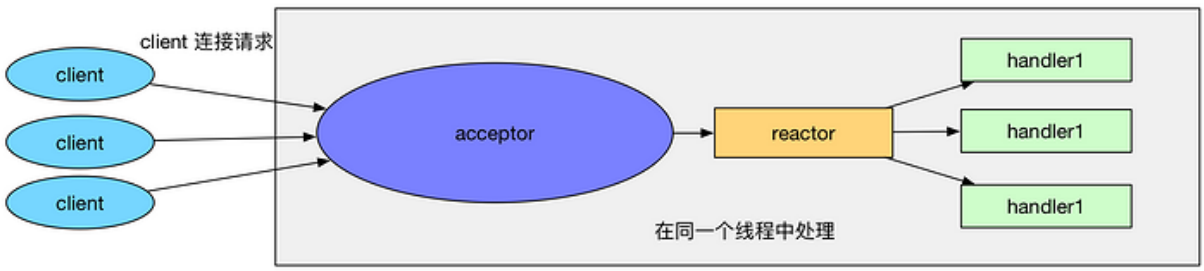
Netty 线程模型分析

> 阅读本文之前，需要了解 NIO 相关知识，可参阅我的文章：[NIO 与 epoll 知识详解](./NIO 与 epoll.pdf)

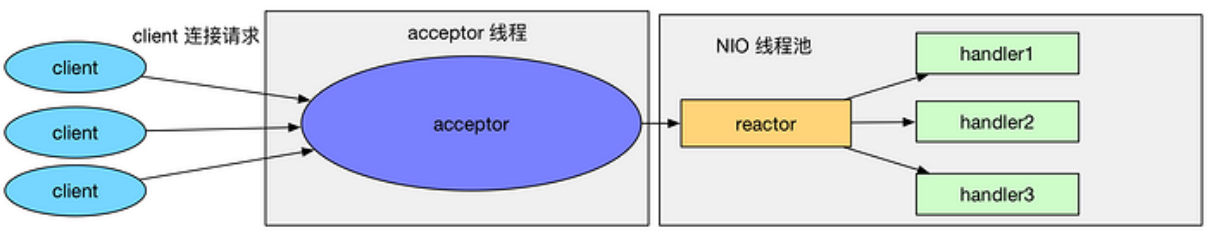
Reactor 线程模型

Netty 的线程模型 是基于 Reactor 线程模型的，Reactor 线程模型分为三种模型：

- 单线程模型：注册所有感兴趣的事件，一个线程管理多个 Channel，单线程轮询 IO 事件是否发生，若事件发生，在同一个线程中调用处理程序，属于 一对一 的模型。

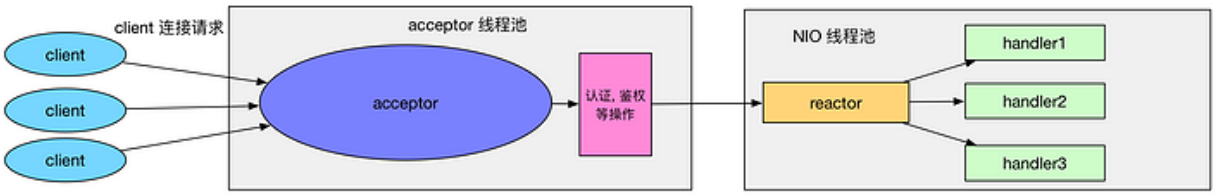


- 多线程模型：多线程模型中，一个线程管理多个 Channel，仍然是单线程轮询 IO 事件是否发生，但将事件发生的处理程序交由线程池运行，属于 一对多 的模型。

