connect()方法，我们看一下这个方法的实现：

```
public ChannelFuture connect(
           final SocketAddress remoteAddress,
           final SocketAddress localAddress, final ChannelPromise promise) {
      // 删除参数检查的代码
       final AbstractChannelHandlerContext next = findContextOutbound();
       EventExecutor executor = next.executor();
       if (executor.inEventLoop()) {
           next.invokeConnect(remoteAddress, localAddress, promise);
       } else {
           safeExecute(executor, new Runnable() {
               @Override
               public void run() {
                   next.invokeConnect(remoteAddress, localAddress, promise);
               }
           }, promise, null);
       }
       return promise;
}
```

上面的代码中有一个关键的地方，即 final AbstractChannelHandlerContext next = findContextOutbound()，这里调用 findContextOutbound()方法 ,从 DefaultChannelPipeline 内的双向链表的 tail 开始 ,不断向前找到第一个 outbound 为 true 的 AbstractChannelHandlerContext，然后调用它的 invokeConnect()方法，其代码如下：

```
private void invokeConnect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise) {
 // 忽略 try...catch 块
 ((ChannelOutboundHandler) handler()).connect(this, remoteAddress, localAddress, promise);

}
```

前面我们有提到，在 DefaultChannelPipeline 的构造器中，实例化了两个对象：head 和 tail，并形成了双向链表的头和尾。head 是 HeadContext 的实例，它实现了 ChannelOutboundHandler 接口，并且它的 outbound 设置为 true。因此在 findContextOutbound()方法中，找到的 AbstractChannelHandlerContext 对象其实就是 head。进而在 invokeConnect()方法中，我们向上转换为 ChannelOutboundHandler 就是没问题的了。而又因为 HeadContext 重写了 connect()方法，因此实际上调用的是 HeadContext 的 connect()方法。我们接着跟踪到 HeadContext 的 connect()方法，看其代码如下：

```
public void connect(
               ChannelHandlerContext ctx,
               SocketAddress remoteAddress, SocketAddress localAddress,
               ChannelPromise promise) throws Exception {
           unsafe.connect(remoteAddress, localAddress, promise);
}
```

这个 connect()方法很简单，只是调用了 unsafe 的 connect()方法。回顾一下 HeadContext 的构造器，我们发现这个 unsafe 其实就是 pipeline.channel().unsafe()返回的 Channel 的 unsafe 字段。到这里为止，我们应该已经知道，其实是 AbstractNioByteChannel.NioByteUnsafe 内部类兜了一大圈圈。最后，我们找到创建 Socket 连接的关键代码继续跟踪，其实调用的就是 AbstractNioUnsafe 的 connect()方法：

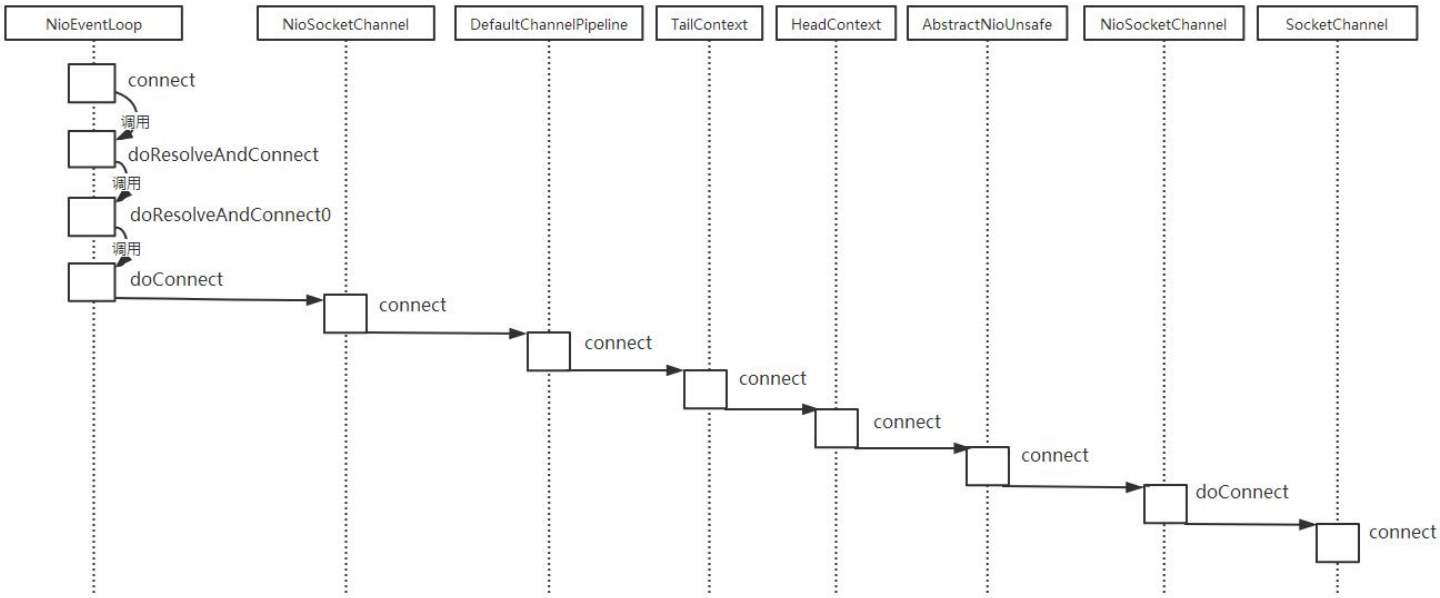
```
public final void connect(
               final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise) {

               // 省去前面的判断
               boolean wasActive = isActive();
               if (doConnect(remoteAddress, localAddress)) {
                   fulfillConnectPromise(promise, wasActive);
               } else {
                   // 此处省略 N 行代码
               }
}
```

