

# Netty 源码分析之EventLoop

## 简述

这一节大家一起学习一下 Netty 的 EventLoop 的底层原理, 让大家对 Netty 的线程模型有更加深入的了解.

## NioEventLoopGroup

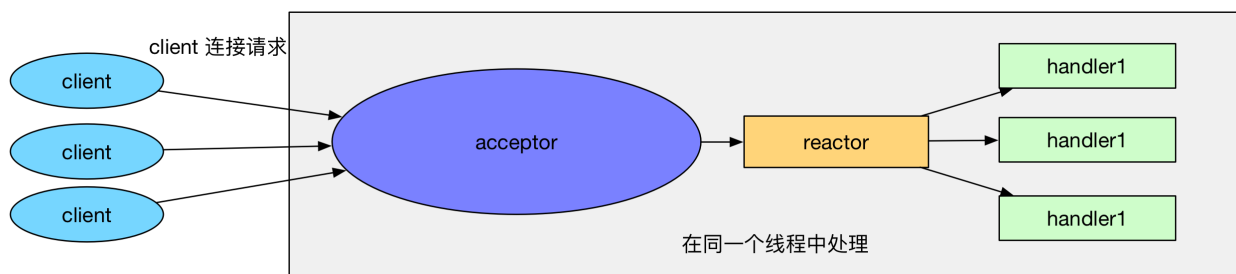
一个 Netty 程序启动时, 至少要指定一个 EventLoopGroup(如果使用到的是 NIO, 那么通常是 NioEventLoopGroup), 那么这个 NioEventLoopGroup 在 Netty 中到底扮演着什么角色呢? 我们知道, Netty 是 Reactor 模型的一个实现, 那么首先从 Reactor 的线程模型开始吧.

## 关于 Reactor 的线程模型

首先我们来看一下 Reactor 的线程模型. 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**



所谓单线程, 即 acceptor 处理和 handler 处理都在一个线程中处理. 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 模型用的比较少.

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:

Reactor 多线程模型 有如下特点:

- 有专门一个线程, 即 Acceptor 线程用于监听客户端的 TCP 连接请求.
- 客户端连接的 IO 操作都是由一个特定的 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.
- 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, 但是我们要考虑一下如下情况: 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, 进行一些权限的检查, 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。它的线程模型如下

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。

## NioEventLoopGroup 与 Reactor 线程模型的对应

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。

### 单线程模型

来看一下下面的例子:

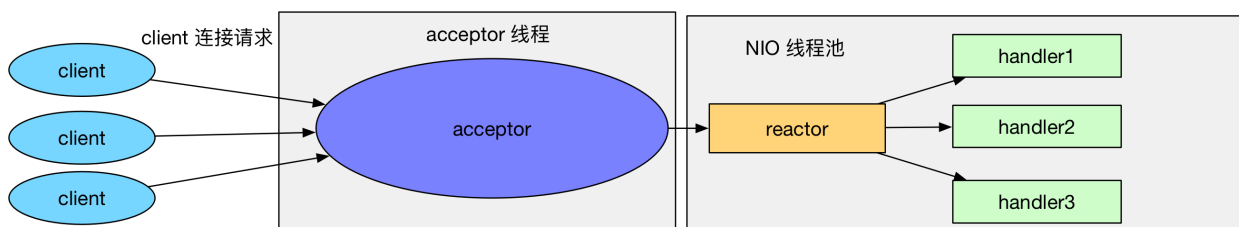
```
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup)
  .channel(NioServerSocketChannel.class)
  ...
```

注意, 我们实例化了一个 NioEventLoopGroup, 构造器参数是1, 表示 NioEventLoopGroup 的线程池大小是1。然后接着我们调用 **b.group(bossGroup)** 设置了服务器端的 EventLoopGroup。有些朋友可能会有疑惑: 我记得在启动服务器端的 Netty 程序时, 是需要设置 bossGroup 和 workerGroup 的, 为什么这里就只有一个 bossGroup? 其实很简单, ServerBootstrap 重写了 group 方法:

```
@Override
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。这时候呢, 因为 bossGroup 和 workerGroup 就是同一个 NioEventLoopGroup, 并且这个 NioEventLoopGroup 只有一个线程, 这样就会导致 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 单线程模型**。

### 多线程模型



同理, 再来看一下下面的例子:

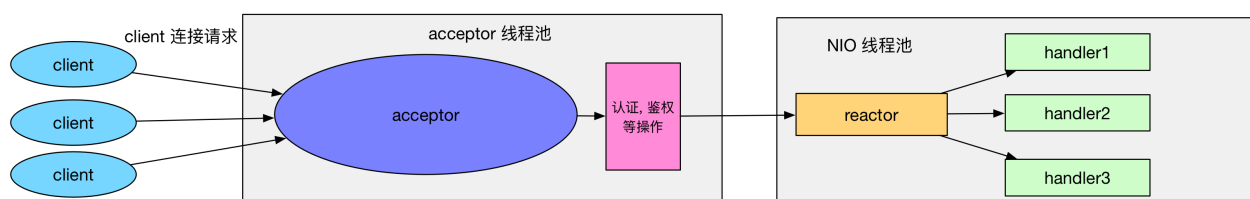
```

EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup)
  .channel(NioServerSocketChannel.class)
  ...

```

bossGroup 中只有一个线程, 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 多线程模型**.

## 主从多线程模型



```

EventLoopGroup bossGroup = new NioEventLoopGroup(4);
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup)
  .channel(NioServerSocketChannel.class)
  ...

