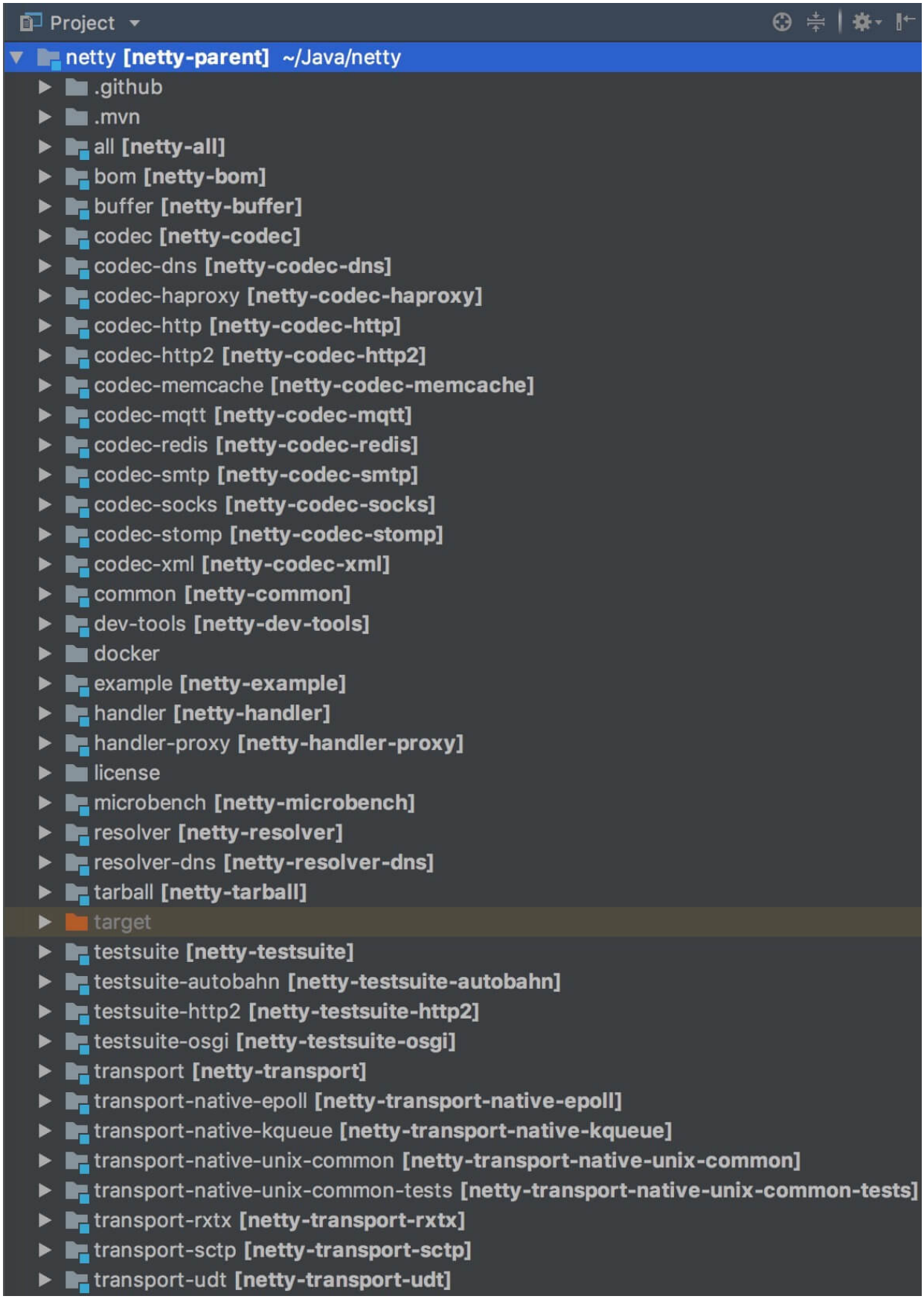
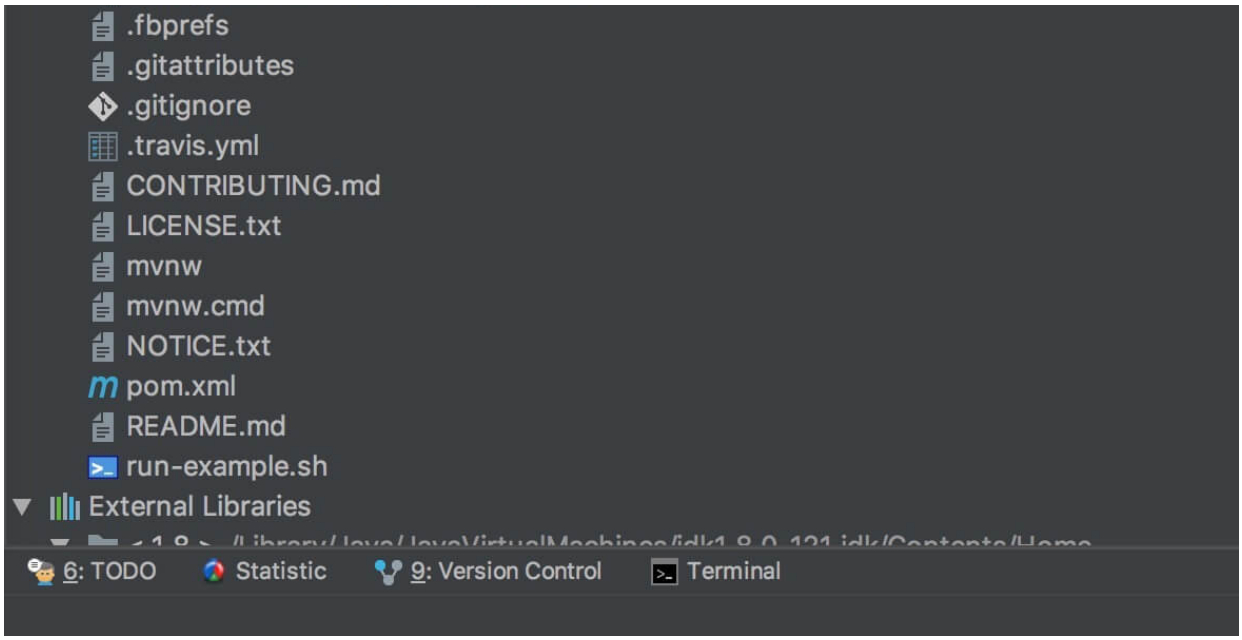
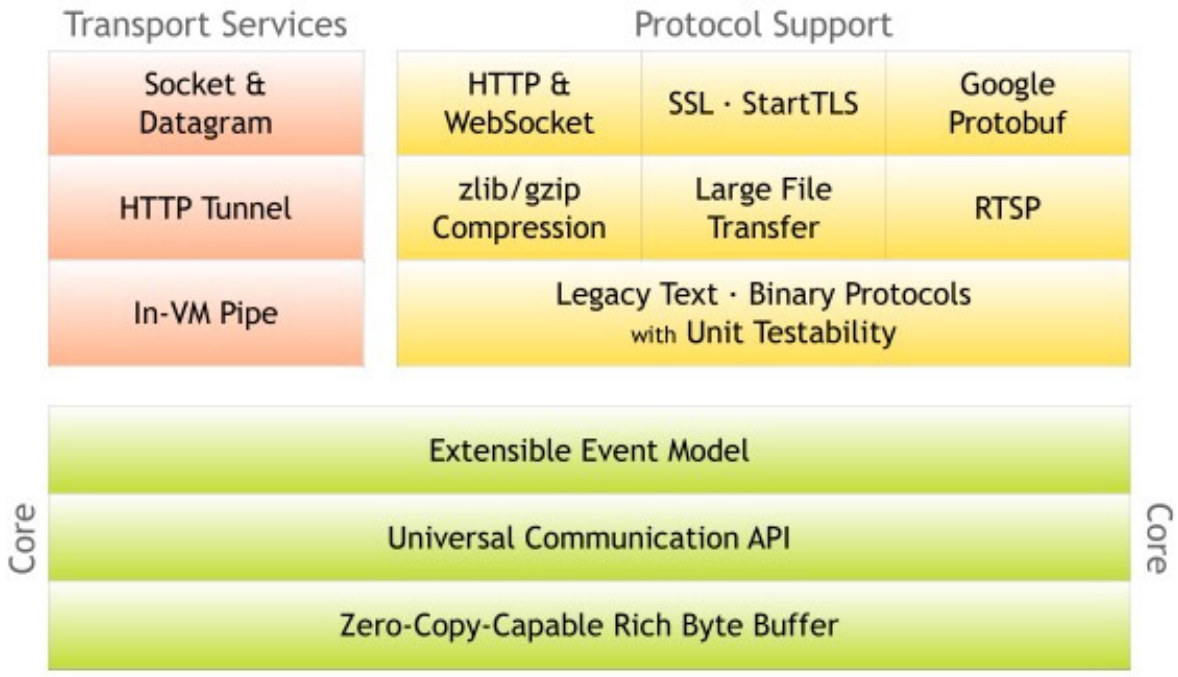
Netty源码分析之服务端启动&EventLoop