在这个 connect()方法中，又调用了 doConnect()方法。注意：这个方法并不是 AbstractNioUnsafe 的方法，而是 AbstractNioChannel 的抽象方法。doConnect()方法是在 NioSocketChannel 中实现的，因此进入到 NioSocketChannel 的 doConnect()方法中：

```
protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress) throws Exception {
       if (localAddress != null) {
           doBind0(localAddress);
       }

       boolean success = false;
       try {
           boolean connected = javaChannel().connect(remoteAddress);
           if (!connected) {
               selectionKey().interestOps(SelectionKey.OP_CONNECT);
           }
           success = true;
           return connected;
       } finally {
           if (!success) {
               doClose();
           }
       }
}
```

我们终于看到的最关键的部分了，庆祝一下！上面的代码不用多说，首先是获取 Java NIO 的 SocketChannel，获取 NioSocketChannel 的 newSocket()返回的 SocketChannel 对象；然后调用 SocketChannel 的 connect()方法完成 Java

NIO 底层的 Socket 的连接。最后总结一下，客户端 BootStrap 发起连接请求的流程可以用如下时序图直观地展示：



6.2 服务端 ServerBootStrap

在分析客户端的代码时，我们已经对 Bootstrap 启动 Netty 有了一个大致的认识，那么接下来分析服务器端时，就会相对简单一些了。首先还是来看一下服务器端的启动代码：

```
public class ChatServer {
    public void start(int port) throws Exception{
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class)
             .childHandler(new ChannelInitializer<SocketChannel>(){
                    @Override
                    public void initChannel(SocketChannel ch) throws Exception {
                        ...

                    }
              })
             .option(ChannelOption.SO_BACKLOG, 128)
             .childOption(ChannelOption.SO_KEEPALIVE, true);

            System.out.println("服务已启动,监听端口" + port + "");

            // 绑定端口，开始接收进来的连接
            ChannelFuture f = b.bind(port).sync();

            // 等待服务器  socket 关闭 。
            // 在这个例子中，这不会发生，但你可以优雅地关闭你的服务器。
            f.channel().closeFuture().sync();
```

```
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();

            System.out.println("服务已关闭");
        }
    }


    public static void main(String[] args) {
     try {
         new ChatServer().start(8080);
     } catch (Exception e) {
         e.printStackTrace();
     }
    }
}
```

服务端基本写法和客户端的代码相比，没有很大的差别，基本上也是进行了如下几个部分的初始化：

1、EventLoopGroup：不论是服务器端还是客户端，都必须指定 EventLoopGroup。在上面的代码中，指定了 NioEventLoopGroup，表示一个 NIO 的 EventLoopGroup，不过服务器端需要指定两个 EventLoopGroup，一个是 bossGroup，用于处理客户端的连接请求；另一个是 workerGroup,用于处理与各个客户端连接的 IO 操作。

2、ChannelType: 指定 Channel 的类型。因为是服务器端，因此使用了 NioServerSocketChannel。

3、Handler：设置数据处理器。

6.2.1 NioServerSocktChannel 的创建

我们在分析客户端 Channel 初始化过程时已经提到，Channel 是对 Java 底层 Socket 连接的抽象，并且知道了客户端 Channel 的具体类型是 NioSocketChannel，那么，自然的服务端的 Channel 类型就是 NioServerSocketChannel 了。那么接下来我们按照分析客户端的流程对服务器端的代码也同样地分析一遍，这样也方便我们对比一下服务器端和客户端有哪些不一样的地方。通过前面的分析，我们已经知道了,在客户端中，Channel 类型的指定是在初始化时，通过 Bootstrap 的 channel()方法设置的，服务端也是同样的方式。

再看服务端代码，我们调用了 ServerBootstarap 的 channel(NioServerSocketChannel.class)方法，传的参数是 NioServerSocketChannel.class 对象。如此，按照客户端代码同样的流程，我们可以确定 NioServerSocketChannel 的实例化也是通过 ReflectiveChannelFactory 工厂类来完成的，而 ReflectiveChannelFactory 中的 clazz 字段被赋值为

NioServerSocketChannel.class，因此当调用 ReflectiveChannelFactory 的 newChannel()方法，就能获取到一个 NioServerSocketChannel 的实例。newChannel()方法的源代码如下：

```
public T newChannel() {
     // 删除了 try 块
     return clazz.newInstance();
}
```