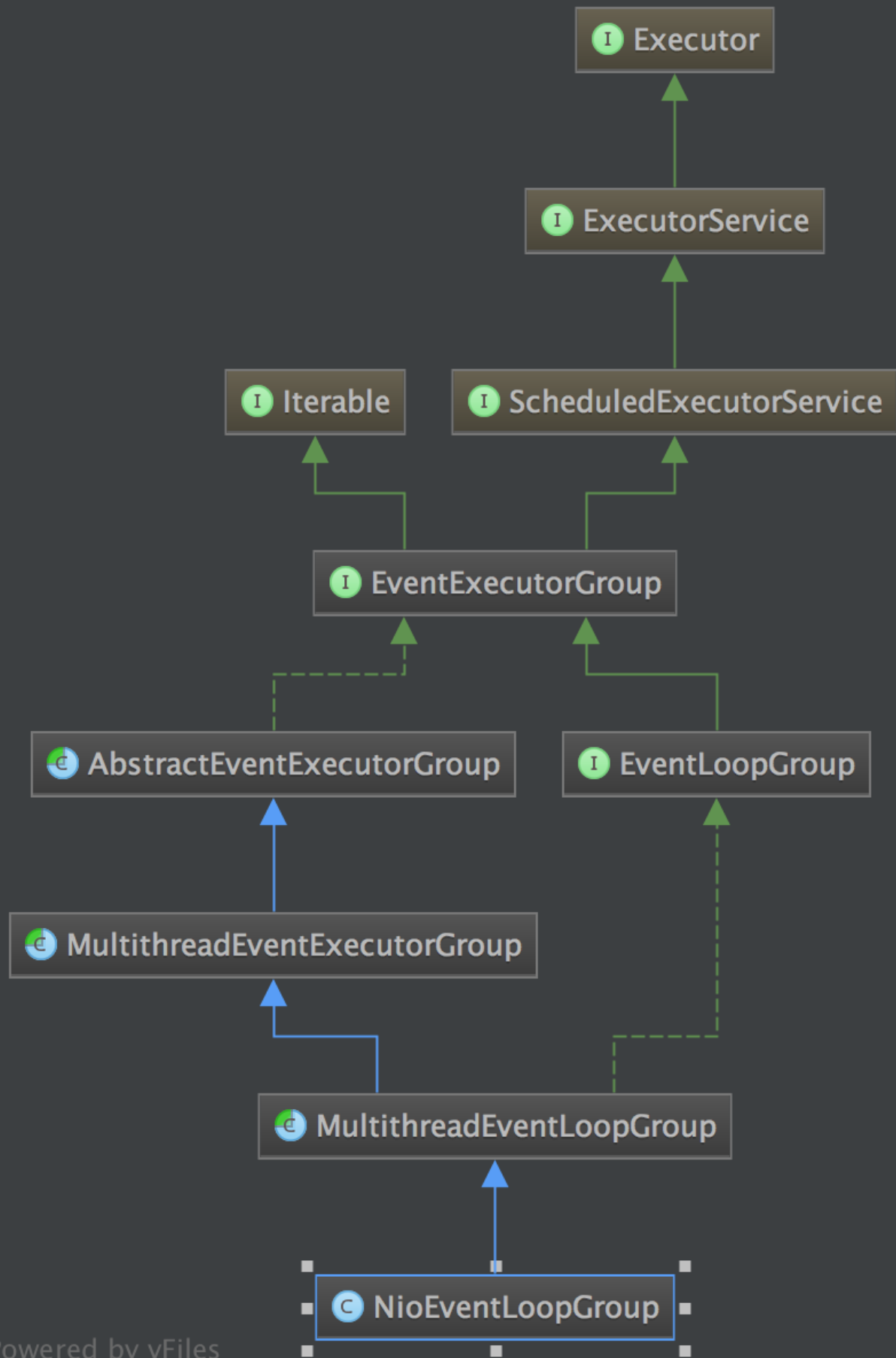
```

bossGroup 线程池中的线程数我们设置为4, 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**.

Netty 的服务器端的 acceptor 阶段, 没有使用到多线程, 因此上面的 **主从多线程模型** 在 Netty 的服务器端是不存在的.

服务器端的 ServerSocketChannel 只绑定到了 bossGroup 中的一个线程, 因此在调用 Java NIO 的 Selector.select 处理客户端的连接请求时, 实际上是在一个线程中的, 所以对只有一个服务的应用来说, bossGroup 设置多个线程是没有什么作用的, 反而还会造成资源浪费.

## NioEventLoopGroup 类层次结构



## NioEventLoopGroup 实例化过程

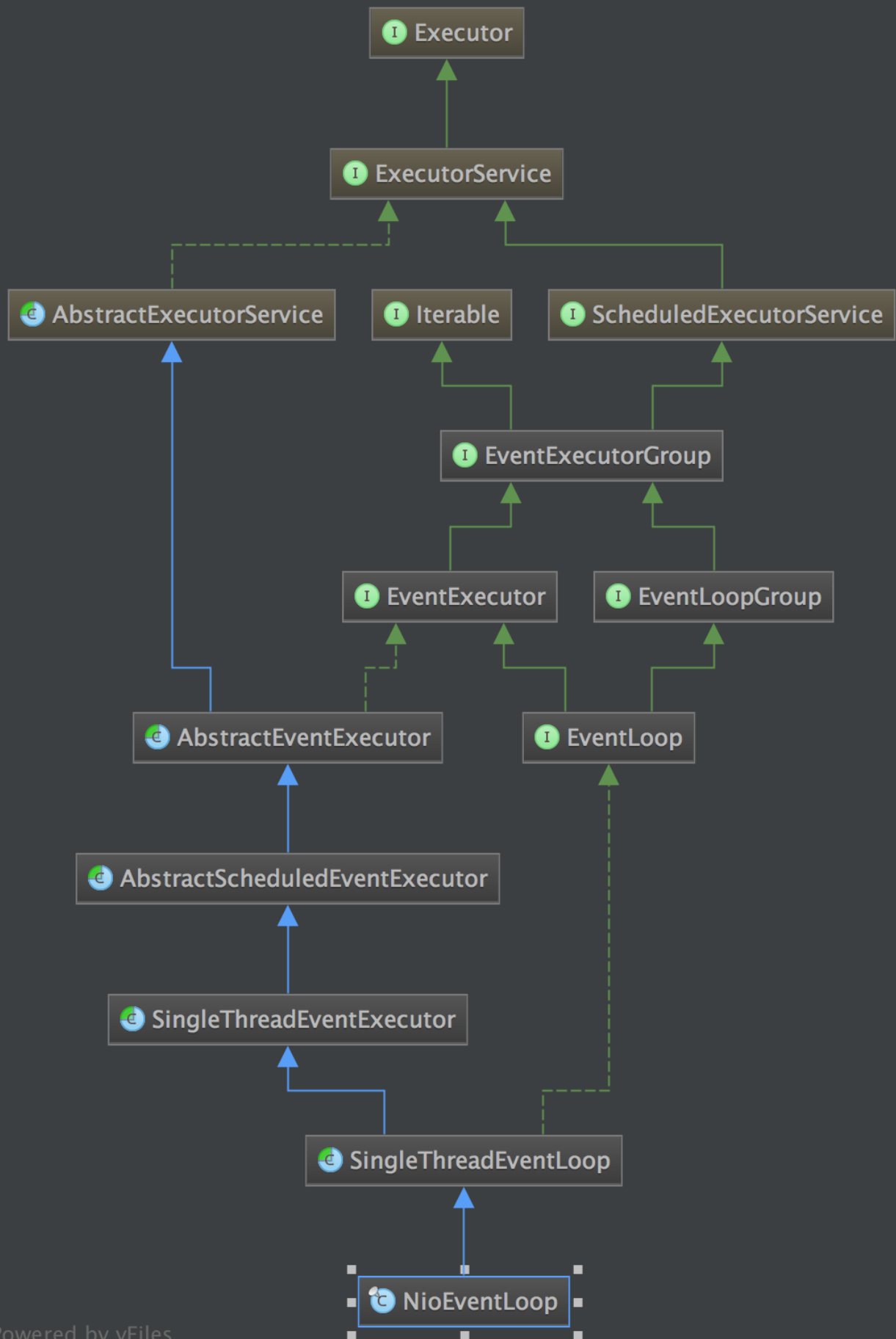
- EventLoopGroup(其实是MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children 数组, 其大小是 nThreads, 这样就构成了一个线程池
- 如果我们在实例化 NioEventLoopGroup 时, 如果指定线程池大小, 则 nThreads 就是指定的值, 反之是处理器核心数 \* 2
- MultithreadEventExecutorGroup 中会调用 newChild 抽象方法来初始化 children 数组
- 抽象方法 newChild 是在 NioEventLoopGroup 中实现的, 它返回一个 NioEventLoop 实例.
- NioEventLoop 属性:
- SelectorProvider provider 属性: NioEventLoopGroup 构造器中通过 SelectorProvider.provider() 获取一个 SelectorProvider
- Selector selector 属性: NioEventLoop 构造器中通过调用通过 selector = provider.openSelector() 获取一个 selector 对象.

## NioEventLoop

---

NioEventLoop 继承于 SingleThreadEventLoop, 而 SingleThreadEventLoop 又继承于 SingleThreadEventExecutor. 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. 因此我们可以认为, 一个 NioEventLoop 其实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 类层次结构



NioEventLoop 的类层次结构图还是比较复杂的, 不过我们只需要关注几个重要的点即可. 首先 NioEventLoop 的继

承链如下:

