3

# 第 3 篇

## Netty 核心篇

7

# 第 7 章

# 大名鼎鼎的 EventLoop

## 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。

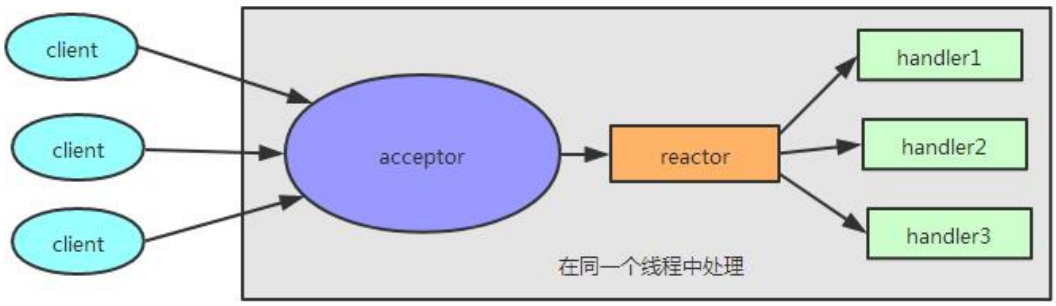
2、未来可能参与中间件开发的人群。

## 7.1 EventLoopGroup 与 Reactor

前面的章节中我们已经知道了，一个 Netty 程序启动时，至少要指定一个 EventLoopGroup(如果使用到的是 NIO，通常是指 NioEventLoopGroup)，那么，这个 NioEventLoopGroup 在 Netty 中到底扮演着什么角色呢？我们知道，Netty 是 Reactor 模型的一个实现，我们就从 Reactor 的线程模型开始。

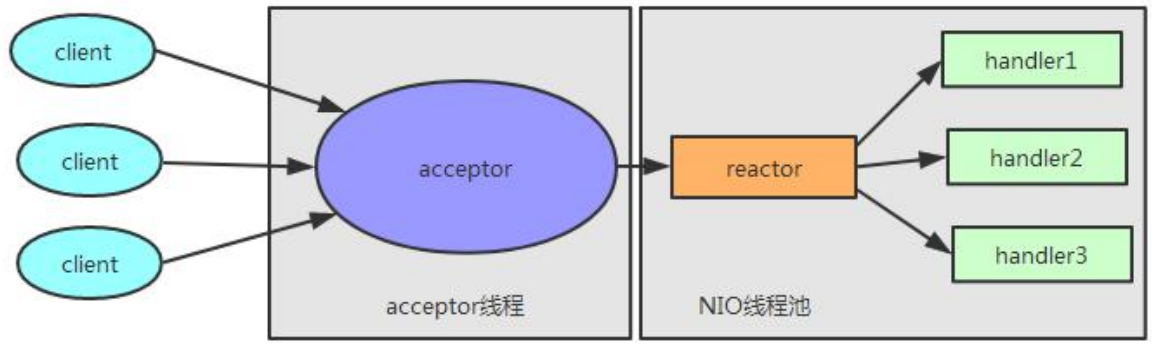
### 7.1.1 浅谈 Reactor 线程模型

Reactor 的线程模型有三种:单线程模型、多线程模型、主从多线程模型。首先来看一下单线程模型，如下图所示：




所谓单线程，即 Acceptor 处理和 andler 处理都在同一个线程中处理。这个模型的坏处显而易见：当其中某个 Handler 阻塞时，会导致其他所有的 Client 的 Handler 都得不到执行，并且更严重的是，Handler 的阻塞也会导致整个服务不能接收新的 Client 请求(因为 Acceptor 也被阻塞了)。因为有这么多的缺陷，因此单线程 Reactor 模型应用场景比较少。

那么，什么是多线程模型呢? Reactor 的多线程模型与单线程模型的区别就是 Acceptor 是一个单独的线程处理，并且有一组特定的 NIO 线程来负责各个客户端连接的 IO 操作。Reactor 多线程模型如下图所示：




Reactor 多线程模型有如下特点:

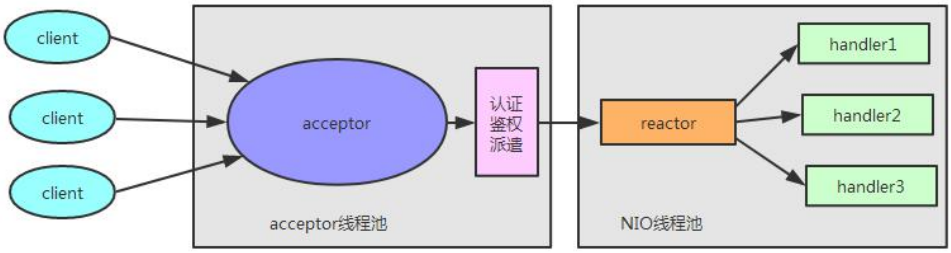
1、有专门一个线程，即 Acceptor 线程用于监听客户端的 TCP 连接请求。

2、客户端连接的 IO 操作都由一个特定的 NIO 线程池负责.每个客户端连接都与一个特定的 NIO 线程绑定,因此在这个客户端连接中的所有 IO 操作都是在同一个线程中完成的。

3、客户端连接有很多，但是 NIO 线程数是比较少的，因此一个 NIO 线程可以同时绑定到多个客户端连接中。

接下来我们再来看一下 Reactor 的主从多线程模型。一般情况下, Reactor 的多线程模式已经可以很好的工作了，但是我们想象一个这样的场景：如果我们的服务器需要同时处理大量的客户端连接请求或我们需要在客户端连接时，进行一些权限的校验，那么单线程的 Acceptor 很有可能就处理不过来，造成了大量的客户端不能连接到服务器。

Reactor 的主从多线程模型就是在这样的情况下提出来的，它的特点是：服务器端接收客户端的连接请求不再是一个线程，而是由一个独立的线程池组成。其线程模型如下图所示：




可以看到，Reactor 的主从多线程模型和 Reactor 多线程模型很类似，只不过 Reactor 的主从多线程模型的 Acceptor 使用了线程池来处理大量的客户端请求。

### 7.1.2 EventLoopGroup 与 Reactor 关联

我们介绍了三种 Reactor 的线程模型，那么它们和 NioEventLoopGroup 又有什么关系呢?其实，不同的设置 NioEventLoopGroup 的方式就对应了不同的 Reactor 的线程模型。

1、单线程模型，来看下面的应用代码：

```
    EventLoopGroup bossGroup = new NioEventLoopGroup(1);
    ServerBootstrap server = new ServerBootstrap();
    server.group(bossGroup);
```

注意，我们实例化了一个 NioEventLoopGroup，然后接着我们调用 server.group(bossGroup)设置了服务器端的 EventLoopGroup。有人可能会有疑惑 ,我记得在启动服务器端的 Netty 程序时，需要设置 bossGroup 和 workerGroup，为何这里只设置一个 bossGroup ? 其实原因很简单，ServerBootstrap 重写了 group 方法：

```
public ServerBootstrap group(EventLoopGroup group) {
       return group(group, group);
}
```

因此当传入一个 group 时，那么 bossGroup 和 workerGroup 就是同一个 NioEventLoopGroup 了。这时，因为 bossGroup 和 workerGroup 就是同一个 NioEventLoopGroup，并且这个 NioEventLoopGroup 线程池数量只设置了 1 个线程，也就是说 Netty 中的 Acceptor 和后续的所有客户端连接的 IO 操作都是在一个线程中处理的。那么对应到 Reactor 的线程模型中，我们这样设置 NioEventLoopGroup 时，就相当于 Reactor 的单线程模型。

2、多线程模型，再来看下面的应用代码：

```
    EventLoopGroup bossGroup = new NioEventLoopGroup(128);
    ServerBootstrap server = new ServerBootstrap();
    server.group(bossGroup);
```