最后我们也来总结一下:

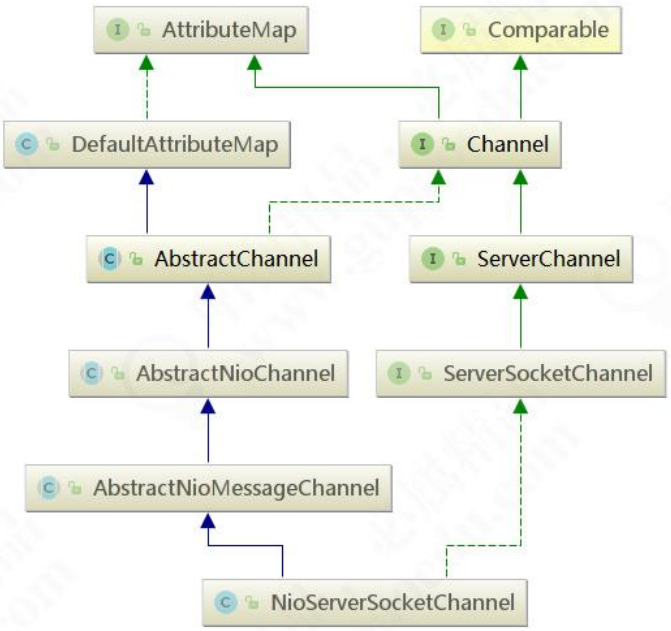
1、ServerBootstrap 中的 ChannelFactory 的实现类是 ReflectiveChannelFactory 类。

2、创建的 Channel 具体类型是 NioServerSocketChannel。

Channel 的实例化过程，其实就是调用 ChannelFactory 的 newChannel()方法，而实例化的 Channel 具体类型就是初始化 ServerBootstrap 时传给 channel()方法的实参。因此，上面代码案例中的服务端 ServerBootstrap，创建的 Channel 实例就是 NioServerSocketChannel 的实例。

6.2.3 服务端 Channel 的初始化

Jiexialai 我们来分析 NioServerSocketChannel 的实例化过程，先看一下 NioServerSocketChannel 的类层次结构图：



首先，我们来跟踪一下 NioServerSocketChannel 的默认构造，和 NioSocketChannel 类似，构造器都是调用 newSocket()来打开一个 Java 的 NIO Socket。不过需要注意的是，客户端的 newSocket()方法调用的是 openSocketChannel()，而

服务端的 newSocket()调用的是 openServerSocketChannel()。顾名思义，一个是客户端的 Java SocketChannel，一个是服务器端的 Java ServerSocketChannel，来看代码：

```
private static ServerSocketChannel newSocket(SelectorProvider provider) {
     return provider.openServerSocketChannel();
}
public NioServerSocketChannel() {
     this(newSocket(DEFAULT_SELECTOR_PROVIDER));
}
```

接下来会调用重载构造方法:

```
public NioServerSocketChannel(ServerSocketChannel channel) {
       super(null, channel, SelectionKey.OP_ACCEPT);
       config = new NioServerSocketChannelConfig(this, javaChannel().socket());
}
```

这个构造方法中，调用父类构造方法时传入的参数是 SelectionKey.OP_ACCEPT。作为对比，我们回顾一下，在客户端的 Channel 初始化时，传入的参数是 SelectionKey.OP_READ。在服务启动后需要监听客户端的连接请求，因此在这里我们设置 SelectionKey.OP_ACCEPT，也就是通知 selector 我们对客户端的连接请求感兴趣。

接着和客户端对比分析一下，会逐级地调用父类的构造器 NioServerSocketChannel -> AbstractNioMessageChannel -> AbstractNioChannel -> AbstractChannel。同样的，在 AbstractChannel 中实例化一个 unsafe 和 pipeline：

```
protected AbstractChannel(Channel parent) {
       this.parent = parent;
       id = newId();
       unsafe = newUnsafe();
       pipeline = newChannelPipeline();
}
```

不过，在这里需要注意的是，客户端的 unsafe 是 AbstractNioByteChannel#NioByteUnsafe 的实例，而服务端的 unsafe 是 AbstractNioMessageChannel.AbstractNioUnsafe 的实例。因为 AbstractNioMessageChannel 重写了 newUnsafe() 方法，其源代码如下：

```
protected AbstractNioUnsafe newUnsafe() {
       return new NioMessageUnsafe();
}
```

最后总结一下，在 NioServerSocketChannel 实例化过程中的执行逻辑：

1、调用 NioServerSocketChannel.newSocket(DEFAULT_SELECTOR_PROVIDER)方法打开一个新的 Java NIO ServerSocketChannel