```
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` 肩负着两种任务, 第一个是作为 IO 线程, 执行与 Channel 相关的 IO 操作, 包括调用 `select` 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.

## NioEventLoop 的实例化过程

`SingleThreadEventExecutor` 有一个名为 **thread** 的 `Thread` 类型字段, 这个字段就代表了与 `SingleThreadEventExecutor` 关联的本地线程. 下面是这个构造器的代码:

```
protected SingleThreadEventExecutor(
    EventExecutorGroup parent, ThreadFactory threadFactory, boolean addTaskWakesUp)
{
    this.parent = parent;
    this.addTaskWakesUp = addTaskWakesUp;

    thread = threadFactory.newThread(new Runnable() {
        @Override
        public void run() {
            boolean success = false;
            updateLastExecutionTime();
            try {
                SingleThreadEventExecutor.this.run();
                success = true;
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor: ", t);
            } finally {
                // 省略清理代码
                ...
            }
        }
    });
    threadProperties = new DefaultThreadProperties(thread);
    taskQueue = newTaskQueue();
}
```

在 `SingleThreadEventExecutor` 构造器中, 通过 **`threadFactory.newThread`** 创建了一个新的 Java 线程. 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.

## EventLoop 与 Channel 的关联

从上图中我们可以看到, 当调用了 **AbstractChannel#AbstractUnsafe.register** 后, 就完成了 Channel 和 EventLoop 的关联. register 实现如下:

```
@Override
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 OneTimeTask() {
                @Override
                public void run() {
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            ...
        }
    }
}
```

在 **AbstractChannel#AbstractUnsafe.register** 中, 会将一个 EventLoop 赋值给 AbstractChannel 内部的 eventLoop 字段, 到这里就完成了 EventLoop 与 Channel 的关联过程.

## 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. 依照这个思想, 我们只要找到在哪里调用了 SingleThreadEventExecutor 的 thread 字段的 **start()** 方法就可以知道是在哪里启动的这个线程了. 从代码中搜索, thread.start() 被封装到 **SingleThreadEventExecutor.startThread()** 方法中了:

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

**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(). 而这个关键的 **startThread()** 方法又是在哪里调用的呢? 经过方法调用关系搜索, 我们发现, startThread 是在 SingleThreadEventExecutor.execute 方法中调用的:

```
@Override
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.execute()` 方法. 如果留心的读者可能已经注意到了, 我们在 **EventLoop 与 Channel 的关联** 这一小节时, 有提到到在注册 channel 的过程中, 会在 **AbstractChannel#AbstractUnsafe.register** 中调用 `eventLoop.execute` 方法, 在 EventLoop 中进行 Channel 注册代码的执行, `AbstractChannel#AbstractUnsafe.register` 部分代码如下:

```

if (eventLoop.inEventLoop()) {
    register0(promise);
} else {
    try {
        eventLoop.execute(new OneTimeTask() {
            @Override
            public void run() {
                register0(promise);
            }
        });
    } catch (Throwable t) {
        ...
    }
}

```

很显然, 一路从 `Bootstrap.bind` 方法跟踪到 `AbstractChannel#AbstractUnsafe.register` 方法, 整个代码都是在主线程中运行的, 因此上面的 `eventLoop.inEventLoop()` 就为 `false`, 于是进入到 `else` 分支, 在这个分支中调用了 **`eventLoop.execute`**. `eventLoop` 是一个 `NioEventLoop` 的实例, 而 `NioEventLoop` 没有实现 `execute` 方法, 因此调用的是 **`SingleThreadEventExecutor.execute`**:

```

@Override
public void execute(Runnable task) {
    ...
    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, 在这里就调用了 `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.execute` **第一次被调用**时, 就会触发 `startThread()` 的调用, 进而导致了 `EventLoop` 所对应的 Java 线程的启动. 我们将 **EventLoop 与 Channel 的关联** 小节中的时序图补全后, 就得到了 `EventLoop` 启动过程的时序图:  Alt text

## Netty 的 IO 处理循环

在 Netty 中, 一个 `EventLoop` 需要负责两个工作, 第一个是作为 IO 线程, 负责相应的 IO 操作; 第二个是作为任务线程, 执行 `taskQueue` 中的任务. 接下来我们先从 IO 操纵方面入手, 看一下 TCP 数据是如何从 Java NIO Socket 传递到我们的 handler 中的.