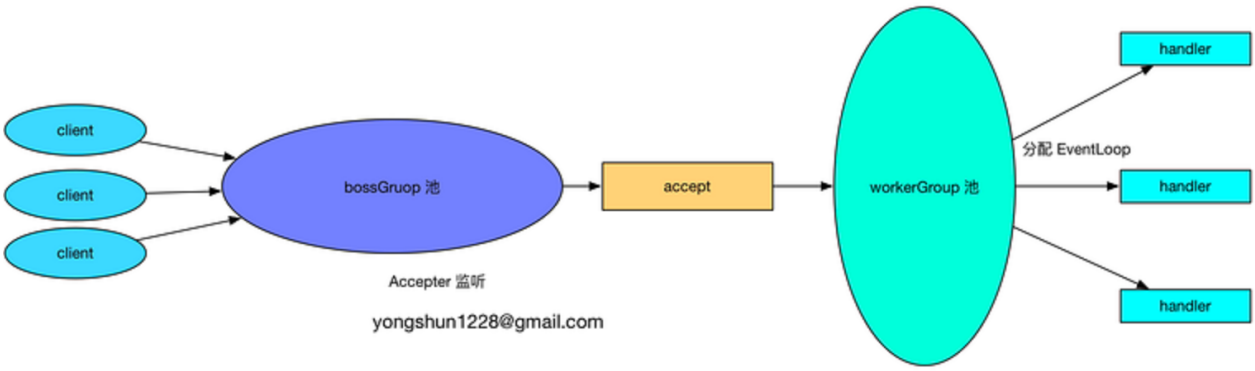


- 主从线程模型：在这种模型中，有多个线程负责 IO 事件的连接，每个负责 IO 事件连接的线程又可以指派多个处理线程，属于 多对多 的模型。

  这种模式通常有两种实现，严格意义上的实现应该是**一个 Channel 可由多个线程管理**，一个线程又可管理多个 Channel，这种实现需要在线程间共享 Channel，Netty 并没有采用这种方式。另一种不太规范的实现是一个线程管理多个 Channel，但创建多个线程，形象的说，就像多个多线程模型一起启动一样。



Netty 中可根据所调参数选择对应的模型，Netty 中有一个专门的线程组 BossGroup 用于接收发送 IO 事件，还有一个线程组 WorkerGroup 用以调用 handler。



> 在 Netty 中，即使设置 bossGroup 为多个线程，一个 Channel 仍然只属于一个线程管理，并不会由多个线程管理，因此如果服务端只有一个 ServerChannel，设置多线程 bossGroup 是没有意义的，除非有多个 Channel。

Netty 初始化线程池

Netty 中的模型就是可创建 BossGroup 和 WorkerGroup 两个线程组，你可以指定对应的线程数。

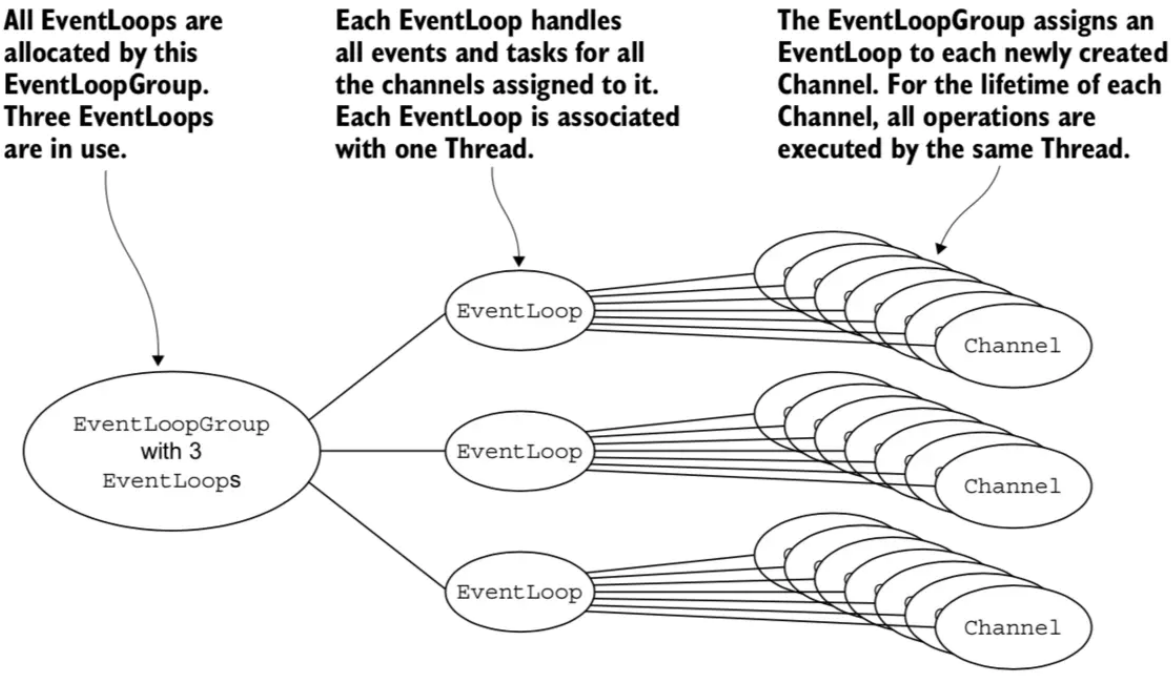


Figure 7.4 `EventLoop` allocation for non-blocking transports (such as NIO and AIO)