一、Netty项目结构



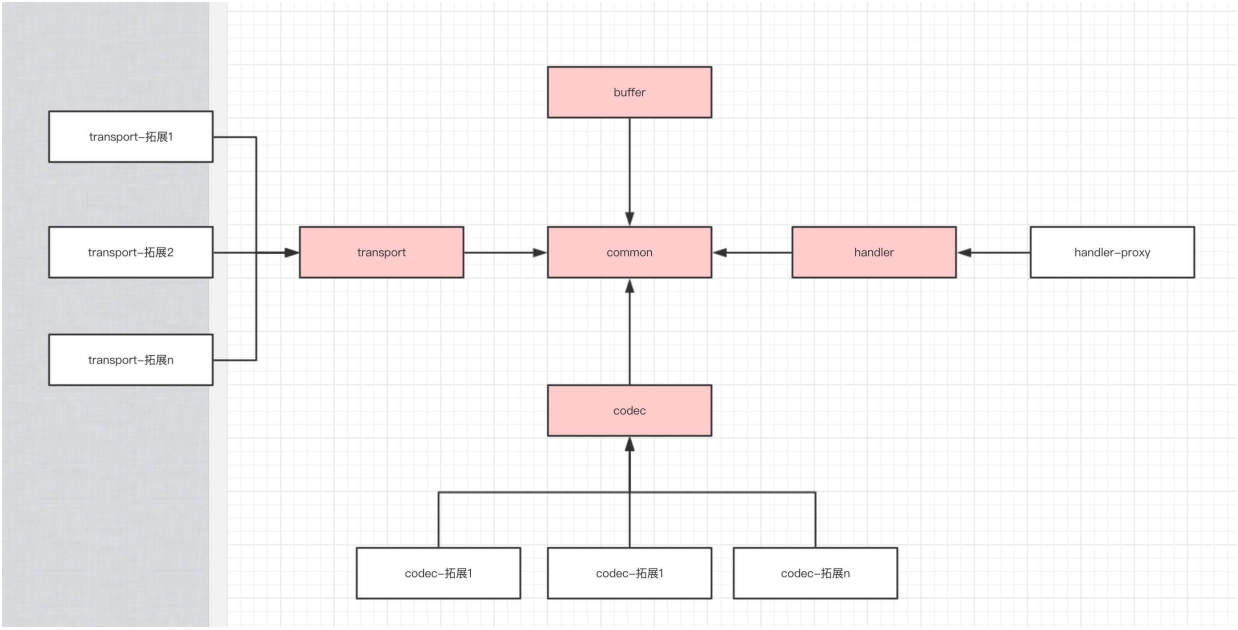


二、Netty架构图



- Core：核心部分，是底层的网络通用抽象和部分实现。
  - Extensible Event Model： 可拓展的事件模型。Netty 是基于事件模型的网络应用框架。
  - Universal Communication API： 通用的通信 API 层。Netty 定义了一套抽象的通用通信层的 API 。
  - Zero-Copy-Capable Rich Byte Buffer： 支持零拷贝特性的 Byte Buffer 实现。
- Transport Services：传输(通信)服务，具体的网络传输的定义与实现。
  - Socket & Datagram： TCP 和 UDP 的传输实现。
  - HTTP Tunnel： HTTP 通道的传输实现。
  - In-VM Piple： JVM 内部的传输实现。
- **Protocol Support**： 协议支持。Netty 对于一些通用协议的编解码实现。例如：HTTP、Redis、DNS 等等。

三、项目依赖

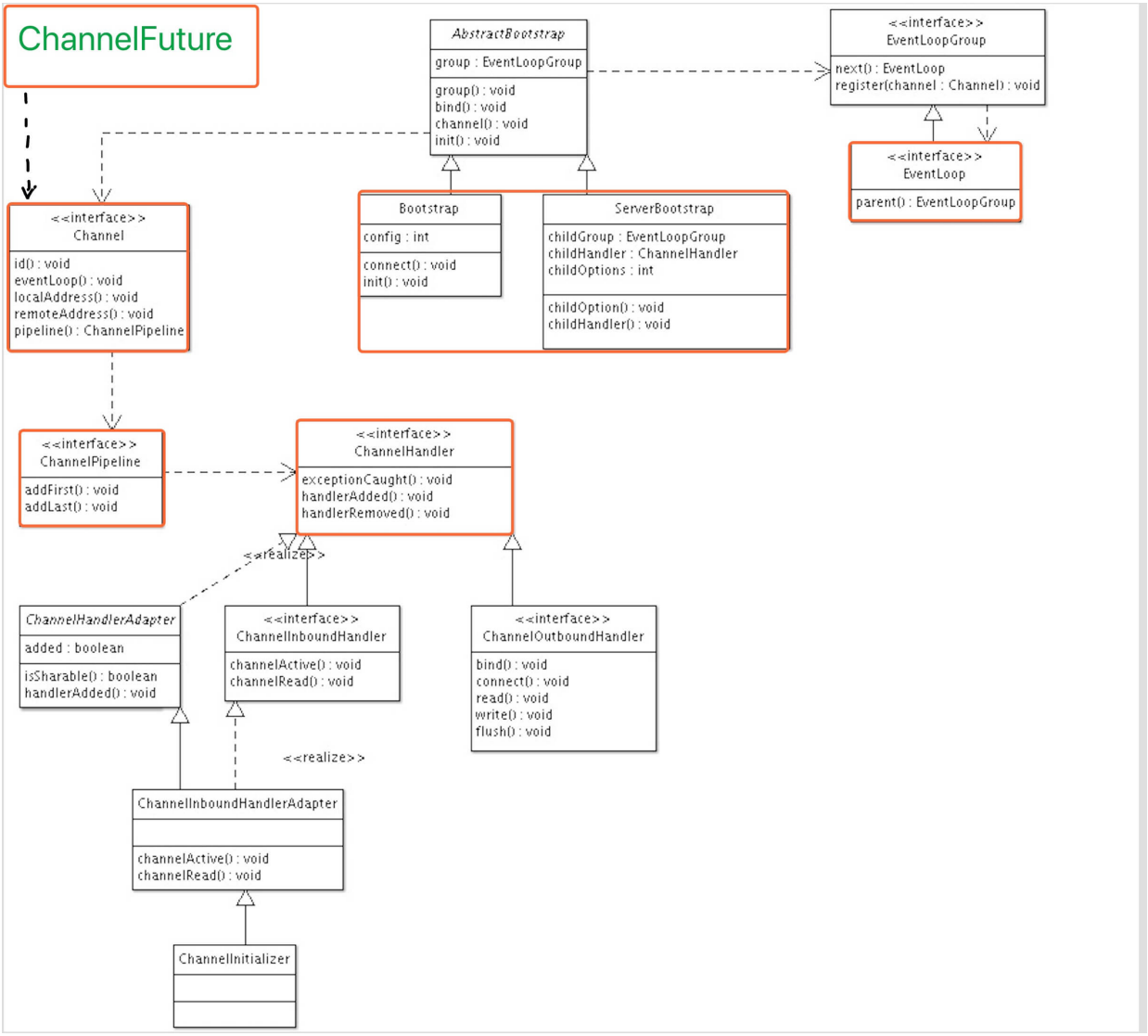


四、 Netty核心组件

Netty 有如下几个核心组件：

- Bootstrap & ServerBootstrap
- Channel
- ChannelFuture
- EventLoop & EventLoopGroup
- ChannelHandler
- ChannelPipeline