Netty 是 Reactor 模型的一个实现, 并且是基于 Java NIO 的, 那么从 **Java NIO 的前生今世之四 NIO Selector 详解** 中我们知道, Netty 中必然有一个 Selector 线程, 用于不断调用 Java NIO 的 `Selector.select` 方法, 查询当前是否有就绪的 IO 事件. 回顾一下在 Java NIO 中所讲述的 Selector 的使用流程:

1. 通过 `Selector.open()` 打开一个 Selector.
2. 将 Channel 注册到 Selector 中, 并设置需要监听的事件(interest set)
3. 不断重复:
  - 调用 `select()` 方法
  - 调用 `selector.selectedKeys()` 获取 selected keys
  - 迭代每个 selected key:
    1. 从 selected key 中获取 对应的 Channel 和附加信息(如果有的话)
    1. 判断是哪些 IO 事件已经就绪了, 然后处理它们. **如果是 OP\_ACCEPT 事件, 则调用 "SocketChannel clientChannel = ((ServerSocketChannel) key.channel()).accept()" 获取 SocketChannel, 并将它设置为非阻塞的, 然后将这个 Channel 注册到 Selector 中.**
    1. 根据需要更改 selected key 的监听事件.
    1. 将已经处理过的 key 从 selected keys 集合中删除.

上面的使用流程用代码来体现就是:

```

/**
 * @author xiongyongshun
 * @Email yongshun1228@gmail.com
 * @version 1.0
 * @created 16/8/1 13:13
 */

```

```

public class NioEchoServer {
    private static final int BUF_SIZE = 256;
    private static final int TIMEOUT = 3000;

    public static void main(String args[]) throws Exception {
        // 打开服务端 Socket
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

        // 打开 Selector
        Selector selector = Selector.open();

        // 服务端 Socket 监听8080端口, 并配置为非阻塞模式
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));
        serverSocketChannel.configureBlocking(false);

        // 将 channel 注册到 selector 中.
        // 通常我们都是先注册一个 OP_ACCEPT 事件, 然后在 OP_ACCEPT 到来时, 再将这个 Channel 的
        OP_READ
        // 注册到 Selector 中.
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            // 通过调用 select 方法, 阻塞地等待 channel I/O 可操作
            if (selector.select(TIMEOUT) == 0) {
                System.out.print(".");
                continue;
            }

            // 获取 I/O 操作就绪的 SelectionKey, 通过 SelectionKey 可以知道哪些 Channel 的哪
            类 I/O 操作已经就绪.
            Iterator<SelectionKey> keyIterator = selector.selectedKeys().iterator();

            while (keyIterator.hasNext()) {

                // 当获取一个 SelectionKey 后, 就要将它删除, 表示我们已经对这个 IO 事件进行了处
                理.
                keyIterator.remove();

                SelectionKey key = keyIterator.next();

                if (key.isAcceptable()) {
                    // 当 OP_ACCEPT 事件到来时, 我们就有从 ServerSocketChannel 中获取一个
                    SocketChannel,
                    // 代表客户端的连接
                    // 注意, 在 OP_ACCEPT 事件中, 从 key.channel() 返回的 Channel 是
                    ServerSocketChannel.
                    // 而在 OP_WRITE 和 OP_READ 中, 从 key.channel() 返回的是
                    SocketChannel.
                    SocketChannel clientChannel = ((ServerSocketChannel)
                    key.channel()).accept();
                    clientChannel.configureBlocking(false);
                    //在 OP_ACCEPT 到来时, 再将这个 Channel 的 OP_READ 注册到 Selector 中.

```

```
// 注意，这里我们如果没有设置 OP_READ 的话，即 interest set 仍然是
// OP_CONNECT 的话，那么 select 方法会一直直接返回。
clientChannel.register(key.selector(), OP_READ,
ByteBuffer.allocate(BUF_SIZE));
}

if (key.isReadable()) {
    SocketChannel clientChannel = (SocketChannel) key.channel();
    ByteBuffer buf = (ByteBuffer) key.attachment();
    long bytesRead = clientChannel.read(buf);
    if (bytesRead == -1) {
        clientChannel.close();
    } else if (bytesRead > 0) {
        key.interestOps(OP_READ | SelectionKey.OP_WRITE);
        System.out.println("Get data length: " + bytesRead);
    }
}

if (key.isValid() && key.isWritable()) {
    ByteBuffer buf = (ByteBuffer) key.attachment();
    buf.flip();
    SocketChannel clientChannel = (SocketChannel) key.channel();

    clientChannel.write(buf);

    if (!buf.hasRemaining()) {
        key.interestOps(OP_READ);
    }
    buf.compact();
}

}

}

}
```

还记得不, 上面操作的第一步 **通过 Selector.open() 打开一个 Selector** 我们已经在第一章的 **Channel 实例化** 这一小节中已经提到了, Netty 中是通过调用 SelectorProvider.openSocketChannel() 来打开一个新的 Java NIO SocketChannel:

```
private static SocketChannel newSocket(SelectorProvider provider) {
    ...
    return provider.openSocketChannel();
}
```

第二步 将 Channel 注册到 Selector 中, 并设置需要监听的事件(interest set) 的操作我们在第一章 channel 的注册过程中也分析过了, 我们在来回顾一下, 在客户端的 Channel 注册过程中, 会有如下调用链:

```

Bootstrap.initAndRegister ->
    AbstractBootstrap.initAndRegister ->
        MultithreadEventLoopGroup.register ->
            SingleThreadEventLoop.register ->
                AbstractUnsafe.register ->
                    AbstractUnsafe.register0 ->
                        AbstractNioChannel.doRegister

```

在 AbstractUnsafe.register 方法中调用了 register0 方法:

```

@Override
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
    // 省略条件判断和错误处理
    AbstractChannel.this.eventLoop = eventLoop;
    register0(promise);
}

```

register0 方法代码如下:

```

private void register0(ChannelPromise promise) {
    boolean firstRegistration = neverRegistered;
    doRegister();
    neverRegistered = false;
    registered = true;
    safeSetSuccess(promise);
    pipeline.fireChannelRegistered();
    // Only fire a channelActive if the channel has never been registered. This
    prevents firing
    // multiple channel actives if the channel is deregistered and re-registered.
    if (firstRegistration && isActive()) {
        pipeline.fireChannelActive();
    }
}

```

register0 又调用了 AbstractNioChannel.doRegister:

```

@Override
protected void doRegister() throws Exception {
    // 省略错误处理
    selectionKey = javaChannel().register(eventLoop().selector, 0, this);
}

```

在这里 javaChannel() 返回的是一个 Java NIO SocketChannel 对象, 我们将此 SocketChannel 注册到前面第一步获取的 Selector 中.