在 Netty 中，每一个 EventLoop 其实就是一个线程，Netty 中的 IO 事件就是一个任务，每个任务都会被提交到 EventLoop 中，由 EventLoop 在事件循环中执行。

在 Netty 中，创建一个 EventLoopGroup 时就会自动创建多个 EventLoop 线程，数目由 nThreads 指定。

```
NioEventLoopGroup group = new NioEventLoopGroup(nThreads);
```

如果没有指定线程数，在 MultithreadEventLoopGroup 的静态方法中初始化默认的线程数为 CPU 核心数的两倍：

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

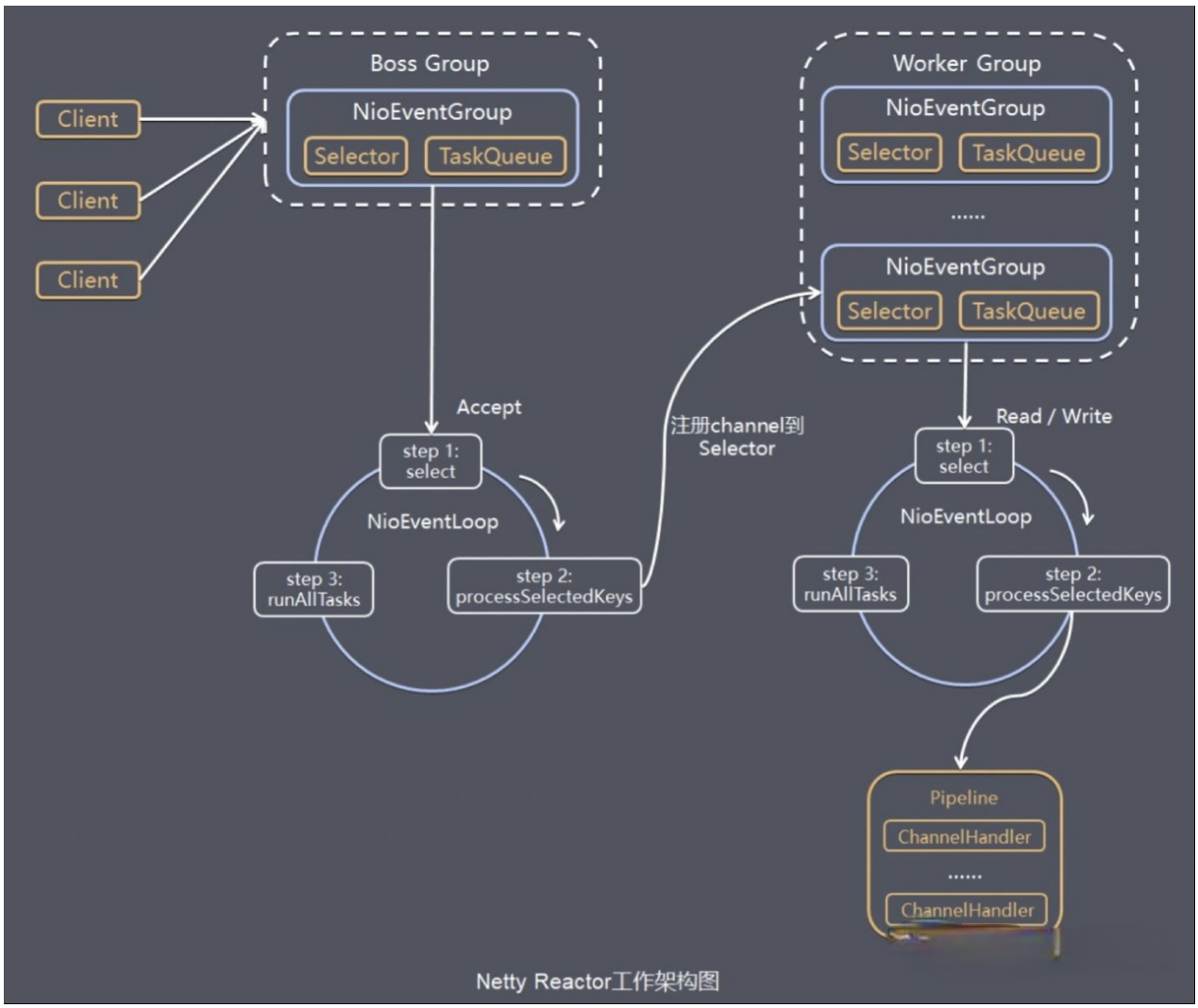
Netty 使用 EventExecutor 数组来保存 EventLoop，EventLoop 继承于 EventExecutor，在 new NioEventLoopGroup 中，通过不断向上传递，在父类 MultithreadEventExecutorGroup 的构造方法中，EventExecutor 数组被创建：

```
private final EventExecutor[] children;
private final Set<EventExecutor> readonlyChildren;

protected MultithreadEventExecutorGroup(int nThreads, Executor executor,
```

```
                                                EventExecutorChooserFactory
chooserFactory, Object... args) {
    children = new EventExecutor[nThreads];
    for (int i = 0; i < nThreads; i ++) {
        // 创建 NioEventLoop，此方法其实就是 new NioEventLoop
        children[i] = newChild(executor, args);
    }

    for (EventExecutor e: children) {
        // 添加监听器
        e.terminationFuture().addListener(terminationListener);
    }
    // 去重
    Set<EventExecutor> childrenSet = new LinkedHashSet<EventExecutor>
(children.length);
    Collections.addAll(childrenSet, children);
    readonlyChildren = Collections.unmodifiableSet(childrenSet);
}
```