从上面代码中可以看出，我们只需要将 bossGroup 的参数就设置为大于 1 的数，其实就是 Reactor 多线程模型。

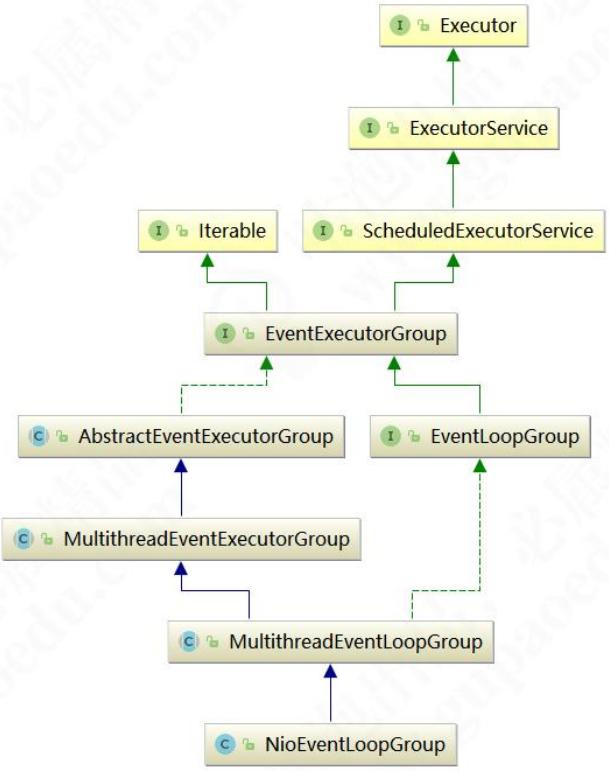
3、主从线程模型，到这里相信大家都已经想到了，实现主从线程模型的代码如下:

```
       EventLoopGroup bossGroup = new NioEventLoopGroup();
       EventLoopGroup workerGroup = new NioEventLoopGroup();
       ServerBootstrap b = new ServerBootstrap();
       b.group(bossGroup, workerGroup);
```

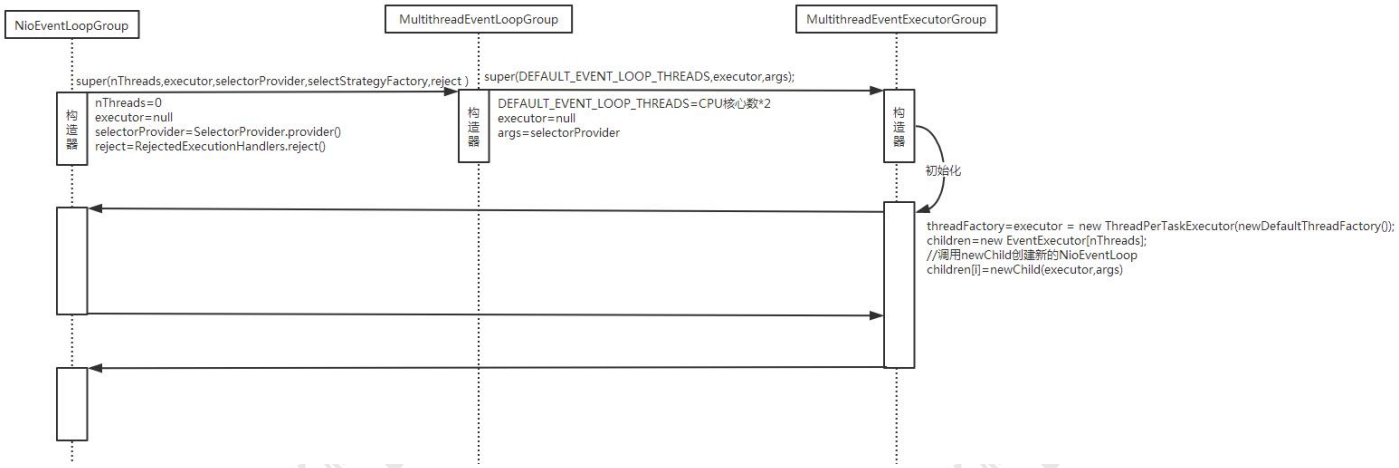
bossGroup 为主线程，而 workerGroup 中的线程是 CPU 核心数乘以 2，因此对应的到 Reactor 线程模型中，我们知道，这样设置的 NioEventLoopGroup 其实就是 Reactor 主从多线程模型。