那么接下来的第三步的循环是在哪里实现的呢? 第三步的操作就是我们今天分析的关键, 下面我会一步一步向读者展示出来.

## thread 的 run 循环

在 **EventLoop 的启动** 一节中, 我们已经了解到了, 当 EventLoop.execute **第一次被调用**时, 就会触发 **startThread()** 的调用, 进而导致了 EventLoop 所对应的 Java 线程的启动. 接着我们来更深入一些, 来看一下此线程启动后都会做什么东东吧. 下面是此线程的 run() 方法, 我已经把一些异常处理和收尾工作的代码都去掉了. 这个 run 方法可以说是非常简单, 主要就是调用了 **SingleThreadEventExecutor.this.run()** 方法. 而 SingleThreadEventExecutor.run() 是一个抽象方法, 它的实现在 NioEventLoop 中.

```
thread = threadFactory.newThread(new Runnable() {
    @Override
    public void run() {
        boolean success = false;
        updateLastExecutionTime();
        try {
            SingleThreadEventExecutor.this.run();
            success = true;
        } catch (Throwable t) {
            logger.warn("Unexpected exception from an event executor: ", t);
        } finally {
            ...
        }
    }
});
```

继续跟踪到 NioEventLoop.run() 方法, 其源码如下:

```
@Override
protected void run() {
    for (;;) {
        boolean oldWakeup = wakeup.getAndSet(false);
        try {
            if (hasTasks()) {
                selectNow();
            } else {
                select(oldWakeup);
                if (wakeup.get()) {
                    selector.wakeup();
                }
            }
        }

        cancelledKeys = 0;
        needsToSelectAgain = false;
        final int ioRatio = this.ioRatio;
        if (ioRatio == 100) {
            processSelectedKeys();
            runAllTasks();
        } else {
            final long ioStartTime = System.nanoTime();

            processSelectedKeys();

            final long ioTime = System.nanoTime() - ioStartTime;
            runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
        }
    }
}
```

```

        if (isShuttingDown()) {
            closeAll();
            if (confirmShutdown()) {
                break;
            }
        }
    } catch (Throwable t) {
        ...
    }
}
}

```

啊哈, 看到了上面代码的 **for(;;)** 所构成的死循环了没? 原来 `NioEventLoop` 事件循环的核心就是这里! 现在我们把上面所提到的 `Selector` 使用步骤的第三步的部分也找到了. 这个 `run` 方法可以说是 `Netty NIO` 的核心, 属于重中之重, 把它分析明白了, 那么对 `Netty` 的事件循环机制也就了解了大部分了. 让我们一鼓作气, 继续分析下去吧!

## IO 事件的轮询

首先, 在 `run` 方法中, 第一步是调用 **`hasTasks()`** 方法来判断当前任务队列中是否有任务:

```

protected boolean hasTasks() {
    assert inEventLoop();
    return !taskQueue.isEmpty();
}

```

这个方法很简单, 仅仅是检查了一下 **`taskQueue`** 是否为空. 至于 `taskQueue` 是什么呢, 其实它就是存放一系列的需由此 `EventLoop` 所执行的任务列表. 关于 `taskQueue`, 我们这里暂时不表, 等到后面再来详细分析它. 当 `taskQueue` 不为空时, 就执行到了 `if` 分支中的 `selectNow()` 方法. 然而当 `taskQueue` 为空时, 执行的是 `select(oldWakeup)` 方法. 那么 **`selectNow()`** 和 **`select(oldWakeup)`** 之间有什么区别呢? 来看一下, `selectNow()` 的源码如下:

```

void selectNow() throws IOException {
    try {
        selector.selectNow();
    } finally {
        // restore wakeup state if needed
        if (wakeup.get()) {
            selector.wakeup();
        }
    }
}
}

```

首先调用了 **`selector.selectNow()`** 方法, 这里 `selector` 是什么大家还有印象不? 我们在第一章 **Netty 源码分析之一 揭开 Bootstrap 神秘的红盖头 (客户端)** 时对它有过介绍, 这个 **`selector`** 字段正是 `Java NIO` 中的多路复用器 **`Selector`**. 那么这里 **`selector.selectNow()`** 就很好理解了, `selectNow()` 方法会检查当前是否有就绪的 `IO` 事件, 如果有, 则返回就绪 `IO` 事件的个数; 如果没有, 则返回 0. 注意, `selectNow()` 是立即返回的, 不会阻塞当前线程. 当 `selectNow()` 调用后, `finally` 语句块中会检查 `wakeup` 变量是否为 `true`, 当为 `true` 时, 调用 `selector.wakeup()` 唤醒 `select()` 的阻塞调用.

看了 if 分支的 `selectNow` 方法后, 我们再来看一下 else 分支的 **`select(oldWakeup)`** 方法. 其实 else 分支的 **`select(oldWakeup)`** 方法的处理逻辑比较复杂, 而我们这里的暂时不是分析这个方法调用的具体工作, 因此我这里长话短说, 只列出我们关注的内如:

```
private void select(boolean oldwakeup) throws IOException {
    Selector selector = this.selector;
    try {
        ...
        int selectedKeys = selector.select(timeoutMillis);
        ...
    } catch (CancelledKeyException e) {
        ...
    }
}
```

在这个 `select` 方法中, 调用了 **`selector.select(timeoutMillis)`**, 而这个调用是会阻塞住当前线程的, `timeoutMillis` 是阻塞的超时时间. 到这里, 我们可以看到, 当 **`hasTasks()`** 为真时, 调用的 **`selectNow()`** 方法是不会阻塞当前线程的, 而当 **`hasTasks()`** 为假时, 调用的 **`select(oldWakeup)`** 是会阻塞当前线程的. 这其实也很好理解: 当 `taskQueue` 中没有任务时, 那么 Netty 可以阻塞地等待 IO 就绪事件; 而当 `taskQueue` 中有任务时, 我们自然地希望所提交的任务可以尽快地执行, 因此 Netty 会调用非阻塞的 `selectNow()` 方法, 以保证 `taskQueue` 中的任务尽快可以执行.