2、AbstractChannel 初始化被赋值是属性：

parent：设置为 null

unsafe 通过 newUnsafe()实例化一个 unsafe 对象，类型是 AbstractNioMessageChannel#AbstractNioUnsafe

pipeline：创建实例 DefaultChannelPipeline 实例

3、AbstractNioChannel 中被赋值的属性：

ch：赋值为 Java NIO 的 ServerSocketChannel，调用 NioServerSocketChannel 的 newSocket()方法获取。

readInterestOp：默认赋值为 SelectionKey.OP_ACCEPT。

ch 设置为非阻塞，调用 ch.configureBlocking(false)方法。

4、NioServerSocketChannel 中被赋值的属性：

config = new NioServerSocketChannelConfig(this, javaChannel().socket())

6.2.4 ChannelPipeline 初始化

服务端 ChannelPipeline 的初始化和客户端一致，因此就不再单独分析了。

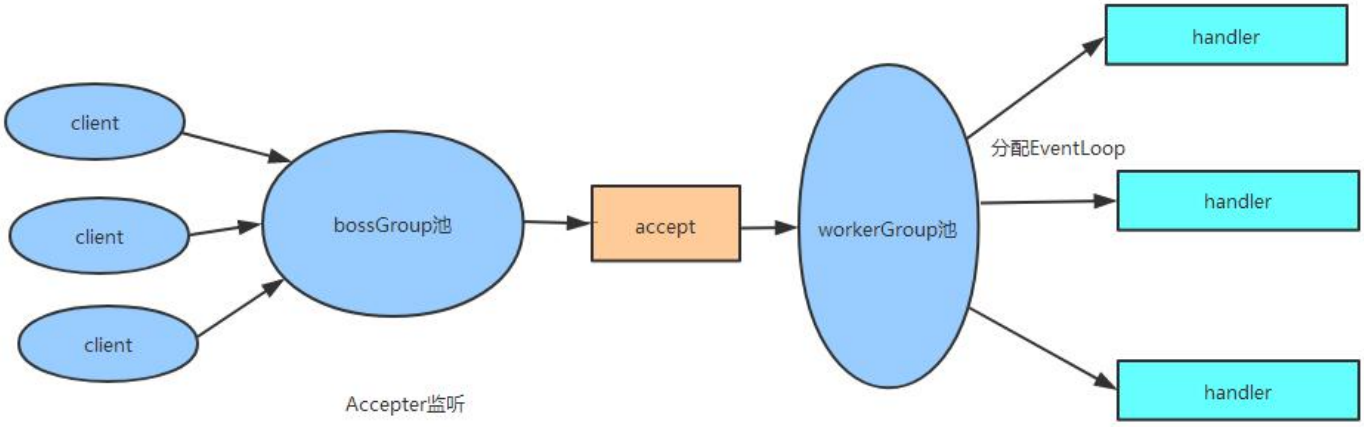
6.2.5 服务端 Channel 注册到 Selector

服务端 Channel 的注册过程和客户端一致，也不再单独分析了。

6.2.6 bossGroup 与 workerGroup

在客户端的时候，我们初始化了一个 EventLoopGroup 对象，而在服务端的初始化时，我们设置了两个 EventLoopGroup，一个是 bossGroup，另一个是 workerGroup。那么这两个 EventLoopGroup 都是干什么用的呢？接下来我们详细探究一下。其实，bossGroup 只用于服务端的 accept，也就是用于处理客户端新连接接入请求。我们可以把 Netty 比作一个餐馆，bossGroup 就像一个大堂经理，当客户来到餐馆吃时，大堂经理就会引导顾客就坐，为顾客端茶送水等。而 workerGroup 就是实际上干活的厨师，它们负责客户端连接通道的 IO 操作：当大堂经历接待顾客

后，顾客可以稍做休息，而此时后厨里的厨师们(workerGroup)就开始忙碌地准备饭菜了。关于 bossGroup 与 workerGroup 的关系，我们可以用如下图来展示，前面的章节我们也分析过，这里再巩固一下：



首先，服务端的 bossGroup 不断地监听是否有客户端的连接，当发现有一个新的客户端连接到来时，bossGroup 就会为此连接初始化各项资源，然后从 workerGroup 中选出一个 EventLoop 绑定到此客户端连接中。那么接下来的服务器与客户端的交互过程就全部在此分配的 EventLoop 中完成。口说无凭，我们还是以源码说话吧。

首先在 ServerBootstrap 初始化时，调用了 b.group(bossGroup, workerGroup)设置了两个 EventLoopGroup，我们跟踪进去以后会看到：

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
       super.group(parentGroup);
      // 此处省略 N 行代码
       this.childGroup = childGroup;
       return this;
}
```

显然，这个方法初始化了两个字段，一个是 group = parentGroup。它是在 super.group(parentGroup)中完成初始化的，另一个是 childGroup = childGroup。接着从应用程序的启动代码来看调用了 b.bind()方法来监听一个本地端口。

bind()方法会触发如下调用链：

AbstractBootstrap.bind() -> AbstractBootstrap.doBind() -> AbstractBootstrap.initAndRegister()

源码看到到这里为止，我们发现 AbstractBootstrap 的 initAndRegister()方法已经是我们的老朋友了，我们在分析客户端程序时和它打过很多交道，现在再来回顾一下这个方法吧：

