roup bossGroup= new NioEventLoopGroup(); EventLoopGroup workGroup= new NioEventLoopGroup(); ServerBootstrap serverBootstrap = new ServerBootstrap(); serverBootstrap.group(bossGroup,workGroup); |

Netty内部如何实现Reactor线程模型
==========

Netty内部实现Reactor线程模型主要依赖于其事件驱动和异步非阻塞的特性。Netty通过`EventLoop`和`EventLoopGroup`等核心组件来实现高效的线程管理和事件分发，从而支持Reactor模式。

以下是Netty内部实现Reactor线程模型的关键步骤和组件：

0. **EventLoopGroup的创建**：

* `EventLoopGroup`是Netty中用于处理I/O操作的多线程事件循环。它实际上是一个线程池，每个线程都包含一个`EventLoop`。对于服务端，通常会有两个`EventLoopGroup`：一个用于接收客户端连接（bossGroup），另一个用于处理已接收连接上的I/O操作（workerGroup）。
1. **EventLoop的创建与注册**：

* 每个`EventLoop`都绑定到一个`Selector`上，用于监听网络事件。当一个新的连接建立时，该连接会被注册到某个`EventLoop`的`Selector`上。
* `EventLoop`内部维护了一个任务队列，用于存放待处理的I/O任务和定时任务。它还负责处理这些任务，并在必要时将任务分发给其他线程执行。

```
protected void run() {
    //Select计数
    int selectCnt = 0;
    for (;;) {
        try {
            int strategy;
            try {
                //计算Select的策略 <1>
                strategy =
selectStrategy.calculateStrategy(selectNowSupplier, hasTasks());
```

```
                switch (strategy) {
                case SelectStrategy.CONTINUE:
                    continue;

                case SelectStrategy.BUSY_WAIT:
                    // fall-through to SELECT since the busy-wait is not
supported with NIO

                case SelectStrategy.SELECT:
                    //当没有普通任务时，返回定时任务最近一次要执行的时间，如
果有没有定时任务则返回-1
                    long curDeadlineNanos =
nextScheduledTaskDeadlineNanos();
                    if (curDeadlineNanos == -1L) {
                        //如果没有定时任务，则将最近执行时间设置为Integer的最
大值
                        curDeadlineNanos = NONE; // nothing on the calendar
                    }
                    //设置下一次的唤醒时间
                    nextWakeupNanos.set(curDeadlineNanos);
                    try {
                        if (!hasTasks()) {
                        }
            } catch (Error e) {
                throw e;
            } catch (Throwable t) {
                handleLoopException(t);
            }
        }
    }
}


```

3. **事件监听与分发**：

* 当`Selector`监听到网络事件（如连接建立、数据可读等）时，它会通知对应的`EventLoop`。
* `EventLoop`根据事件的类型，从已注册的`Channel`中查找对应的`ChannelPipeline`，并将事件分发给`ChannelPipeline`中的`ChannelHandler`进行处理。

4. **ChannelHandler的链式处理**：

* `ChannelHandler`是Netty中处理网络事件的核心组件。它们以链式的方式组织在`ChannelPipeline`中，每个`ChannelHandler`都可以对事件进行拦截和处

理。
* 当事件到达某个`ChannelHandler`时，该`ChannelHandler`可以决定是继续将事件传递给下一个`ChannelHandler`，还是直接处理并结束事件的传递。
5. **异步非阻塞操作**：

* Netty的所有I/O操作都是异步非阻塞的。这意味着当发起一个I/O操作时，Netty不会等待操作完成就立即返回。相反，它会将操作的结果或异常通过回调的方式通知给调用者。
* 这种异步非阻塞的特性使得Netty能够高效地处理大量并发连接和事件，提高了系统的吞吐量和响应速度。
6. **线程模型优化**：

* Netty还支持多种线程模型优化技术，如多Reactor模式、零拷贝等，以进一步提高性能。
* 在多Reactor模式中，Netty可以创建多个`EventLoopGroup`来处理不同类型的任务或不同优先级的任务，从而更好地利用系统资源。

通过上述组件和机制，Netty内部实现了高效的Reactor线程模型，使得开发者能够轻松地构建高性能、高可靠性的网络应用。

在事件分发过程中，Netty如何避免竞争条件
==========

Netty在事件分发过程中通过一系列机制来避免竞争条件，确保线程安全和高效的事件处理。以下是一些关键的避免竞争条件的策略：

0. **单线程模型中的事件处理**： 在单线程Reactor模型中，虽然存在竞争条件的潜在风险，但由于所有事件都在一个线程中处理，因此实际上避免了多线程间的竞争。然而，这种模型在处理高并发时可能会成为瓶颈，因为任何事件的处理都会阻塞其他事件的处理。
1. **多线程模型中的线程隔离**： 对于多线程模型（如主从多线程Reactor模型），Netty通过将不同类型的任务分配给不同的线程或线程组来减少竞争。例如，接受新连接的任务和处理已建立连接上I/O任务可以被分配给不同的线程池。这样，处理不同任务的线程之间就不会发生竞争。
2. **无锁化设计**： Netty在内部实现中尽可能使用无锁数据结构和算法，以减少锁竞争的开销。无锁数据结构通过原子操作和内存可见性保证线程安全，避免了传统锁机制带来的性能开销和潜在的死锁问题。
3. **细粒度的锁策略**： 当需要使用锁时，Netty会采用细粒度的锁策略，只对需要同步的共享资源加锁，以减少锁的粒度，从而降低锁竞争的可能性。这通常涉及到对关键部分的精确分析和优化。
4. **事件队列的线程安全**： 事件队列是Netty中事件分发的重要组件，它必须是线程安全的。Netty通过内部同步机制确保多个线程对事件队列的并发访问

不会导致数据不一致或其他竞争条件。
5. **事件分发的顺序性**： Netty确保事件按照它们发生的顺序进行分发，这有助于避免由于事件乱序导致的竞争条件。例如，对于同一个Channel的事件，Netty会保证它们按照发生的顺序被处理。
6. **避免共享状态**： 尽量减少跨线程共享的状态可以大大降低竞争条件的风险。Netty鼓励开发者使用局部变量或线程局部变量来存储状态，而不是依赖全局变量或共享对象。
原文链接: https://juejin.cn/post/7356772920276238386