## IO 事件的处理

在 `NioEventLoop.run()` 方法中, 第一步是通过 `select/selectNow` 调用查询当前是否有就绪的 IO 事件. 那么当有 IO 事件就绪时, 第二步自然就是处理这些 IO 事件啦. 首先让我们来看一下 `NioEventLoop.run` 中循环的剩余部分:

```
final int ioRatio = this.ioRatio;
if (ioRatio == 100) {
    processSelectedKeys();
    runAllTasks();
} else {
    final long ioStartTime = System.nanoTime();

    processSelectedKeys();

    final long ioTime = System.nanoTime() - ioStartTime;
    runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
}
```

上面列出的代码中, 有两个关键的调用, 第一个是 **`processSelectedKeys()`** 调用, 根据字面意思, 我们可以猜出这个方法肯定是查询就绪的 IO 事件, 然后处理它; 第二个调用是 **`runAllTasks()`**, 这个方法我们也可以一眼就看出来它的功能就是运行 `taskQueue` 中的任务. 这里的代码还有一个十分有意思的地方, 即 **`ioRatio`**. 那什么是 **`ioRatio`** 呢? 它表示的是此线程分配给 IO 操作所占的时间比(即运行 `processSelectedKeys` 耗时在整个循环中所占用的时间). 例如 `ioRatio` 默认是 50, 则表示 IO 操作和执行 task 的所占用的线程执行时间比是 1 : 1. 当知道了 IO 操作耗时和它所占用的时间比, 那么执行 task 的时间就可以很方便的计算出来了:



设 IO 操作耗时为 `ioTime`, `ioTime` 占的时间比例为 `ioRatio`, 则:

$$\text{ioTime} / \text{ioRatio} = \text{taskTime} / \text{taskRatio}$$
$$\text{taskRatio} = 100 - \text{ioRatio}$$
$$\Rightarrow \text{taskTime} = \text{ioTime} * (100 - \text{ioRatio}) / \text{ioRatio}$$

根据上面的公式, 当我们设置 `ioRate = 70` 时, 则表示 IO 运行耗时占比为 70%, 即假设某次循环一共耗时为 100ms, 那么根据公式, 我们知道 **`processSelectedKeys()`** 方法调用所耗时大概为 70ms (即 IO 耗时), 而 **`runAllTasks()`** 耗时大概为 30ms (即执行 task 耗时). 当 `ioRatio` 为 100 时, Netty 就不考虑 IO 耗时的占比, 而是分别调用 **`processSelectedKeys()`**、**`runAllTasks()`**; 而当 `ioRatio` 不为 100 时, 则执行到 `else` 分支, 在这个分支中, 首先记录下 **`processSelectedKeys()`** 所执行的时间 (即 IO 操作的耗时), 然后根据公式, 计算出执行 task 所占用的时间, 然后以此为参数, 调用 **`runAllTasks()`**.

我们这里先分析一下 **`processSelectedKeys()`** 方法调用, **`runAllTasks()`** 我们留到下一节再分析.

**`processSelectedKeys()`** 方法的源码如下:

```
private void processSelectedKeys() {
    if (selectedKeys != null) {
        processSelectedKeysOptimized(selectedKeys.flip());
    } else {
        processSelectedKeysPlain(selector.selectedKeys());
    }
}
```

这个方法中, 会根据 **`selectedKeys`** 字段是否为空, 而分别调用 **`processSelectedKeysOptimized`** 或 **`processSelectedKeysPlain`**. **`selectedKeys`** 字段是在调用 `openSelector()` 方法时, 根据 JVM 平台的不同, 而有设置不同的值, 在我所调试这个值是不为 `null` 的. 其实 **`processSelectedKeysOptimized`** 方法 **`processSelectedKeysPlain`** 没有太大的区别, 为了简单起见, 我们以 **`processSelectedKeysOptimized`** 为例分析一下源码的工作流程吧.

```
private void processSelectedKeysOptimized(SelectionKey[] selectedKeys) {
    for (int i = 0;; i++) {
        final SelectionKey k = selectedKeys[i];
        if (k == null) {
            break;
        }
        selectedKeys[i] = null;

        final Object a = k.attachment();

        if (a instanceof AbstractNioChannel) {
            processSelectedKey(k, (AbstractNioChannel) a);
        } else {
            @SuppressWarnings("unchecked")
            NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
            processSelectedKey(k, task);
        }
        ...
    }
}
```

其实你别看它代码挺多的, 但是关键的点就两个: 迭代 **selectedKeys** 获取就绪的 IO 事件, 然后为每个事件都调用 **processSelectedKey** 来处理它. 这里正好完美对应上了我们提到的 Selector 的使用流程中的第三步里操作. 还有一点需要注意的是, 我们可以调用 **selectionKey.attach(object)** 给一个 selectionKey 设置一个附加的字段, 然后通过 **Object attachedObj = selectionKey.attachment()** 获取它. 上面代码正是通过了 **k.attachment()** 来获取一个附加在 selectionKey 中的对象, 那么这个对象是什么呢? 它又是在哪里设置的呢? 我们再来回忆一下 SocketChannel 是如何注册到 Selector 中的: 在客户端的 Channel 注册过程中, 会有如下调用链:

```
Bootstrap.initAndRegister ->
    AbstractBootstrap.initAndRegister ->
        MultithreadEventLoopGroup.register ->
            SingleThreadEventLoop.register ->
                AbstractUnsafe.register ->
                    AbstractUnsafe.register0 ->
                        AbstractNioChannel.doRegister
```

最后的 AbstractNioChannel.doRegister 方法会调用 **SocketChannel.register** 方法注册一个 SocketChannel 到指定的 Selector:

```
@Override
protected void doRegister() throws Exception {
    // 省略错误处理
    selectionKey = javaChannel().register(eventLoop().selector, 0, this);
}
```

特别注意一下 **register** 的第三个参数, 这个参数是设置 selectionKey 的附加对象的, 和调用 **selectionKey.attach(object)** 的效果一样. 而调用 **register** 所传递的第三个参数是 **this**, 它其实就是一个 **NioSocketChannel** 的实例. 那么这里就很清楚了, 我们在将 SocketChannel 注册到 Selector 中时, 将 SocketChannel 所对应的 NioSocketChannel 以附加字段的方式添加到了 selectionKey 中. 再回到 **processSelectedKeysOptimized** 方法中, 当我们获取到附加的对象后, 我们就调用 **processSelectedKey** 来处理这个 IO 事件:

```
final Object a = k.attachment();

if (a instanceof AbstractNioChannel) {
    processSelectedKey(k, (AbstractNioChannel) a);
} else {
    @SuppressWarnings("unchecked")
    NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
    processSelectedKey(k, task);
}
```