```
final ChannelFuture initAndRegister() {
       // 省略异常判断
```

```
        Channel channel = channelFactory.newChannel();
        init(channel);
        // 省略非关键代码
        ChannelFuture regFuture = config().group().register(channel);

        return regFuture;
}
```

这里 group()方法返回的是上面我们提到的 bossGroup ,而这里的 channel 其实就是 NioServerSocketChannel 的实例，因此我们可以猜测 group().register(channel)将 bossGroup 和 NioServerSocketChannel 应该就关联起来了。那么 workerGroup 具体是在哪里与 NioServerSocketChannel 关联的呢？我们继续往下看 init(channel)方法：

```
void init(Channel channel) throws Exception {
       // 省略参数判断
        ChannelPipeline p = channel.pipeline();

        final EventLoopGroup currentChildGroup = childGroup;
        final ChannelHandler currentChildHandler = childHandler;
        final Entry<ChannelOption<?>, Object>[] currentChildOptions;
        final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

       // 省略非关键代码

        p.addLast(new ChannelInitializer<Channel>() {
            @Override
            public void initChannel(Channel ch) throws Exception {
                final ChannelPipeline pipeline = ch.pipeline();
                ChannelHandler handler = config.handler();
                if (handler != null) {
                    pipeline.addLast(handler);
                }
                ch.eventLoop().execute(new Runnable() {
                    @Override
                    public void run() {
                        pipeline.addLast(new ServerBootstrapAcceptor(
                                currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
                    }
                });
            }
        });
}
```

实际上 init()方法在 ServerBootstrap 中被重写了，从上面的代码片段中我们看到，它为 pipeline 中添加了一个 ChannelInitializer,而这个 ChannelInitializer 中添加了一个非常关键的 ServerBootstrapAcceptor 的 handler。关于 handler 的添加与初始化的过程，我们留到之后的章节再详细分析。现在，我们来关注一下 ServerBootstrapAcceptor 类。在 ServerBootstrapAcceptor 中重写了 channelRead()方法，其主要代码如下：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
            final Channel child = (Channel) msg;
            child.pipeline().addLast(childHandler);
            // 省略非关键代码
                childGroup.register(child).addListener(...);
}
```

ServerBootstrapAcceptor 中的 childGroup 是构造此对象是传入的 currentChildGroup，也就是 workerGroup 对象。而这里的 Channel 是 NioSocketChannel 的实例，因此这里的 childGroup 的 register()方法就是将 workerGroup 中的某个 EventLoop 和 NioSocketChannel 关联上了。既然如此，那么现在的问题是 ServerBootstrapAcceptor 的 channelRead()方法是在哪里被调用的呢？其实当一个 client 连接到 server 时，Java 底层 NIO 的 ServerSocketChannel 就会有一个 SelectionKey.OP_ACCEPT 的事件就绪，接着就会调用到 NioServerSocketChannel 的 doReadMessages()方法：

```
protected int doReadMessages(List<Object> buf) throws Exception {
        SocketChannel ch = javaChannel().accept();
        // 省略异常处理
        buf.add(new NioSocketChannel(this, ch));
        return 1;
        // 省略错误处理
}
```

在 doReadMessages()方法中，通过调用 javaChannel().accept()方法获取到客户端新连接的 SocketChannel 对象，紧接着就实例化一个 NioSocketChannel，并且传入 NioServerSocketChannel 对象(即 this)。由此可知，我们创建的这个 NioSocketChannel 的父类 Channel 就是 NioServerSocketChannel 实例。接下来就经由 Netty 的 ChannelPipeline 机制 将读取事件逐级发送到各个 handler 中，于是就会触发前面我们提到的 ServerBootstrapAcceptor 的 channelRead()方法。

6.2.6 服务端 Selector 事件轮询

再回到服务端 ServerBootStrap 的启动代码，是从 bind()方法开始的。ServerBootStrapt 的 bind()方法实际上就是其父类 AbstractBootstrap 的 bind()方法，来看代码：

```
    private static void doBind0(
            final ChannelFuture regFuture, final Channel channel,
            final SocketAddress localAddress, final ChannelPromise promise) {

        channel.eventLoop().execute(new Runnable() {
            @Override
            public void run() {
                if (regFuture.isSuccess()) {
                    channel.bind(localAddress, promise).addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
                } else {
                    promise.setFailure(regFuture.cause());
                }
            }
        });
```

```
    }
```

在 doBind0()方法中，调用的是 EventLoop 的 execute()方法，我们继续跟进去：

```
public void execute(Runnable task) {
        //省略了空判断

        boolean inEventLoop = inEventLoop();
        if (inEventLoop) {
            addTask(task);
        } else {
            startThread();
            addTask(task);
            //省略删除任务的逻辑
        }

        //省略判断逻辑
    }
```

在 execute()主要就是创建线程，将线程添加到 EventLoop 的无锁化串行任务队列。我们重点关注 startThread()方法，

继续看源代码：

```
    private void startThread() {
        if (state == ST_NOT_STARTED) {
            if (STATE_UPDATER.compareAndSet(this, ST_NOT_STARTED, ST_STARTED)) {
                doStartThread();
            }
        }
    }

    private void doStartThread() {
                //省略部分代码
                  SingleThreadEventExecutor.this.run();
                 //省略部分代码
    }
```