核心组件的高层类图如下：



下图是 Channel、EventLoop、Thread、EventLoopGroup 之间的关系(摘自《Netty In Action》)：

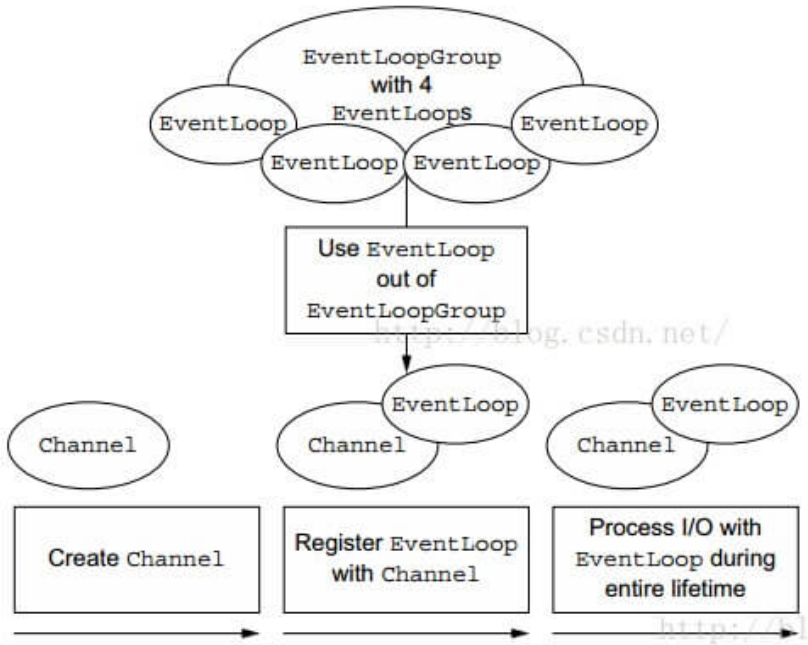


Figure 3.1 Channels, EventLoops, and EventLoopGroups

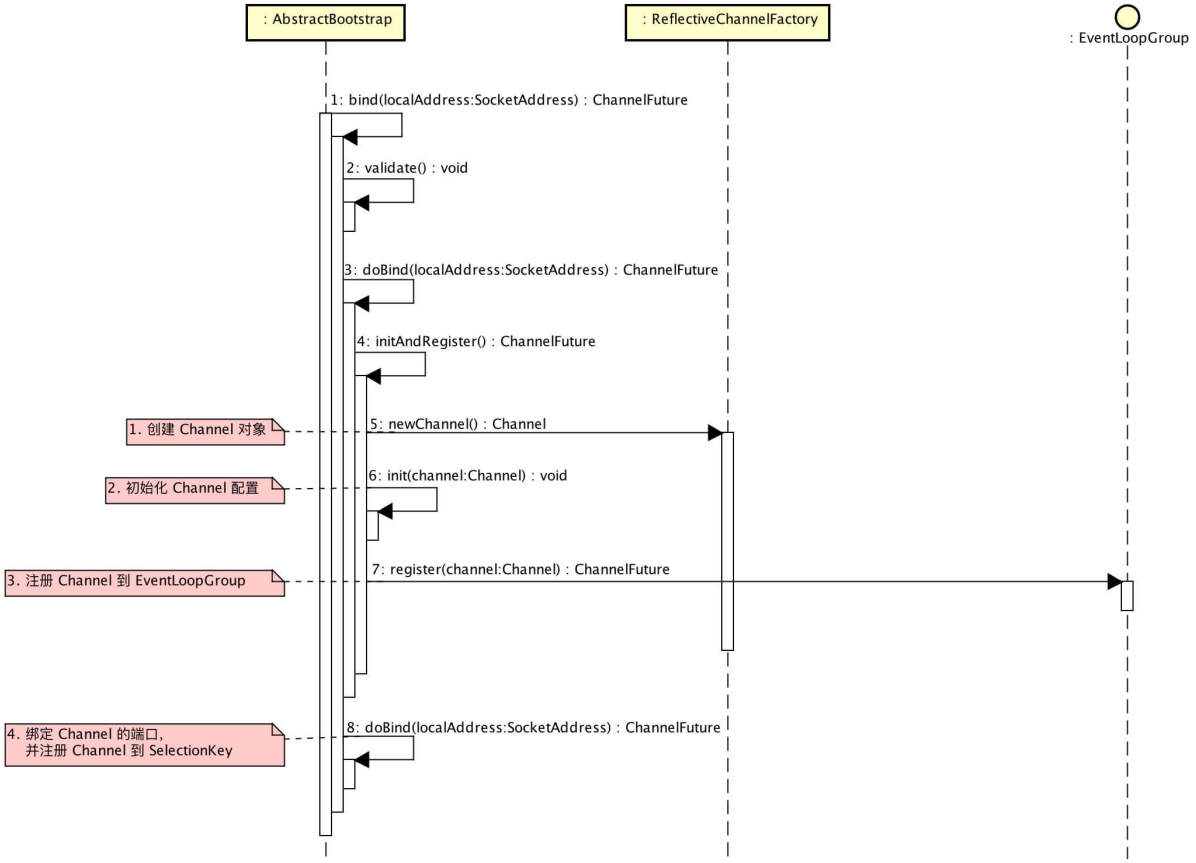
- 一个 EventLoopGroup 包含一个或多个 EventLoop ，即 EventLoopGroup : EventLoop = `1 : n` 。
- 一个 EventLoop 在它的生命周期内，只能与一个 Thread 绑定，即 EventLoop : Thread = `1 : 1` 。
- 所有有 EventLoop 处理的 I/O 事件都将在它**专有**的 Thread 上被处理，从而保证线程安全，即 Thread : EventLoop = `1 : 1` 。
- 一个 Channel 在它的生命周期内只能注册到一个 EventLoop 上，即 Channel : EventLoop = `n : 1` 。
- 一个 EventLoop 可被分配至一个或多个 Channel ，即 EventLoop : Channel = `1 : n` 。

当一个连接到达时，Netty 就会创建一个 Channel，然后从 EventLoopGroup 中分配一个 EventLoop 来给这个 Channel 绑定上，在该 Channel 的整个生命周期中都是有这个绑定的 EventLoop 来服务的。