**processSelectedKey** 方法源码如下:

```
private static void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final NioUnsafe unsafe = ch.unsafe();
    ...
    try {
        int readyOps = k.readyOps();

        // 可读事件
```

```

        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 ||
readyOps == 0) {
            unsafe.read();
            if (!ch.isOpen()) {
                // Connection already closed - no need to handle write.
                return;
            }
        }

        // 可写事件
        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            // Call forceFlush which will also take care of clear the OP_WRITE once
there is nothing left to write
            ch.unsafe().forceFlush();
        }

        // 连接建立事件
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            // remove OP_CONNECT as otherwise Selector.select(..) will always return
without blocking
            // See https://github.com/netty/netty/issues/924
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);

            unsafe.finishConnect();
        }
    } catch (CancelledKeyException ignored) {
        unsafe.close(unsafe.voidPromise());
    }
}

```

这个代码是不是很熟悉啊? 完全是 Java NIO 的 Selector 的那一套处理流程嘛! **processSelectedKey** 中处理了三个事件, 分别是:

- OP\_READ, 可读事件, 即 Channel 中收到了新数据可供上层读取.
- OP\_WRITE, 可写事件, 即上层可以向 Channel 写入数据.
- OP\_CONNECT, 连接建立事件, 即 TCP 连接已经建立, Channel 处于 active 状态.

下面我们分别根据这三个事件来看一下 Netty 是怎么处理的吧.

## OP\_READ 处理

当就绪的 IO 事件是 **OP\_READ**, 代码会调用 **unsafe.read()** 方法, 即:

```

// 可读事件
if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
    unsafe.read();
    if (!ch.isOpen()) {
        // Connection already closed - no need to handle write.
        return;
    }
}

```

unsafe 这个字段, 我们已经和它打了太多的交道了, 在第一章 **Netty 源码分析之一 揭开 Bootstrap 神秘的红盖头 (客户端)** 中我们已经对它进行过浓墨重彩地分析了, 最后我们确定了它是一个 **NioSocketChannelUnsafe** 实例, 负责的是 Channel 的底层 IO 操作. 我们可以利用 IntelliJ IDEA 提供的 **Go To Implementations** 功能, 寻找到这个方法的实现. 最后我们发现这个方法没有在 **NioSocketChannelUnsafe** 中实现, 而是在它的父类 **AbstractNioByteChannel** 实现的, 它的实现源码如下:

```
@Override
public final void read() {
    ...
    ByteBuf byteBuf = null;
    int messages = 0;
    boolean close = false;
    try {
        int totalReadAmount = 0;
        boolean readPendingReset = false;
        do {
            byteBuf = allocHandle.allocate(allocator);
            int writable = byteBuf.writableBytes();
            int localReadAmount = doReadBytes(byteBuf);

            // 检查读取结果.
            ...

            pipeline.fireChannelRead(byteBuf);
            byteBuf = null;

            ...

            totalReadAmount += localReadAmount;

            // 检查是否是配置了自动读取, 如果不是, 则立即退出循环.
            ...
        } while (++ messages < maxMessagesPerRead);

        pipeline.fireChannelReadComplete();
        allocHandle.record(totalReadAmount);

        if (close) {
            closeOnRead(pipeline);
            close = false;
        }
    } catch (Throwable t) {
        handleReadException(pipeline, byteBuf, t, close);
    } finally {
    }
}
```

**read()** 源码比较长, 我为了篇幅起见, 删除了部分代码, 只留下了主干. 不过我建议读者朋友们自己一定要看一下 **read()** 源码, 这对理解 Netty 的 EventLoop 十分有帮助. 上面 **read** 方法其实归纳起来, 可以认为做了如下工作:

1. 分配 ByteBuf
2. 从 SocketChannel 中读取数据

3. 调用 `pipeline.fireChannelRead` 发送一个 inbound 事件.

前面两点没什么好说的, 第三点 `pipeline.fireChannelRead` 读者朋友们看到了有没有会心一笑地感觉呢? 反正我看到这里时是有的. `pipeline.fireChannelRead` 正好就是我们在第二章 **Netty 源码分析之二 贯穿Netty的大动脉 — ChannelPipeline (二)** 中分析的 inbound 事件起点. 当调用了 `pipeline.fireIN_EVT()` 后, 那么就产生了一个 inbound 事件, 此事件会以 `head -> customContext -> tail` 的方向依次流经 ChannelPipeline 中的各个 handler. 调用了 `pipeline.fireChannelRead` 后, 就是 ChannelPipeline 中所需要做的工作了, 这些我们已经在第二章中有过详细讨论, 这里就展开了.

## OP\_WRITE 处理

**OP\_WRITE** 可写事件代码如下. 这里代码比较简单, 没有详细分析的必要了.

```
if ((readyOps & SelectionKey.OP_WRITE) != 0) {
    // Call forceFlush which will also take care of clear the OP_WRITE once there is
    nothing left to write
    ch.unsafe().forceFlush();
}
```

## OP\_CONNECT 处理

最后一个事件是 **OP\_CONNECT**, 即 TCP 连接已建立事件.

```
if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
    // remove OP_CONNECT as otherwise Selector.select(..) will always return without
    blocking
    // See https://github.com/netty/netty/issues/924
    int ops = k.interestOps();
    ops &= ~SelectionKey.OP_CONNECT;
    k.interestOps(ops);

    unsafe.finishConnect();
}
```

**OP\_CONNECT** 事件的处理中, 只做了两件事情:

1. 正如代码中的注释所言, 我们需要将 **OP\_CONNECT** 从就绪事件集中清除, 不然会一直有 **OP\_CONNECT** 事件.
2. 调用 `unsafe.finishConnect()` 通知上层连接已建立

`unsafe.finishConnect()` 调用最后会调用到 `pipeline().fireChannelActive()`, 产生一个 inbound 事件, 通知 pipeline 中的各个 handler TCP 通道已建立(即 `ChannelInboundHandler.channelActive` 方法会被调用)

到了这里, 我们整个 `NioEventLoop` 的 IO 操作部分已经了解完了, 接下来的一节我们要重点分析一下 **Netty 的任务队列机制**.