我们发现 startThread()最终调用的是 SingleThreadEventExecutor.this.run()方法，这个 this 就是 NioEventLoop 对象：

```
protected void run() {
        for (;;) {
                switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {
                    case SelectStrategy.CONTINUE:
                        continue;
                    case SelectStrategy.SELECT:
                        select(wakenUp.getAndSet(false));
                       //省略 select 的唤醒逻辑
                    default:
                }

                cancelledKeys = 0;
                needsToSelectAgain = false;
                final int ioRatio = this.ioRatio;
                if (ioRatio == 100) {
                        processSelectedKeys();
                    //省略异常处理理
                } else {
                    final long ioStartTime = System.nanoTime();
                        processSelectedKeys();
                    //省略异常处理
                }
        }
    }
```

终于看到似曾相识的代码。上面代码主要就是用一个死循环，在不断地轮询 SelectionKey。select()方法，主要用来解决 JDK 空轮训 Bug，而 processSelectedKeys()就是针对不同的轮询事件进行处理。如果客户端有数据写入，最终也会调用 AbstractNioMessageChannel 的 doReadMessages()方法。总结一下：

1、Netty 中 Selector 事件轮询是从 EventLoop 的 execute()方法开始的。

2、在 EventLoop 的 execute()方法中，会为每一个任务创建一个独立的线程，并保存到无锁化串行任务队列。

3、线程任务队列的每个任务实际调用的是 NioEventLoop 的 run()方法。

4、在 run 方法中调用 processSelectedKeys()处理轮询事件。

6.2.7 Netty 解决 JDK 空轮训 Bug

各位应该早有耳闻臭名昭著的 Java NIO epoll 的 bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK1.6 版本的 update18 修复了该问题，但是直到 JDK1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有被根本解决。出现此 Bug 是因为当 Selector 的轮询结果为空，也没有 wakeup 或新消息处理，则发生空轮询，CPU 使用率达到 100%。我们来看下这个问题在 issue 中的原始描述：

> This is an issue with poll (and epoll) on Linux. If a file descriptor for a connected socket is polled with a request event mask of 0, and if the connection is abruptly terminated (RST) then the poll wakes up with the POLLHUP (and maybe POLLERR) bit set in the returned event set. The implication of this behaviour is that Selector will wakeup and as the interest set for the SocketChannel is 0 it means there aren't any selected events and the select method returns 0.

具体解释为：在部分 Linux 的 2.6 的 kernel 中，poll 和 epoll 对于突然中断的连接 socket 会对返回的 eventSet 事件集合置为 POLLHUP，也可能是 POLLERR，eventSet 事件集合发生了变化，这就可能导致 Selector 会被唤醒。

这是与操作系统机制有关系的，JDK 虽然仅仅是一个兼容各个操作系统平台的软件，但很遗憾在 JDK5 和 JDK6 最初的版本中（严格意义上来将，JDK 部分版本都是），这个问题并没有解决，而将这个帽子抛给了操作系统方，这也就是这个 bug 最终一直到 2013 年才最终修复的原因。

在 Netty 中最终的解决办法是：创建一个新的 Selector，将可用事件重新注册到新的 Selector 中来终止空轮训。回顾

事件轮询的关键代码：

```
protected void run() {
        for (;;) {
                switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {
                    case SelectStrategy.CONTINUE:
                        continue;
                    case SelectStrategy.SELECT:
                        select(wakenUp.getAndSet(false));
                       //省略 select 的唤醒逻辑
                    default:
                }

                //事件轮询处理逻辑
        }
}
```

前面我们有提到 select()方法解决了 JDK 空轮训的 Bug，它到底是如何解决的呢？下面我们来一探究竟，进入 select()

方法的源码：

```
public final class NioEventLoop extends SingleThreadEventLoop {

    ...

    int selectorAutoRebuildThreshold = SystemPropertyUtil.getInt("io.netty.selectorAutoRebuildThreshold", 512);
    //省略判断代码
        SELECTOR_AUTO_REBUILD_THRESHOLD = selectorAutoRebuildThreshold;
    ...
    private void select(boolean oldWakenUp) throws IOException {
        Selector selector = this.selector;
        long currentTimeNanos = System.nanoTime();
         for (;;) {
             //省略非关键代码
               long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;
               int selectedKeys = selector.select(timeoutMillis);
               selectCnt ++;

              //省略非关键代码

               long time = System.nanoTime();
               if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos) {
                   // timeoutMillis elapsed without anything selected.
                   selectCnt = 1;
               } else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
                       selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
                   // 日志打印代码

                   rebuildSelector();
                   selector = this.selector;

                   // Select again to populate selectedKeys.
                   selector.selectNow();
                   selectCnt = 1;
                   break;
               }

               currentTimeNanos = time;
           }

           //省略非关键代码
```

```
        }
    }
    ...
}
```