五、Echo Demo 使用场景

EchoServer：

（一）bind：核心流程

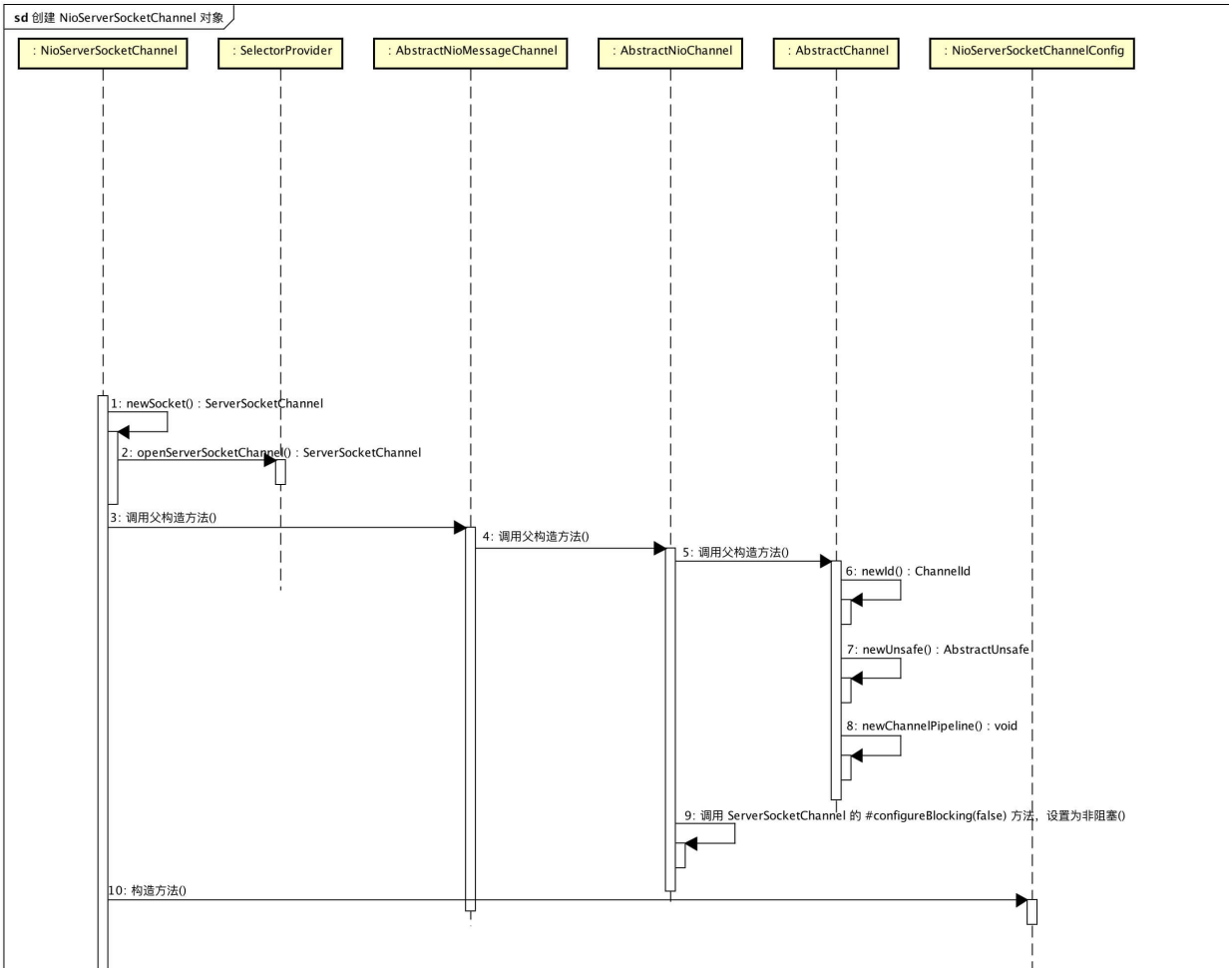


- doBind()

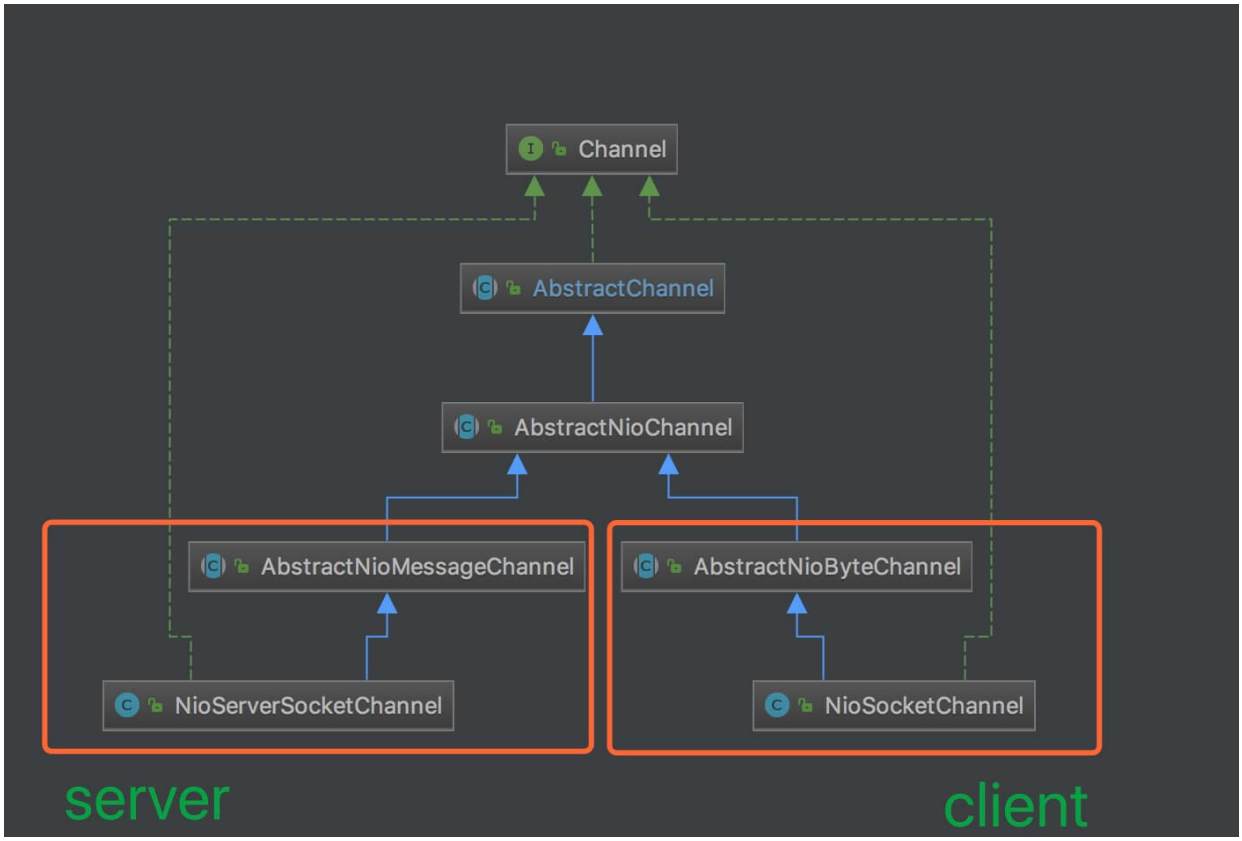
  initAndRegister
- doBind0()
- initAndRegister()

  init()
- beginRead()