### 7.1.3 EventLoopGroup 的实例化

首先，我们先纵览一下 EventLoopGroup 的类结构图，如下图所示：




在前面的章节中我们已经简单地介绍了一下 NioEventLoopGroup 初始化的基本过程，这里我们再回顾一下时序图：




基本步骤如下：

1、EventLoopGroup(其实是 MultithreadEventExecutorGroup)内部维护一个类为 EventExecutor children 数组，其大小是 nThreads，这样就初始化了一个线程池。

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，否则是 CPU 核数 * 2。

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

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

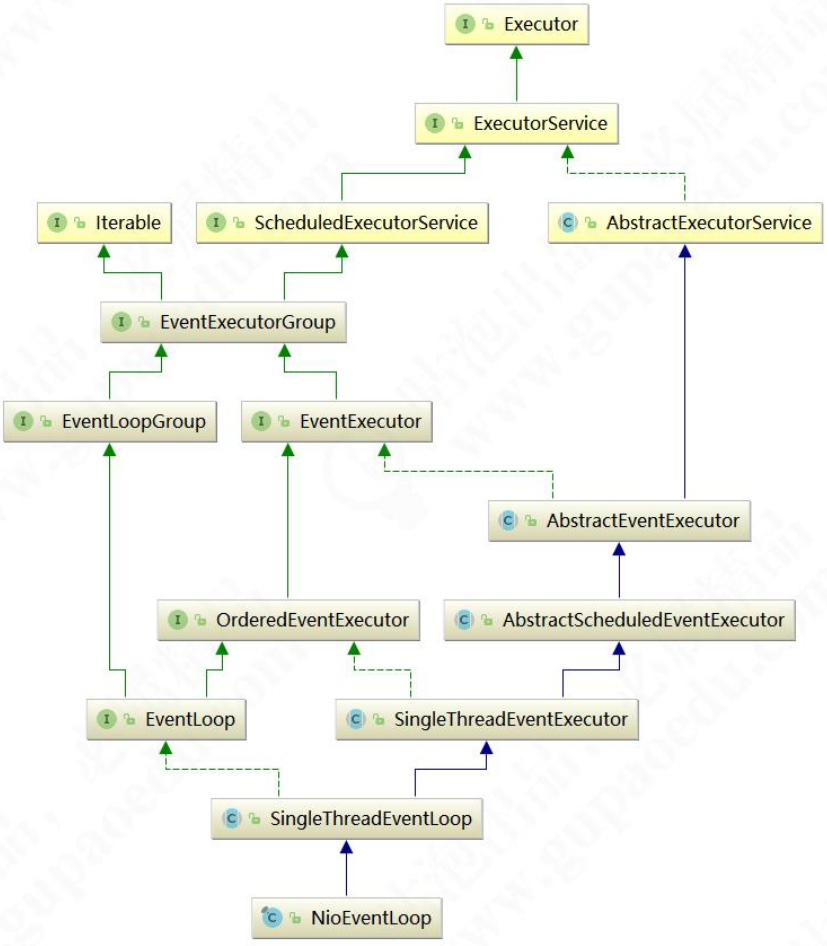
5、初始化 NioEventLoop 主要属性：

provider：在 NioEventLoopGroup 构造器中通过 SelectorProvider 的 provider()方法获取 SelectorProvider。

selector：在 NioEventLoop 构造器中调用 selector = provider.openSelector()方法获取 Selector 对象。

## 7.2 任务执行者 EventLoop

NioEventLoop 继承自 SingleThreadEventLoop，而 SingleThreadEventLoop 又继承自 SingleThreadEventExecutor。而 SingleThreadEventExecutor 是 Netty 中对本地线程的抽象，它内部有一个 Thread thread 属性，存储了一个本地 Java 线程。因此我们可以简单地认为，一个 NioEventLoop 其实就是和一个特定的线程绑定，并且在其生命周期内，绑定的线程都不会再改变。