从上面的代码中可以看出,Selector 每一次轮询都计数 selectCnt++，开始轮询会计时赋值给 timeoutMillis，轮询完成会计时赋值给 time，这两个时间差会有一个时间差，而这个时间差就是每次轮询所消耗的时间。从上面的的逻辑看出，如果每次轮询消耗的时间为 0，且重复次数超过 512 次，则调用 rebuildSelector()方法，即重构 Selector。我们跟进到源码中就会发现：

```
public void rebuildSelector() {
    //省略判断语句
    rebuildSelector0();
}

private void rebuildSelector0() {
    final Selector oldSelector = selector;
    final SelectorTuple newSelectorTuple;

    newSelectorTuple = openSelector();
   //省略非关键代码

    // Register all channels to the new Selector.
    int nChannels = 0;
    for (SelectionKey key: oldSelector.keys()) {
       //省略非关键代码和异常处理
            key.cancel();
            SelectionKey newKey = key.channel().register(newSelectorTuple.unwrappedSelector, interestOps, a);

    }

    //省略非关键代码
}
```

在 rebuildSelector()方法中，主要做了三件事情：

1、创建一个新的 Selector。

2、将原来 Selector 中注册的事件全部取消。

3、将可用事件重新注册到新的 Selector 中，并激活。

6.2.8 Netty 对 Selector 中 KeySet 的优化

分析完 Netty 对 JDK 空轮训 Bug 的解决方案，接下来我们再来看一个很有意思的细节。Netty 对 Selector 中存储 SelectionKey 的 HashSet 也做了优化。在前面的分析中，Netty 对 Selector 有重构，创建一个新的 Selector 其实是调用 openSelector()方法，来看代码：

```
   private void rebuildSelector0() {
       final Selector oldSelector = selector;
       final SelectorTuple newSelectorTuple;
       newSelectorTuple = openSelector();
      //省略非关键代码
   }
```

下面我们进入到 openSelector()方法的代码逻辑中：

```
   private SelectorTuple openSelector() {
      final Selector unwrappedSelector;
             //省略异常处理代码
              unwrappedSelector = provider.openSelector();
       //省略非关键代码

       final SelectedSelectionKeySet selectedKeySet = new SelectedSelectionKeySet();

       Object maybeSelectorImplClass = AccessController.doPrivileged(new PrivilegedAction<Object>() {
           @Override
           public Object run() {
                   return Class.forName(
                           "sun.nio.ch.SelectorImpl",
                           false,
                           PlatformDependent.getSystemClassLoader());
               //省略异常处理代码
           }
       });

       //省略非关键代码

       final Class<?> selectorImplClass = (Class<?>) maybeSelectorImplClass;

       Object maybeException = AccessController.doPrivileged(new PrivilegedAction<Object>() {
           @Override
           public Object run() {

                   Field selectedKeysField = selectorImplClass.getDeclaredField("selectedKeys");
                   Field publicSelectedKeysField = selectorImplClass.getDeclaredField("publicSelectedKeys");

                   //省略非关键代码

                   selectedKeysField.set(unwrappedSelector, selectedKeySet);
                   publicSelectedKeysField.set(unwrappedSelector, selectedKeySet);
                   return null;
                   //省略异常处理代码
           }
       });

       //省略非关键代码
   }
```

上面代码的主要功能就是利用反射机制，获取到 JDK 底层的 Selector 的 class 对象，用反射方法从 class 对象中获得两个字段 selectedKeys 和 publicSelectedKeys，这两个字段就是用来存储已注册事件的。然后，将这个两个对象重新赋值为 Netty 创建的 SelectedSelectionKeySet，是不是有种"偷梁换柱"的感觉？

我们先来看 selectedKeys 和 publicSelectedKeys 到底是什么类型，打开 SelectorImpl 的源码，看起构造方法：

```
public abstract class SelectorImpl extends AbstractSelector {
   protected Set<SelectionKey> selectedKeys = new HashSet();
```

```
    protected HashSet<SelectionKey> keys = new HashSet();
    private Set<SelectionKey> publicKeys;
    private Set<SelectionKey> publicSelectedKeys;

    protected SelectorImpl(SelectorProvider var1) {
        //省略非关键代码
            this.publicKeys = this.keys;
            this.publicSelectedKeys = this.selectedKeys;
         //省略非关键代码
    }
     ...
}
```

我们发现 selectedKeys 和 publicSelectedKeys 就是 HashSet。下面我们再来看 Netty 创建的 SelectedSelectionKeySet 对象的源代码：

```
final class SelectedSelectionKeySet extends AbstractSet<SelectionKey> {

    SelectionKey[] keys;
    int size;

    SelectedSelectionKeySet() {
        keys = new SelectionKey[1024];
    }

    @Override
    public boolean add(SelectionKey o) {
        if (o == null) {
            return false;
        }

        keys[size++] = o;
        if (size == keys.length) {
            increaseCapacity();
        }

        return true;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean remove(Object o) {
        return false;
    }

    @Override
    public boolean contains(Object o) {
        return false;
    }

    @Override
    public Iterator<SelectionKey> iterator() {
        throw new UnsupportedOperationException();
    }

    void reset() {
        reset(0);
```

```
    }

    void reset(int start) {
        Arrays.fill(keys, start, size, null);
        size = 0;
    }

    private void increaseCapacity() {
        SelectionKey[] newKeys = new SelectionKey[keys.length << 1];
        System.arraycopy(keys, 0, newKeys, 0, size);
        keys = newKeys;
    }
}
```