（二）创建 Channel 对象



整个流程涉及到 NioServerSocketChannel 的父类们。类图如下：



1.NioServerSocketChannel

2.AbstractNioMessageChannel

3.AbstractNioChannel

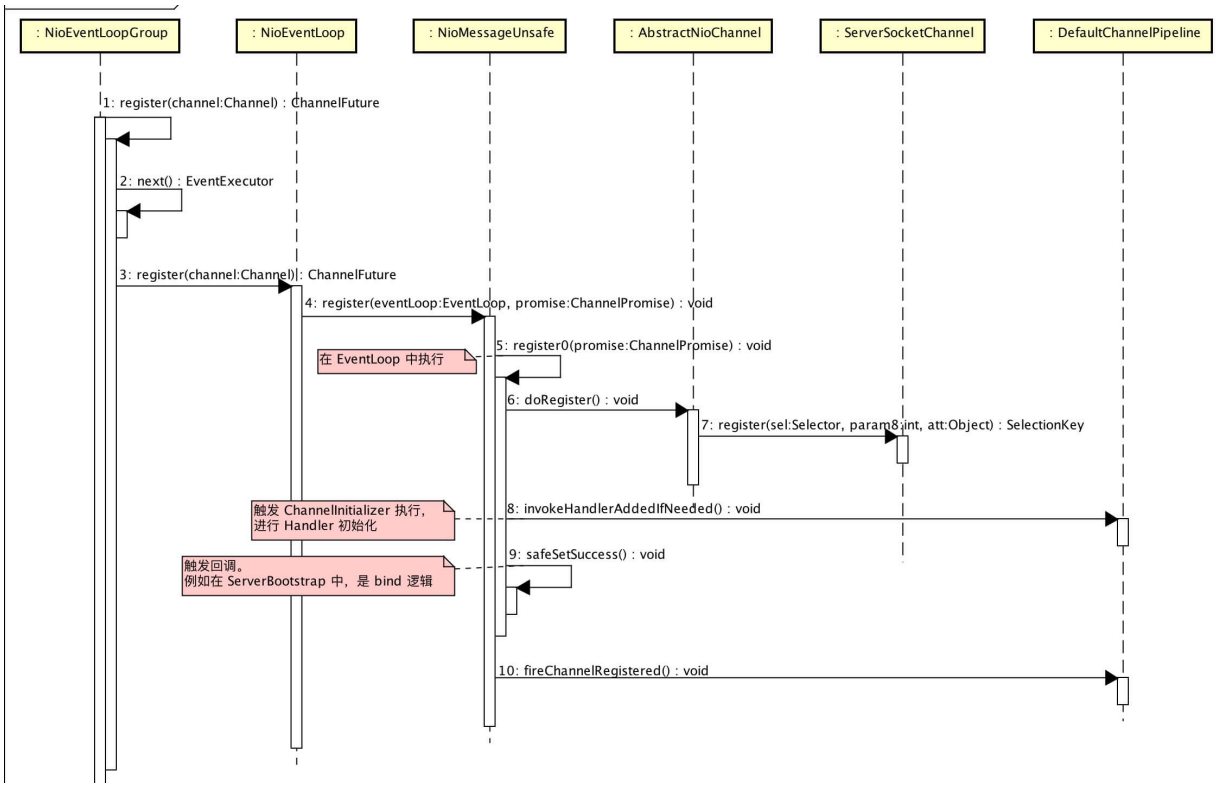
4.AbstractChannel

（三） 初始化 Channel 配置

abstract void init(Channel channel) throws Exception;

(四) 注册 Channel 到 EventLoopGroup

`EventLoopGroup#register(Channel channel)` 方法，注册 Channel 到 EventLoopGroup 中。整体流程如下：



1. register:

   AbstractUnsafe#register(EventLoop eventLoop, final ChannelPromise promise)` 方法开始校验传入的 `eventLoop` 参数非空。

   调用 `#isRegistered()` 方法

2. register0

   `#register0(ChannelPromise promise)` 方法

`SelectableChannel#register(Selector sel, int ops, Object att)` 方法，注册 Java 原生 NIO 的 Channel 对象到 Selector 对象上。但是为什么感兴趣的事件是为 **0** 呢？正常情况下，对于服务端来说，需要注册 `SelectionKey.OP_ACCEPT` 事件呢

（1）注册方式是多态的，它既可以被 NIOServerSocketChannel 用来监听客户端的连接接入，也可以注册 SocketChannel 用来监听网络读或者写操作。

（2）通过 `SelectionKey#interestOps(int ops)` 方法可以方便地修改监听操作位。所以，此处注册需要获取 SelectionKey 并给 AbstractNIOChannel 的成员变量 `selectionKey` 赋值。

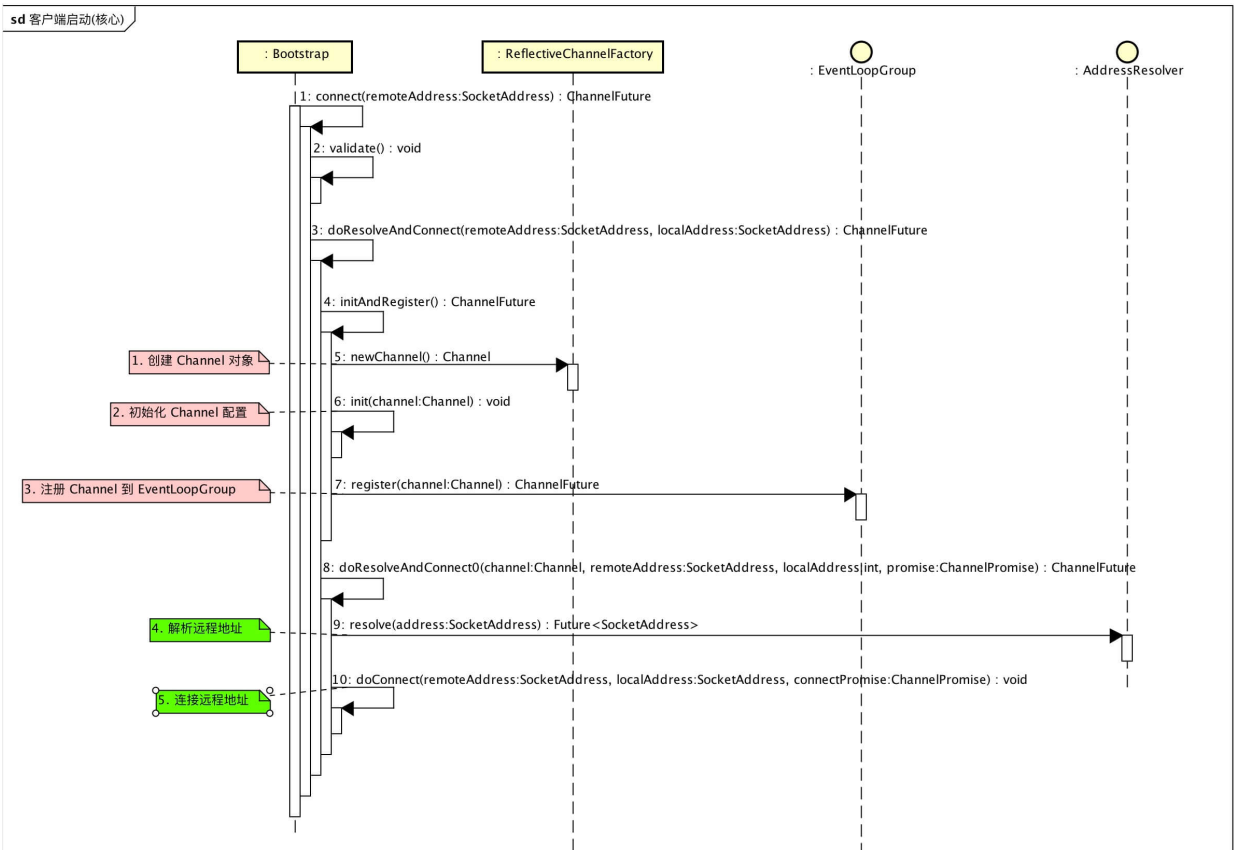
(五) ServerBootstrap

为什么要使用 ChannelInitializer 进行处理器的初始化呢？而不直接添加到 pipeline 中。例如修改为如下代码：

- 因为此时 Channel 并未注册到 EventLoop 中。如果调用 `EventLoop#execute(Runnable runnable)` 方法，会抛出 `Exception in thread "main" java.lang.IllegalStateException: channel not registered to an event loop` 异常。

EchoClient：

`#connect(...)` 方法，核心流程如下图



1.doResolveAndConnect

`#doResolveAndConnect(final SocketAddress remoteAddress, final SocketAddress localAddress)` 方法

initAndRegister:初始化并注册一个 Channel 对象。因为注册是**异步**的过程，所以返回一个 ChannelFuture 对象。

2.doResolveAndConnect0

3.doConnect

4.finishConnect:AbstractNioUnsafe#finishConnect()

doFinishConnect

fulfillConnectPromise 成功：`AbstractNioUnsafe#fulfillConnectPromise(ChannelPromise promise, Throwable cause)` 方法，通知 `connectPromise` 连接完成