所以，在创建 Group 的时候主要就是初始化了 EventLoop 线程组。

BossGroup 工作原理



绑定 Channel

当调用了 `bootstrap.bind()` 方法时，开始正式进入核心环节，我们主要关注 Channel 是如何被绑定到 EventLoop 中的，调用此方法时，会经过如下调用链：

```
AbstractBootstrap.bind()
        -> doBind()
                -> initAndRegister()
                        -> config().group().register(channel)
                                ->
MultithreadEventLoopGroup.register(channel)
                                        ->
SingleThreadEventLoop.register(channel)
                                                ->
AbstractChannel.AbstractUnsafe.register()
```

从 MultithreadEventLoopGroup.register(channel) 方法看起：

```
@Override
public ChannelFuture register(Channel channel) {
    return next().register(channel);
}
```

这个方法调用了 EventExecutorChooserFactory#next() 方法，next() 方法其实就是获取一个 EventLoop，跟进 next() 方法，**发现 Netty 中只提供了轮询的方式获取 EventLoop**：

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

还记得 executors 数组吗？这里 executors 数组其实就是 Group 初始化的 EventLoop 数组，因此通过调用 next() 方法就轮询获得了一个 EventLoop，现在开始将 Channel 绑定到 EventLoop 中。

将目光移到调用链中的 AbstractUnsafe.register:

```
public final void register(EventLoop eventLoop, final ChannelPromise
promise) {
    // 省略, promise 是 channel 的包装, eventLoop 是调用 next() 得到的线程
    AbstractChannel.this.eventLoop = eventLoop;
        // 如果给定的线程已经被启动运行了，那么直接注册
    if (eventLoop.inEventLoop()) {
        register0(promise);
    } else {
        // 否则，要先启动一下，再注册，这里其实是向 Netty 提交了一个任务
```

```
        eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                        register0(promise);
                }
        });
    }
}
```

execute 方法是事件循环的开始方法，但这里我们先关注 register0(promise) 方法，register0 方法内**调用了 AbstractNioChannel#doRegister() 方法，这个方法其实是 Netty NIO 与 JDK NIO 交互的地方：**

```
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            selectionKey =
javaChannel().register(eventLoop().unwrappedSelector(), 0, this);
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                eventLoop().selectNow();
                selected = true;
            } else {
                throw e;
            }
        }
    }
}
```

javaChannel() 方法返回了**已被绑定的、 jdk 原生的 Channel**，此 Channel 由传递的地址注册，eventLoop().unwrappedSelector() 是对应线程的选择器，也是 jdk 原生的 Selector，**doRegister() 方法还向通道附加了 this(AbstractNioChannel) 对象**，在这里 **Channel 正式向选择器注册了**，而 **selectionKey 被 AbstractNioChannel 所保存**，用于后续 select。