# Netty 的任务队列机制

我们已经提到过, 在 Netty 中, 一个 `NioEventLoop` 通常需要肩负起两种任务, 第一个是作为 IO 线程, 处理 IO 操作; 第二个就是作为任务线程, 处理 `taskQueue` 中的任务. 这一节的重点就是分析一下 `NioEventLoop` 的任务队列机制的.

## Task 的添加

## 普通 Runnable 任务

NioEventLoop 继承于 SingleThreadEventExecutor, 而 SingleThreadEventExecutor 中有一个 **Queue taskQueue** 字段, 用于存放添加的 Task. 在 Netty 中, 每个 Task 都使用一个实现了 Runnable 接口的实例来表示. 例如当我们需要将一个 Runnable 添加到 taskQueue 中时, 我们可以进行如下操作:

```
EventLoop eventLoop = channel.eventLoop();
eventLoop.execute(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello, Netty!");
    }
});
```

当调用 execute 后, 实际上是调用到了 SingleThreadEventExecutor.execute() 方法, 它的实现如下:

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

    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        addTask(task);
    } else {
        startThread();
        addTask(task);
        if (isShutdown() && removeTask(task)) {
            reject();
        }
    }

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

而添加任务的 **addTask** 方法的源码如下:

```
protected void addTask(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    if (isShutdown()) {
        reject();
    }
    taskQueue.add(task);
}
```

因此实际上, **taskQueue** 是存放着待执行的任务的队列.

## schedule 任务

除了通过 `execute` 添加普通的 `Runnable` 任务外, 我们还可以通过调用 `eventLoop.scheduleXXX` 之类的方法来添加一个定时任务. `EventLoop` 中实现任务队列的功能在超类 `SingleThreadEventExecutor` 实现的, 而 `schedule` 功能的实现是在 `SingleThreadEventExecutor` 的父类, 即 `AbstractScheduledEventExecutor` 中实现的. 在 `AbstractScheduledEventExecutor` 中, 有以 `scheduledTaskQueue` 字段:

```
Queue<ScheduledFutureTask<?>> scheduledTaskQueue;
```

`scheduledTaskQueue` 是一个队列(`Queue`), 其中存放的元素是 **`ScheduledFutureTask`**. 而 **`ScheduledFutureTask`** 我们很容易猜到, 它是对 `Schedule` 任务的一个抽象. 我们来看一下 `AbstractScheduledEventExecutor` 所实现的 **`schedule`** 方法吧:

```
@Override
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {
    ObjectUtil.checkNotNull(command, "command");
    ObjectUtil.checkNotNull(unit, "unit");
    if (delay < 0) {
        throw new IllegalArgumentException(
            String.format("delay: %d (expected: >= 0)", delay));
    }
    return schedule(new ScheduledFutureTask<Void>(
        this, command, null,
        ScheduledFutureTask.deadlineNanos(unit.toNanos(delay))));
}
```

这是其中一个重载的 `schedule`, 当一个 `Runnable` 传递进来后, 会被封装为一个 **`ScheduledFutureTask`** 对象, 这个对象会记录下这个 `Runnable` 在何时运行、已何种频率运行等信息. 当构建了 **`ScheduledFutureTask`** 后, 会继续调用另一个重载的 `schedule` 方法:

```
<V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {
    if (inEventLoop()) {
        scheduledTaskQueue().add(task);
    } else {
        execute(new OneTimeTask() {
            @Override
            public void run() {
                scheduledTaskQueue().add(task);
            }
        });
    }

    return task;
}
```

在这个方法中, `ScheduledFutureTask` 对象就会被添加到 **`scheduledTaskQueue`** 中了.

## 任务的执行

当一个任务被添加到 `taskQueue` 后, 它是怎么被 `EventLoop` 执行的呢? 让我们回到 `NioEventLoop.run()` 方法中, 在这个方法里, 会分别调用 **`processSelectedKeys()`** 和 **`runAllTasks()`** 方法, 来进行 IO 事件的处理和 task 的处理. **`processSelectedKeys()`** 方法我们已经分析过了, 下面我们来看一下 **`runAllTasks()`** 中到底有什么名堂吧. `runAllTasks` 方法有两个重载的方法, 一个是无参数的, 另一个有一个参数的. 首先来看一下无参数的 `runAllTasks`:

```
protected boolean runAllTasks() {
    fetchFromScheduledTaskQueue();
    Runnable task = pollTask();
    if (task == null) {
        return false;
    }

    for (;;) {
        try {
            task.run();
        } catch (Throwable t) {
            logger.warn("A task raised an exception.", t);
        }

        task = pollTask();
        if (task == null) {
            lastExecutionTime = ScheduledFutureTask.nanoTime();
            return true;
        }
    }
}
```

我们前面已经提到过, `EventLoop` 可以通过调用 **`EventLoop.execute`** 来将一个 `Runnable` 提交到 `taskQueue` 中, 也可以通过调用 **`EventLoop.schedule`** 来提交一个 `schedule` 任务到 **`scheduledTaskQueue`** 中. 在此方法的一开始调用的 **`fetchFromScheduledTaskQueue()`** 其实就是将 **`scheduledTaskQueue`** 中已经可以执行的(即定时时间已到)的 `schedule` 任务) 拿出来并添加到 `taskQueue` 中, 作为可执行的 `task` 等待被调度执行. 它的源码如下:

```
private void fetchFromScheduledTaskQueue() {
    if (hasScheduledTasks()) {
        long nanoTime = AbstractScheduledEventExecutor.nanoTime();
        for (;;) {
            Runnable scheduledTask = pollScheduledTask(nanoTime);
            if (scheduledTask == null) {
                break;
            }
            taskQueue.add(scheduledTask);
        }
    }
}
```

接下来 **`runAllTasks()`** 方法就会不断调用 **`task = pollTask()`** 从 **`taskQueue`** 中获取一个可执行的 `task`, 然后调用它的 **`run()`** 方法来运行此 `task`.

**注意**, 因为 `EventLoop` 既需要执行 IO 操作, 又需要执行 `task`, 因此我们在调用 `EventLoop.execute` 方法提交任务时, 不要提交耗时任务, 更不能提交一些会造成阻塞的任务, 不然会导致我们的 IO 线程得不到调度, 影响整个程序的并发量.