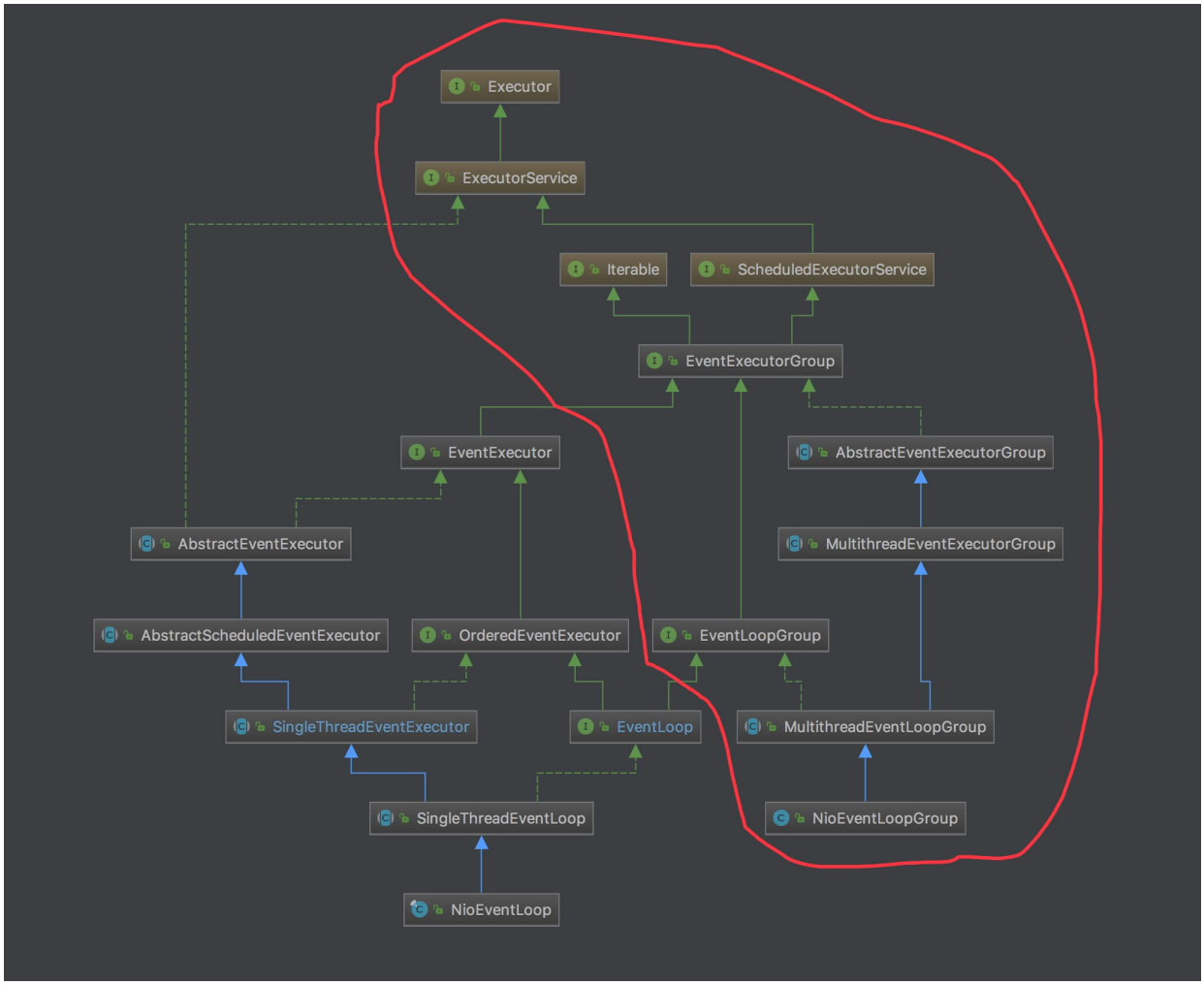
六、EventLoop

1. 理解 EventLoop 有哪些属性
2. 创建 EventLoop 的过程
3. Channel 注册到 EventLoop 的过程
4. EventLoop 的任务提交。

- Channel 为Netty 网络操作抽象类，EventLoop 负责处理注册到其上的 Channel 处理 I/O 操作，两者配合参与 I/O 操作。
- EventLoopGroup 是一个 EventLoop 的分组，它可以获取到一个或者多个 EventLoop 对象，因此它提供了迭代出 EventLoop 对象的方法。

(一)、类结构图

EventLoopGroup 的整体类结构如下图：



（二)、 EventExecutorGroup

(三)、AbstractEventExecutorGroup

核心API：submit、schedule、execute、invokeAll、invokeAny、shutdown

(四)、MultithreadEventExecutorGroup

基于多线程的 EventExecutor (事件执行器)的分组抽象类

(五)、ThreadPerTaskExecutor

`io.netty.util.concurrent.ThreadPerTaskExecutor` ，实现 Executor 接口，每个任务一个线程的执行器实现类

(六)、EventExecutorChooserFactory&DefaultEventExecutorChooserFactory

`io.netty.util.concurrent.EventExecutorChooserFactory` ，EventExecutorChooser 工厂接口

`#newChooser(EventExecutor[] executors)` 方法，调用 `#isPowerOfTwo(int val)` 方法，判断 EventExecutor 数组的大小是否为 2 的幂次方。

```
#isPowerOfTwo(int val)方法，为什么(val & -val) == val
```

可以判断数字是否为 2 的幂次方呢?

- 我们以 8 来举个例子。
  - 8 的二进制为 `1000` 。
  - -8 的二进制使用补码表示。所以，先求反生成反码为 `0111` ，然后加一生成补码为 `1000` 。
  - 8 和 -8 &操作后，还是 8 。
  - 实际上，以 2 为幂次方的数字，都是最高位为 1 ，剩余位为 0 ，所以对应的负数，求完补码还是自己。

(七)、PowerOfTwoEventExecutorChooser

```
AbstractScheduledEventExecutor executors[idx.getAndIncrement() &
executors.length - 1]
```

实现比较巧妙，通过 idx 自增，并使用【EventExecutor 数组的大小 - 1】进行进行 &操作。

- 因为 `-` (二元操作符) 的计算优先级高于 `&` (一元操作符) 。
- 因为 EventExecutor 数组的大小是以 2 为幂次方的数字，那么减一后，除了最高位是 0 ，剩余位都为 1 (例如 8 减一后等于 7 ，而 7 的二进制为 0111 。)，那么无论 `idx` 无论如何递增，再进行 `&` 并操作，都不会超过 EventExecutor 数组的大小。并且，还能保证顺序递增。

```
DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
NettyRuntime.availableProcessors() * 2));
```

- 为什么会 * 2 呢？因为目前 CPU 基本都是超线程，**一个 CPU 可对应 2 个线程**。

(八)、 NioEventLoopGroup

`io.netty.channel.nio.NioEventLoopGroup`，继承 MultithreadEventLoopGroup 抽象类，NioEventLoop 的分组实现类。

setIoRatio(): `#setIoRatio(int ioRatio)` 方法，设置所有 EventLoop 的 IO 任务占用执行时间的比例

rebuildSelectors(): 因为 JDK 有 epoll 100% CPU Bug 。实际上，NioEventLoop 当触发该 Bug 时，也会自动调用 NioEventLoop#rebuildSelector() 方法，进行重建 Selector 对象，以修复该问题。

(九)、 EventExecutor&OrderedEventExecutor

- OrderedEventExecutor：没有定义任何方法，仅仅是一个标记接口，表示该执行器会有序 / 串行的方式执行。

(十)、 AbstractEventExecutor

inEventLoop()