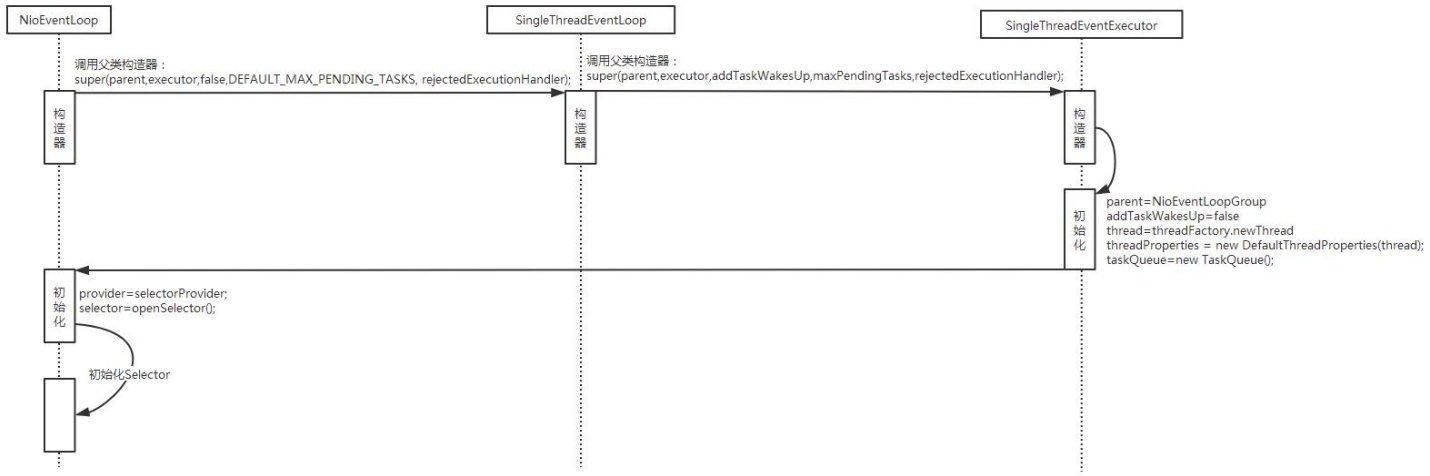


NioEventLoop 的类层次结构图还是有些复杂的，不过我们只需要关注几个重要点即可。首先来看 NioEventLoop 的继承链：NioEventLoop->SingleThreadEventLoop->SingleThreadEventExecutor->AbstractScheduledEventExecutor。在 AbstractScheduledEventExecutor 中，Netty 实现了 NioEventLoop 的 schedule 功能，即我们可以通过调用一个 NioEventLoop 实例的 schedule 方法来运行一些定时任务。而在 SingleThreadEventLoop 中，又实现了任务队列的功能 ,通过它 ,我们可以调用一个 NioEventLoop 实例的 execute()方法来向任务队列中添加一个 task,并由 NioEventLoop 进行调度执行。

通常来说，NioEventLoop 负责执行两个任务：第一个任务是作为 IO 线程，执行与 Channel 相关的 IO 操作，包括调用 Selector 等待就绪的 IO 事件、读写数据与数据的处理等；而第二个任务是作为任务队列，执行 taskQueue 中的任务，例如用户调用 eventLoop.schedule 提交的定时任务也是这个线程执行的。

### 7.2.1 NioEventLoop 的实例化过程

先简单回顾一下 EventLoop 实例化的运行时序图：




从上图可以看到，SingleThreadEventExecutor 有一个名为 thread 的 Thread 类型字段，这个字段就是与 SingleThreadEventExecutor 关联的本地线程。我们看看 thread 在哪里被赋值的：