源码篇幅不长 ，但很精辟。SelectedSelectionKeySet 同样继承了 AbstractSet，因此赋值给 selectedKeys 和 publicSelectedKeys 不存在类型强制转换的问题。细心的小伙伴应该已经发现在 SelectedSelectionKeySet 中禁用了 remove()方法、contains()方法和 iterator()方法 ,只保留 add()方法 ,而且底层存储结构用的是数组 SelectionKey[] keys。那么，Netty 为什么要这样设计呢？主要目的还是简化我们在轮询事件时的操作，不需要每次轮询都要移除 key。

6.2.9 Handler 的添加过程

服务端 handler 的添加过程和客户端的有点区别，跟 EventLoopGroup 一样服务端的 handler 也有两个：一个是通过 handler()方法设置的 handler，另一个是通过 childHandler()方法设置的 childHandler。通过前面的 bossGroup 和 workerGroup 的分析，其实我们在这里可以大胆地猜测：handler 与 accept 过程有关。即 handler 负责处理客户端新连接接入的请求 ;而 childHandler 就是负责和客户端连接的 IO 交互。那么实际上是不是这样的呢？我们继续用代码来证明。在前面章节我们已经了解 ServerBootstrap 重写了 init()方法，在这个方法中也添加了 handler：

```
void init(Channel channel) throws Exception {
      // 省去逻辑判断
       ChannelPipeline p = channel.pipeline();

       final EventLoopGroup currentChildGroup = childGroup;
       final ChannelHandler currentChildHandler = childHandler;
       final Entry<ChannelOption<?>, Object>[] currentChildOptions;
       final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

       p.addLast(new ChannelInitializer<Channel>() {
           @Override
           public void initChannel(Channel ch) throws Exception {
               final ChannelPipeline pipeline = ch.pipeline();
               ChannelHandler handler = config.handler();
               if (handler != null) {
                   pipeline.addLast(handler);
               }

               ch.eventLoop().execute(new Runnable() {
```

```
                @Override
                public void run() {
                    pipeline.addLast(new ServerBootstrapAcceptor(
                            currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));
                }
            });
        }
    });
}
```

在上面代码的 initChannel()方法中，首先通过 handler()方法获取一个 handler，如果获取的 handler 不为空，则添加到 pipeline 中。然后接着，添加了一个 ServerBootstrapAcceptor 的实例。那么这里的 handler()方法返回的是哪个对象呢？其实它返回的是 handler 字段，而这个字段就是我们在服务器端的启动代码中设置的：

```
b.group(bossGroup, workerGroup)
```

那么这个时候, pipeline 中的 handler 情况如下:

ServerBootstrap.init方法中

根据我们原来客户端代码的分析来，我们指定 channel 绑定到 eventLoop(在这里是指 NioServerSocketChannel 绑定到 bossGroup)后 ,会在 pipeline 中触发 fireChannelRegistered 事件 ,接着就会触发对 ChannelInitializer 的 initChannel()方法的调用。因此在绑定完成后，此时的 pipeline 的内如下：

channelRegistered后的pipeline内容

在前面我们分析 bossGroup 和 workerGroup 时,已经知道了 ServerBootstrapAcceptor 的 channelRead()方法会为新建的 Channel 设置 handler 并注册到一个 eventLoop 中，即：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
          final Channel child = (Channel) msg;
          child.pipeline().addLast(childHandler);
          // 省去非关键代码
          childGroup.register(child).addListener(...);

}
```

而这里的 childHandler 就是我们在服务器端启动代码中设置的 handler：

```
...
.childHandler(new ChannelInitializer<SocketChannel>(){
```

```
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ...
                }
            })
```

后续的步骤我们基本上已经清楚了，当客户端连接 Channel 注册后，就会触发 ChannelInitializer 的 initChannel()方法的调用。最后我们来总结一下服务端 handler 与 childHandler 的区别与联系：

1、在服务器 NioServerSocketChannel 的 pipeline 中添加的是 handler 与 ServerBootstrapAcceptor。

2、当有新的客户端连接请求时，调用 ServerBootstrapAcceptor 的 channelRead()方法创建此连接的 NioSocketChannel 并添加 childHandler 到 NioSocketChannel 对应的 pipeline 中，并将此 channel 绑定到 workerGroup 中的某个 eventLoop 中。

3、handler 是在 accept 阶段起作用，它处理客户端的连接请求。

4、childHandler 是在客户端连接建立以后起作用，它负责客户端连接的 IO 交互。

最后来看一张图，加深理解。下图描述了服务端从启动初始化到有新连接接入的变化过程：