`#inEventLoop()` 方法，判断当前线程是否在 EventLoop 线程中

(十一)、AbstractScheduledEventExecutor&SingleThreadEventExecutor

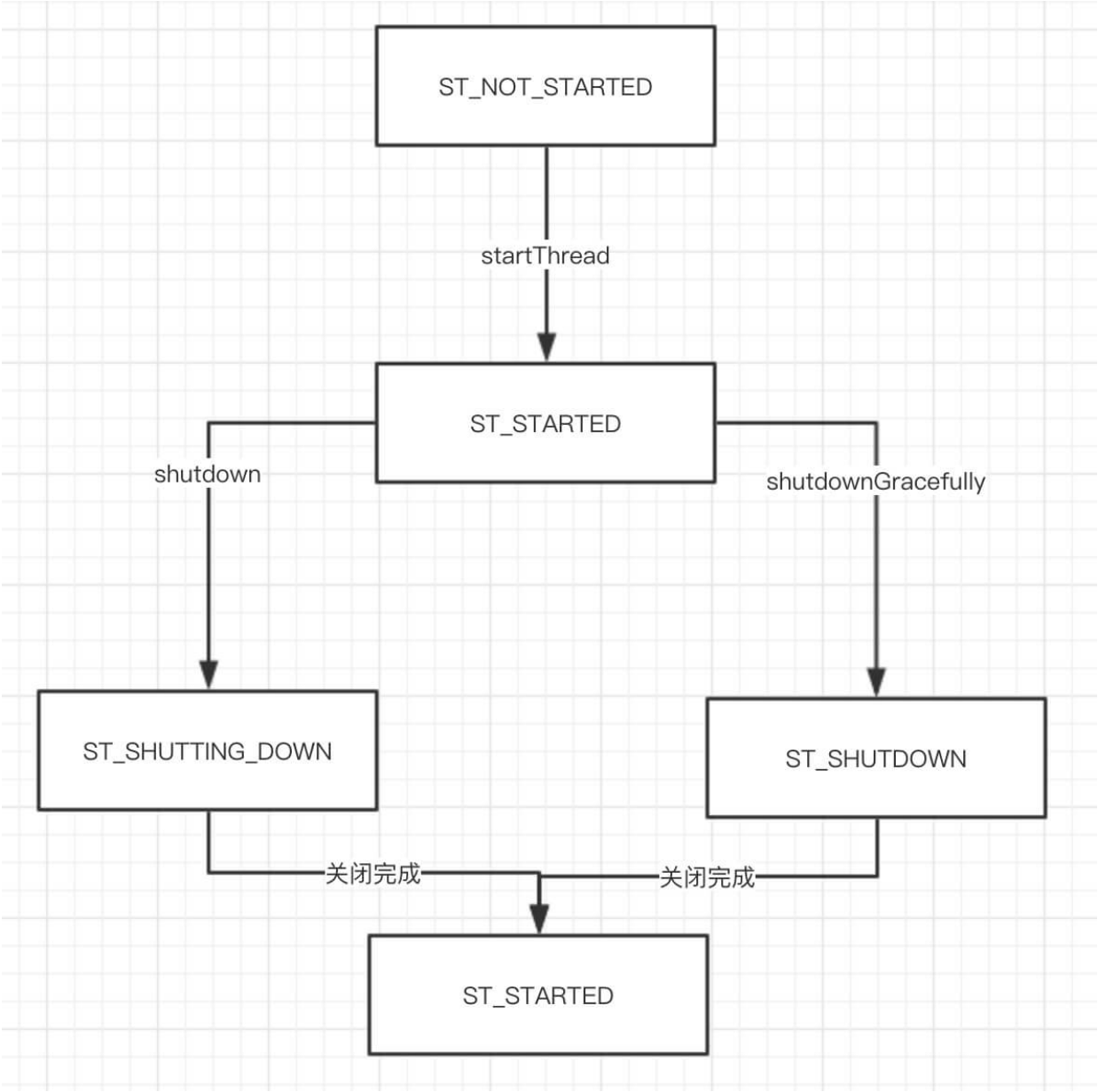
`AbstractScheduledEventExecutor：io.netty.util.concurrent.AbstractScheduledEventExecutor`，继承 AbstractEventExecutor 抽象类，**支持定时任务**的 EventExecutor 的抽象类

SingleThreadEventExecutor:基于单线程的 EventExecutor 抽象类,**即一个 EventExecutor 对应一个线程**。

state:

```

```

状态变更流程如下图：



(十二)、execute

`#execute(Runnable task)` 方法，执行一个任务。

```
1: @Override
 2: public void execute(Runnable task) {
 3:     if (task == null) {
 4:         throw new NullPointerException("task");
 5:     }
 6:
 7:     // 获得当前是否在 EventLoop 的线程中
 8:     boolean inEventLoop = inEventLoop();
 9:     // 添加到任务队列
10:     addTask(task);
11:     if (!inEventLoop) {
12:         // 创建线程
13:         startThread();
```

```
14:          // 若已经关闭，移除任务，并进行拒绝
15:          if (isShutdown() && removeTask(task)) {
16:              reject();
17:          }
18:      }
19:
20:      // 唤醒线程
21:      if (!addTaskWakesUp && wakesUpForTask(task)) {
22:          wakeup(inEventLoop);
23:      }
24: }
```

!addTaskWakesUp表示“添加任务时，是否唤醒线程”？！但是，怎么使用 取反了。这样反倒变成了，“添加任务时，是否【不】唤醒线程”。具体的原因是为什么呢?

真正的意思是，“添加任务后，任务是否会自动导致线程唤醒”。为什么呢?

- 对于 Nio 使用的 NioEventLoop ，它的线程执行任务是基于 Selector 监听感兴趣的事件，所以当任务添加到 `taskQueue` 队列中时，线程是无感知的，所以需要调用 `#wakeup(boolean inEventLoop)` 方法，进行**主动**的唤醒。
- 对于 Oio 使用的 ThreadPerChannelEventLoop ，它的线程执行是基于 `taskQueue` 队列监听(**阻塞拉取**)事件和任务，所以当任务添加到 `taskQueue` 队列中时，线程是可感知的，相当于说，进行**被动**的唤醒。