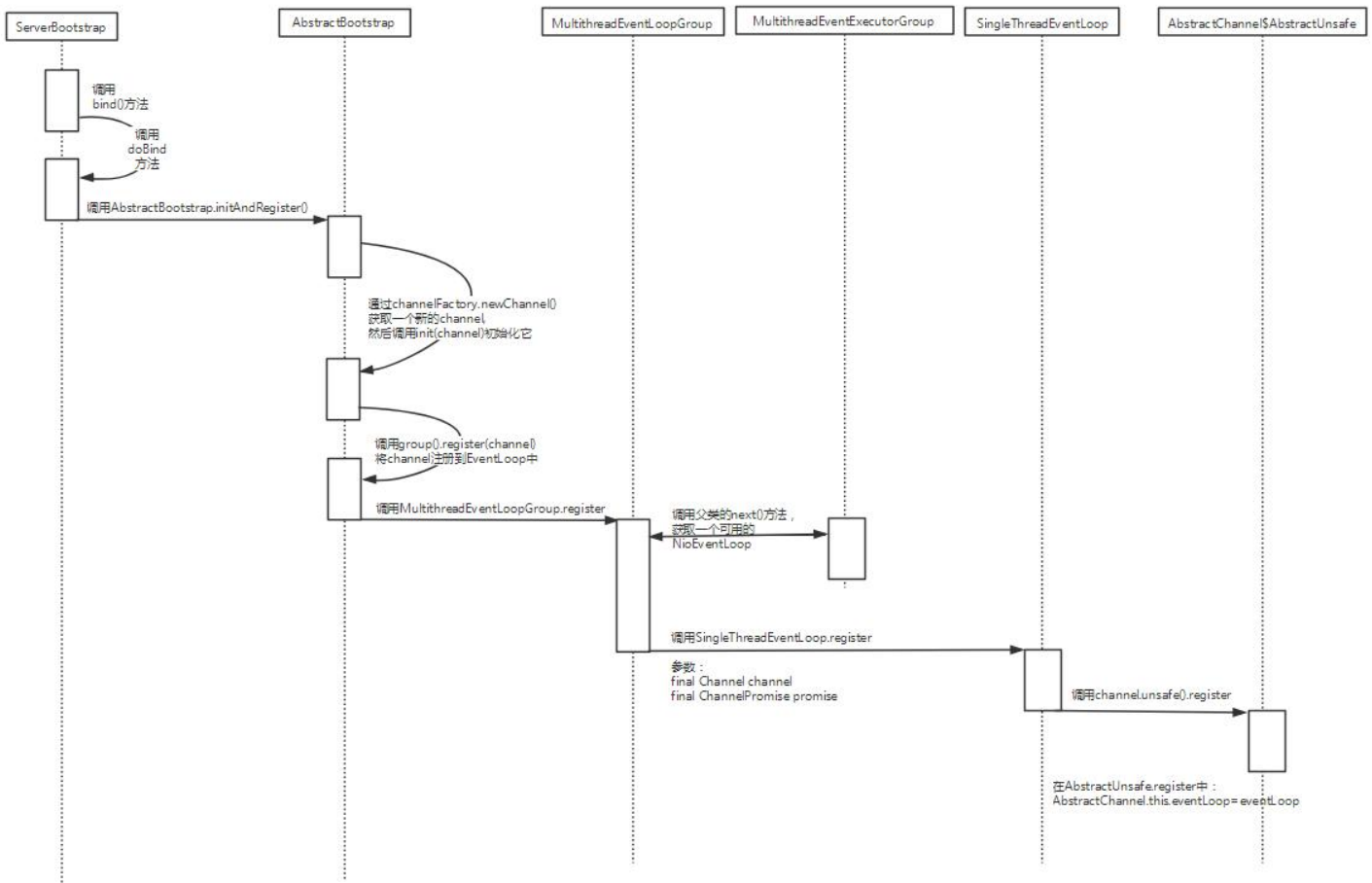
```
private void doStartThread() {
    assert thread == null;
    executor.execute(new Runnable() {
        @Override
        public void run() {
            thread = Thread.currentThread();
            boolean success = false;
```

```
            updateLastExecutionTime();
            try {
                SingleThreadEventExecutor.this.run();
                success = true;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                // 此处省略清理代码                }
        }
    });
}
```

之前的章节我们分析过 ,SingleThreadEventExecutor 启动时会调用 doStartThread()方法 ,然后调用 executor.execute()方法，将当前线程赋值给 thread。在这个线程中所做的事情主要就是调用 SingleThreadEventExecutor.this.run()方法，而因为 NioEventLoop 实现了这个方法，因此根据多态性，其实调用的是 NioEventLoop.run()方法。

### 7.2.2 EventLoop 与 Channel 的关联

在 Netty 中，每个 Channel 都有且仅有一个 EventLoop 与之关联，它们的关联过程如下：




从上图中我们可以看到，当调用 AbstractChannel$AbstractUnsafe.register()方法后，就完成了 Channel 和 EventLoop