事件循环

现在来看看 `execute`(SingleThreadEventExecutor类下) 方法，这个方法将任务(doRegister方法)添加到任务队列后，然后调用了 `startThread` 方法，`startThread` 方法会判断线程是否已被启动：

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

```
            } catch (Throwable cause) {
                STATE_UPDATER.set(this, ST_NOT_STARTED);
                PlatformDependent.throwException(cause);
            }
        }
    }
}
```

如果未启动则调用 `dotartThread` 方法：

```
private void doStartThread() {
    executor.execute(new Runnable() {
        public void run() {
            try {
                SingleThreadEventExecutor.this.run();
            } catch (Throwable t) {
                logger.warn("Unexpected exception from an event executor:
", t);
            }
        }
    }
}
```

此方法**提交任务到 Netty**，执行 SingleThreadEventExecutor.this.run() 方法，**这会新建线程执行**：

```
public void execute(Runnable command) {
    threadFactory.newThread(command).start();
}
```

也就是说到了这里才新建了线程，来看看 run 方法：

```
protected void run() {
    for (;;) {
        switch(selectStrategy.calculateStrategy(selectNowSupplier,
hasTasks())) {
            case SelectStrategy.CONTINUE:
                continue;
            case SelectStrategy.SELECT:
                // 堵塞等待 IO 事件到来，在这里面有解决 NIO 空轮询的 BUG
                // 内部实现有超时控制，一旦超时就会判断下有没有 Task，有 task 就
退出循环，没有就继续循环x
                select(false);
            default:
        }
                // IO 事件与 时间调度任务执行的时间比例，默认是 50%
        final int ioRatio = this.ioRatio;
        final long ioStartTime = System.nanoTime();
        try {
            // 处理 IO 事件
```

```
                processSelectedKeys();
            } finally {
                final long ioTime = System.nanoTime() - ioStartTime;
                // 处理任务，根据 ioRatio 与 IO 事件执行的时间计算出执行任务的最大允
许时间
                // 这不是很准确的，因为无论如何 Netty 必须会等待一个任务完成
                // Netty 会等待一个任务完成再计算时间，如果超时，在下个任务开始前返回
                runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
            }
        }
    }
```

```
 public int calculateStrategy(IntSupplier selectSupplier, boolean
hasTasks) throws Exception {
     // 如果有任务就执行非堵塞的 slect，否则返回 SELECT 执行堵塞的 select
     return hasTasks ? selectSupplier.get() : SelectStrategy.SELECT;
}
```

在 select(wakenUp.getAndSet(false)) 方法中，有效解决了 Java NIO 空轮询的 BUG，出现这个 BUG 的原因是 某个连接出现异常，操作系统返回异常类型，因此会**唤醒阻塞在selector.select上的线程**，但由于 Java NIO 事件设计中并没有此异常事件，也没有对应的解决方案，因此被唤醒的线程将不停的运行，因为 select() 总是会返回(问题没有解决，在 Java 中会返回 0)，以至于占满 CPU。

> 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。

在 Netty 的解决方案中，Netty 会统计空轮询出现的次数，一旦达到阈值时，Netty 会重新新建一个选择器，将原先选择器上有效的连接迁移至新的选择器上，重新运行。

当 IO 事件到来时，Netty 会根据 ioRatio 配置执行 IO 事件 和 任务，执行任务的时间与执行 IO 事件时间所占比例不高于 ioRatio，这么做是为了**防止任务事件长时间堵塞而导致 IO 事件无法执行**。

要想提交一条任务，只需执行如下代码，但记住，千万不要执行长时间堵塞代码，这会导致 IO 事件的执行也被堵塞：

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

执行 IO 事件

processSelectedKeys 方法其实调用了 processSelectedKeysOptimized 方法：

```
private void processSelectedKeysOptimized() {
    for (int i = 0; i < selectedKeys.size; ++i) {
        final SelectionKey k = selectedKeys.keys[i];
        // 为了 GC，将其回收
        selectedKeys.keys[i] = null;
                processSelectedKey(k, (AbstractNioChannel) a);
    }
}
```