(十三)、startThread

`#startThread()` 方法，启动 EventLoop **独占**的线程，即 `thread` 属性。

(十四)、EventLoop 运行之空轮训BUG

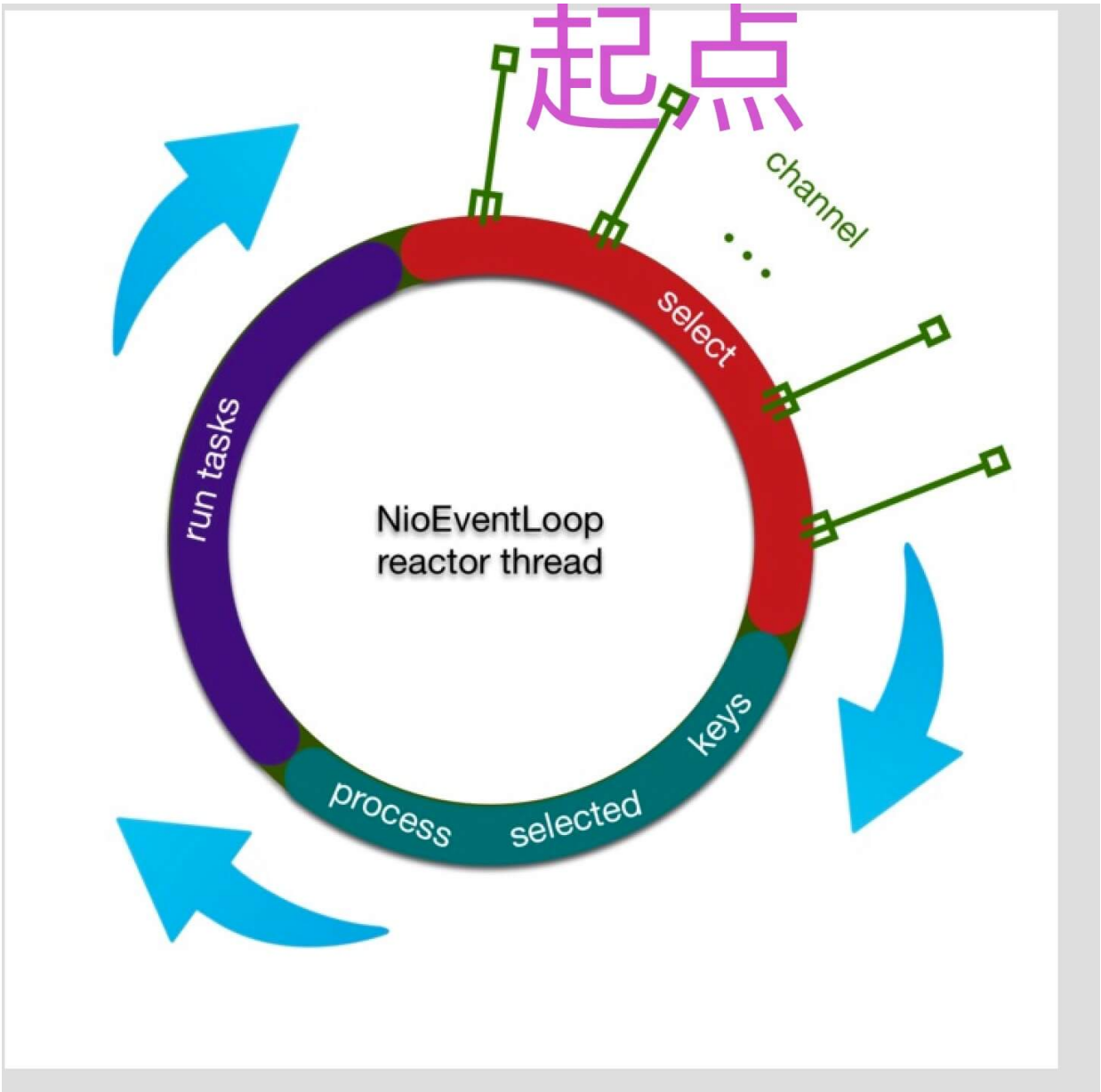
```
/**

  */
  private static final int CLEANUP_INTERVAL = 256; // XXX Hard-coded value,
but won't need customization.

/**

- 是否禁用 SelectionKey 的优化，默认开启
  */
  private static final boolean DISABLE_KEYSET_OPTIMIZATION =
SystemPropertyUtil.getBoolean("io.netty.noKeySetOptimization", false);

/**

- 少于该 N 值，不开启空轮询重建新的 Selector 对象的功能
  */
  private static final int MIN_PREMATURE_SELECTOR_RETURNS = 3;
```

```
  /**
- NIO Selector 空轮询该 N 次后，重建新的 Selector 对象
  */
  private static final int SELECTOR_AUTO_REBUILD_THRESHOLD;

static {
   // 解决 Selector#open() 方法 // <1>
    final String key = "sun.nio.ch.bugLevel";
    final String buglevel = SystemPropertyUtil.get(key);
    if (buglevel == null) {
        try {
           AccessController.doPrivileged(new PrivilegedAction<Void>() {
                @Override
                public Void run() {
                    System.setProperty(key, "");
                    return null;
                }
            });
        } catch (final SecurityException e) {
            logger.debug("Unable to get/set System Property: " + key, e);
        }
   }
// 初始化
int selectorAutoRebuildThreshold =
SystemPropertyUtil.getInt("io.netty.selectorAutoRebuildThreshold", 512);
if (selectorAutoRebuildThreshold < MIN_PREMATURE_SELECTOR_RETURNS) {
    selectorAutoRebuildThreshold = 0;
}
SELECTOR_AUTO_REBUILD_THRESHOLD = selectorAutoRebuildThreshold;

if (logger.isDebugEnabled()) {
    logger.debug("-Dio.netty.noKeySetOptimization: {}",
DISABLE_KEYSET_OPTIMIZATION);
    logger.debug("-Dio.netty.selectorAutoRebuildThreshold: {}",
SELECTOR_AUTO_REBUILD_THRESHOLD);
}
```


}
```

- `CLEANUP_INTERVAL` 属性，TODO 1007 NioEventLoop cancel
- `DISABLE_KEYSET_OPTIMIZATION` 属性，是否禁用 SelectionKey 的优化，默认开启。

  > SELECTOR_AUTO_REBUILD_THRESHOLD属性，NIO Selector 空轮询该 N 次后，重建新的 Selector 对象，用以解决 JDK NIO 的 epoll 空轮询 Bug 。

  - `MIN_PREMATURE_SELECTOR_RETURNS` 属性，少于该 N 值，不开启空轮询重建新的 Selector 对象的功能。
```

- `<1>` 处，解决 `Selector#open()` 方法，发生 NullPointException 异常。
- `<2>` 处，初始化 `SELECTOR_AUTO_REBUILD_THRESHOLD` 属性。默认 512 。

run




```
 1: @Override
 2: protected void run() {
 3:     for (;;) {
 4:         try {
 5:             switch (selectStrategy.calculateStrategy(selectNowSupplier,
hasTasks())) {
 6:                 case SelectStrategy.CONTINUE: // 默认实现下，不存在这个情
况。
 7:                     continue;
 8:                 case SelectStrategy.SELECT:
```

```
 9:                     // 重置 wakenUp 标记为 false
10:                     // 选择( 查询 )任务
11:                     select(wakenUp.getAndSet(false));
12:
13:                     // 'wakenUp.compareAndSet(false, true)' is always
evaluated
14:                     // before calling 'selector.wakeup()' to reduce the
wake-up
15:                     // overhead. (Selector.wakeup() is an expensive
operation.)
16:                     //
17:                     // However, there is a race condition in this
approach.
18:                     // The race condition is triggered when 'wakenUp'
is set to
19:                     // true too early.
20:                     //
21:                     // 'wakenUp' is set to true too early if:
22:                     // 1) Selector is waken up between
'wakenUp.set(false)' and
23:                     //    'selector.select(...)'. (BAD)
24:                     // 2) Selector is waken up between
'selector.select(...)' and
25:                     //    'if (wakenUp.get()) { ... }'. (OK)
26:                     //
27:                     // In the first case, 'wakenUp' is set to true and
the
28:                     // following 'selector.select(...)' will wake up
immediately.
29:                     // Until 'wakenUp' is set to false again in the
next round,
30:                     // 'wakenUp.compareAndSet(false, true)' will fail,
and therefore
31:                     // any attempt to wake up the Selector will fail,
too, causing
32:                     // the following 'selector.select(...)' call to
block
33:                     // unnecessarily.
34:                     //
35:                     // To fix this problem, we wake up the selector
again if wakenUp
36:                     // is true immediately after selector.select(...).
37:                     // It is inefficient in that it wakes up the
selector for both
38:                     // the first case (BAD - wake-up required) and the
second case
39:                     // (OK - no wake-up required).
40:
41:                     // 唤醒。原因，见上面中文注释
```

```
42:                     if (wakenUp.get()) {
43:                         selector.wakeup();
44:                     }
45:                     // fall through
46:                 default:
47:             }
48:
49:             // TODO 1007 NioEventLoop cancel 方法
50:             cancelledKeys = 0;
51:             needsToSelectAgain = false;
52:
53:             final int ioRatio = this.ioRatio;
54:             if (ioRatio == 100) {
55:                 try {
56:                     // 处理 Channel 感兴趣的就绪 IO 事件
57:                     processSelectedKeys();
58:                 } finally {
59:                     // 运行所有普通任务和定时任务，不限制时间
60:                     // Ensure we always run tasks.
61:                     runAllTasks();
62:                 }
63:             } else {
64:                 final long ioStartTime = System.nanoTime();
65:                 try {
66:                     // 处理 Channel 感兴趣的就绪 IO 事件
67:                     processSelectedKeys();
68:                 } finally {
69:                     // 运行所有普通任务和定时任务，限制时间
70:                     // Ensure we always run tasks.
71:                     final long ioTime = System.nanoTime() -
ioStartTime;
72:                     runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
73:                 }
74:             }
75:         } catch (Throwable t) {
76:             handleLoopException(t);
77:         }
78:         // TODO 1006 EventLoop 优雅关闭
79:         // Always handle shutdown even if the loop processing threw an
exception.
80:         try {
81:             if (isShuttingDown()) {
82:                 closeAll();
83:                 if (confirmShutdown()) {
84:                     return;
85:                 }
86:             }
87:         } catch (Throwable t) {
88:             handleLoopException(t);
```

```
89:          }
90:      }
91: }
```