的关联。register()方法的具体实现如下：

```
    public final void register(EventLoop eventLoop, final ChannelPromise promise) {
        // 删除条件检查

        AbstractChannel.this.eventLoop = eventLoop;

        if (eventLoop.inEventLoop()) {
            register0(promise);
        } else {
            try {
                eventLoop.execute(new Runnable() {
                    @Override
                    public void run() {
                        register0(promise);
                    }
                });
            } catch (Throwable t) {
                // 删除 catch 块内容
            }
        }
    }
```

在 AbstractChannel$AbstractUnsafe.register() 方法中，会将一个 EventLoop 赋值给 AbstractChannel 内部的 eventLoop 字段，这句代码就是完成 EventLoop 与 Channel 的关联过程。

### 7.2.3 EventLoop 的启动

在前面我们已经知道了，NioEventLoop 本身就是一个 SingleThreadEventExecutor，因此 NioEventLoop 的启动，其实就是 NioEventLoop 所绑定的本地 Java 线程的启动。

按照这个思路，我们只需要找到在哪里调用了 SingleThreadEventExecutor 中 thread 字段的 start()方法就可以知道是在哪里启动的这个线程了。从前面章节的分析中，其实我们已经清楚：thread.start() 被封装到了 SingleThreadEventExecutor.startThread()方法中，来看代码：

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

STATE_UPDATER 是 SingleThreadEventExecutor 内部维护的一个属性，它的作用是标识当前的 thread 的状态。在初始的时候，STATE_UPDATER == ST_NOT_STARTED，因此第一次调用 startThread()方法时，就会进入到 if 语句内，进而调用到 thread.start()方法。而这个关键的 startThread()方法又是在哪里调用的呢?用方法调用关系反向查找功能，

我们发现，startThread 是在 SingleThreadEventExecutor 的 execute()方法中调用的：

```
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }

    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        addTask(task);
    } else {
        startThread();  // 调用 startThread 方法、启动 EventLoop 线程
        addTask(task);
        if (isShutdown() && removeTask(task)) {
            reject();
        }
    }

    if (!addTaskWakesUp && wakesUpForTask(task)) {
        wakeup(inEventLoop);
    }
}
```

既然如此，那现在我们的工作就变为了寻找在哪里第一次调用了 SingleThreadEventExecutor 的 execute()方法。

如果细心的小伙伴可能已想到了，我们在前面章节中，我们有提到到在注册 channel 的过程中，会在 AbstractChannel$AbstractUnsafe 的 register()中调用 eventLoop.execute()方法，在 EventLoop 中进行 Channel 注册代码的执行，AbstractChannel$AbstractUnsafe 的 register()部分代码如下：

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
       // 删除判断
       AbstractChannel.this.eventLoop = eventLoop;

       if (eventLoop.inEventLoop()) {
           register0(promise);
       } else {
           try {
               eventLoop.execute(new Runnable() {
                   @Override
                   public void run() {
                       register0(promise);
                   }
               });
           } catch (Throwable t) {
                 // 删除异常处理代码
           }
       }
   }
```

很显然，一路从 Bootstrap 的 bind()方法跟踪到 AbstractChannel$AbstractUnsafe 的 register()方法，整个代码都是在主线程中运行的，因此上面的 eventLoop.inEventLoop()返回为 false，于是进入到 else 分支，在这个分支中调用了 eventLoop.execute()方法，而 NioEventLoop 没有实现 execute()方法，因此调用的是 SingleThreadEventExecutor 的

execute()方法：

```
public void execute(Runnable task) {
    // 条件判断
    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        addTask(task);
    } else {
        startThread();
        addTask(task);
        if (isShutdown() && removeTask(task)) {
            reject();
        }
    }
    if (!addTaskWakesUp && wakesUpForTask(task)) {
        wakeup(inEventLoop);
    }
}
```

我们已经分析过了，inEventLoop == false，因此执行到 else 分支，在这里就调用 startThread()方法来启动 SingleThreadEventExecutor 内部关联的 Java 本地线程了。

总结一句话：当 EventLoop 的 execute()第一次被调用时，就会触发 startThread()方法的调用，进而导致 EventLoop 所对应的 Java 本地线程启动。

我们将上一小节中的时序图补全后，就得到了 EventLoop 启动过程完整的时序图：