在这个方法内，遍历 select() 得到的 key，**Netty 中准备就绪的 key 集合被封装成一个数组对象，而在 jdk 中的实现是一个 HashSet，这么做的好处是为了提高性能，操作数组比操作哈希快得多**，当然，这里 Netty 更多的考虑是考虑到在实践中 HashSet.add() 发送哈希冲突的概率并不小，一旦发生哈希冲突，add 将是 O(N) 级别的。

随后调用 processSelectedKey 方法：

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    try {
        int readyOps = k.readyOps();
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);

            unsafe.finishConnect();
        }

        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            ch.unsafe().forceFlush();
        }
                // 这里会触发连接事件
        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT))
!= 0 || readyOps == 0) {
            unsafe.read();
        }
    } catch (CancelledKeyException ignored) {
        unsafe.close(unsafe.voidPromise());
    }
}
```

这完全就是 jdk 的那一套嘛！根据 readyOps 判断事件的类型，进而调用不同的 unsafe 方法，注意这里 **unsafe 是与对应通道绑定的**，这在代码 `final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe()` 中有体现，所以通过 unsafe 是可以获得到对应通道的配置和数据的，

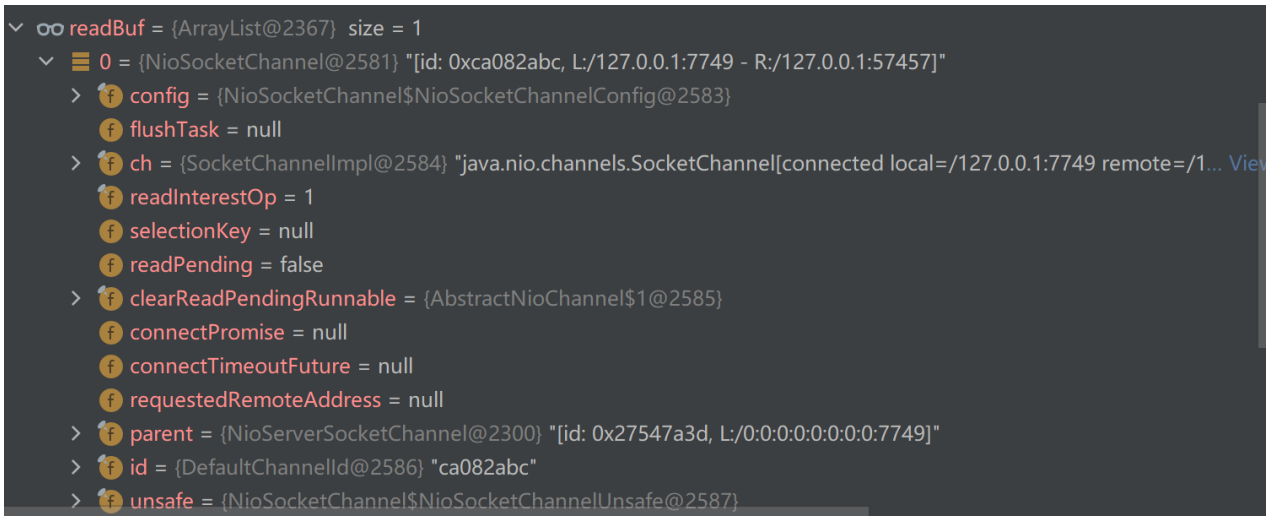
这里分析下 unsafe.read() 方法，：

```
@Override
public void read() {
    assert eventLoop().inEventLoop();
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final RecvByteBufAllocator.Handle allocHandle =
unsafe().recvBufAllocHandle();
    Throwable exception = null;
    try {
        try {
            do {
                // 将消息写入缓存
                int localRead = doReadMessages(readBuf);
                allocHandle.incMessagesRead(localRead);
            } while (allocHandle.continueReading());
        } catch (Throwable t) {
            exception = t;
        }

        int size = readBuf.size();
        for (int i = 0; i < size; i ++) {
            readPending = false;
            // 将每个消息交由 pipeline, Netty 会帮我们整合
            // readBuf 是一个 Object 数组
            pipeline.fireChannelRead(readBuf.get(i));
        }
        readBuf.clear();
        allocHandle.readComplete();
        pipeline.fireChannelReadComplete();
    } finally {
        if (!readPending && !config.isAutoRead()) {
            removeReadOp();
        }
    }
}
}
```

简单来说，此方法就是将数据读入缓存中，默认是使用池化技术的 ByteBuf，然后调用 pipeline.fireChannelRead(byteBuf) 方法进行转发。

一个 pipeline 对应一个 Channel，pipeline 其实就是一个双端链表，链表的节点由 Handler 组成，入站消息由 head 向 tail 依次遍历 Handler，出战事件由 tail 向 head 依次遍历 Handler。

来看看 readBuf.get(i) 到底是什么：



消息竟然是一个 NioSocketChannel，此 Channel 已经是对应 IO的通道了，将用于 WorkerGroup 工作！

NioSocketChannel 到底是如何产生的呢？一切都在 `int localRead = doReadMessages(readBuf);` 这一行代码中：

```
protected int doReadMessages(List<Object> buf) throws Exception {
    // 调用 javaChannel() 获取包装后的服务器 channel，然后调用 accept 获取与客户端的连接
    SocketChannel ch = SocketUtils.accept(javaChannel());
    try {
        if (ch != null) {
            // 将包装后的 channel 添加至缓存
            buf.add(new NioSocketChannel(this, ch));
            return 1;
        }
    } catch (Throwable t) {
        logger.warn("Failed to create a new channel from an accepted socket.", t);
    }

    return 0;
}
```

现在，一个崭新的 NioSocketChannel 终于诞生了，它将作为参数，调用 `fireChannelRead`。

WorkerGroup 工作

上述仅仅只是在 BossGroup 中工作，IO 事件已经就绪，ByteBuf 也已经准备好了，现在终于开始调用pipeline.fireChannelRead(byteBuf) 方法。

在一系列调用链中，会调用 ChannelInboundHandler#channelRead 方法，此方法仅当当前Channel已从对等方读取消息时调用。

这是个接口类，经过调试，这会调用 ServerBootstrap 下 channelRead 方法：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    final Channel child = (Channel) msg;
    child.pipeline().addLast(childHandler);

    try {
        childGroup.register(child).addListener(new
ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws
Exception {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    } catch (Throwable t) {
        forceClose(child, t);
    }
}
```

这个方法向 pipeline 中添加了我们在启动程序中配置的处理者，然后调用了 childGroup.register，这与我们之前分析的 register 方法是一样的，只不过之前默认是采用父类的 BossGroup 的方法，而现在指明调用 childGroup.register 方法，也就是向 WorkerGroup 注册一个通道！**不过现在这个通道已经是对应 IO 的 NioSocketChannel，而不再是服务端接收请求的通道了。**

然后这又会经历：轮询得到 EventLoop -> 执行 register0 方法绑定 Channel -> 执行 doStratThread 方法启动新线程 -> 事件循环 -> 执行 select()，select 必然会立即返回 -> 遍历结果 key 集合。

不过这一次，由于 Channel 已经是和客户端的连接，此次事件将会直接读取数据！

总结

Netty 线程模型其实核心就是 boss 和 worker 两个线程组，通过构造这两个线程组，Netty 完美的贴切了 Reactor 模型，并且通过调整参数，我们可以任意的选择不同的 Reactor 模式。

在 Netty 中，我们可以提交自己的任务， 但 Netty 中的 任务 和 IO 事件处理是在同一个事件循环中运行的，长时间的任务会堵塞 IO 事件的处理，Netty 中的 ioRatio 适当缓解了这个问题，但没有根治，我们仍需小心提交任务。

Netty 中使用优化的 SelectedSelectionKeySet，在 Netty 中，存储 key 集合不再是 HashSet，而是一个数组，这使得 add 十分高效，Netty 通过反射的方式巧妙的替换掉了原生 Selector 的字段。

Netty 自定义的 select 方法相较于原生方法更加高效，并且也解决了一些问题，例如空轮询 Bug，Netty 通过检测空轮询次数，一旦到达阈值，则重建 Selector。

一旦消息经 boss 到达 worker，并由 worker 经过相同 select 步骤后，消息就正式进入管道，在双端链表中传递。