18 源码篇：解密 Netty Reactor 线程模型

通过第一章 Netty 基础课程的学习，我们知道 Reactor 线程模型是 Netty 实现高性能的核心所在，在 Netty 中 EventLoop 是 Reactor 线程模型的核心处理引擎，那么 EventLoop 到底是如何实现的呢？又是如何保证高性能和线程安全性的呢？今天这节课让我们一起一探究竟。

> 说明：本文参考的 Netty 源码版本为 4.1.42.Final。

Reactor 线程执行的主流程

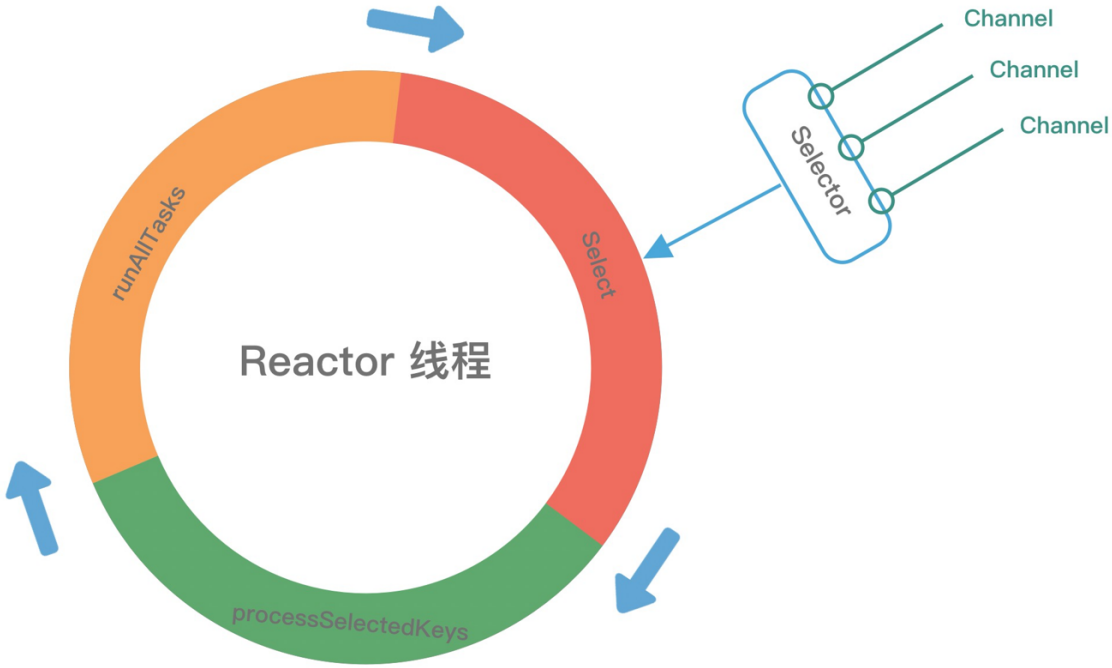
在《事件调度层：为什么 EventLoop 是 Netty 的精髓》的课程中，我们介绍了 EventLoop 的概貌，因为 Netty 是基于 NIO 实现的，所以推荐使用 NioEventLoop 实现，我们再次通过 NioEventLoop 的核心入口 run() 方法回顾 Netty Reactor 线程模型执行的主流程，并以此为基础继续深入研究 NioEventLoop 的逻辑细节。

```
protected void run() {
    for (;;) {
        try {
            try {
                switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTask
                case SelectStrategy.CONTINUE:
                    continue;
                case SelectStrategy.BUSY_WAIT:
                case SelectStrategy.SELECT:
                    select(wakenUp.getAndSet(false)); // 轮询 I/O 事件
                    if (wakenUp.get()) {
                        selector.wakeup();
```

```
                }
            default:
            }
        } catch (IOException e) {
            rebuildSelector0();
            handleLoopException(e);
            continue;
        }
        cancelledKeys = 0;
        needsToSelectAgain = false;
        final int ioRatio = this.ioRatio;
        if (ioRatio == 100) {
            try {
                processSelectedKeys(); // 处理 I/O 事件
            } finally {
                runAllTasks(); // 处理所有任务
            }
        } else {
            final long ioStartTime = System.nanoTime();
            try {
                processSelectedKeys(); // 处理 I/O 事件
            } finally {
                final long ioTime = System.nanoTime() - ioStartTime;
                runAllTasks(ioTime * (100 - ioRatio) / ioRatio); // 处理完 I/O
            }
        }
    } catch (Throwable t) {
        handleLoopException(t);
```

```
            }
            try {
                if (isShuttingDown()) {
                    closeAll();
                    if (confirmShutdown()) {
                        return;
                    }
                }
            } catch (Throwable t) {
                handleLoopException(t);
            }
        }
    }
```

NioEventLoop 的 run() 方法是一个无限循环，没有任何退出条件，在不间断循环执行以下三件事情，可以用下面这张图形象地表示。



- 轮询 I/O 事件（select）：轮询 Selector 选择器中已经注册的所有 Channel 的 I/O 事件。
- 处理 I/O 事件（processSelectedKeys）：处理已经准备就绪的 I/O 事件。
- 处理异步任务队列（runAllTasks）：Reactor 线程还有一个非常重要的职责，就是处理任务队列中的非 I/O 任务。Netty 提供了 ioRatio 参数用于调整 I/O 事件处理和任务处理的时间比例。

下面我们对 NioEventLoop 的三个步骤进行详细的介绍。

轮询 I/O 事件

我们首先聚焦在轮询 I/O 事件的关键代码片段：

```
case SelectStrategy.CONTINUE:

    continue;

case SelectStrategy.BUSY_WAIT:

case SelectStrategy.SELECT:

    select(wakenUp.getAndSet(false));

    if (wakenUp.get()) {

        selector.wakeup();

    }
```

NioEventLoop 通过核心方法 select() 不断轮询注册的 I/O 事件。当没有 I/O 事件产生时，为了避免 NioEventLoop 线程一直循环空转，在获取 I/O 事件或者异步任务时需要阻塞线程，等待 I/O 事件就绪或者异步任务产生后才唤醒线程。NioEventLoop 使用 wakeUp 变量表示是否唤醒 selector，Netty 在每一次执行新的一轮循环之前，都会将 wakeUp 设置为 false。

Netty 提供了选择策略 SelectStrategy 对象，它用于控制 select 循环行为，包含 CONTINUE、SELECT、BUSY_WAIT 三种策略，因为 NIO 并不支持 BUSY_WAIT，所以 BUSY_WAIT 与 SELECT 的执行逻辑是一样的。在 I/O 事件循环的过程中 Netty 选择使用何种策略，具体的判断依据如下：

```
// DefaultSelectStrategy#calculateStrategy

public int calculateStrategy(IntSupplier selectSupplier, boolean hasTasks) throws E
```

```
        return hasTasks ? selectSupplier.get() : SelectStrategy.SELECT;
    }
    // NioEventLoop#selectNowSupplier
    private final IntSupplier selectNowSupplier = new IntSupplier() {
        @Override
        public int get() throws Exception {
            return selectNow();
        }
    }
    // NioEventLoop#selectNow
    int selectNow() throws IOException {
        try {
            return selector.selectNow();
        } finally {
            if (wakenUp.get()) {
                selector.wakeup();
            }
        }
    }
```

如果当前 NioEventLoop 线程存在异步任务，会通过 selectSupplier.get() 最终调用到 selectNow() 方法，selectNow() 是非阻塞，执行后立即返回。如果存在就绪的 I/O 事件，那么会走到 default 分支后直接跳出，然后执行 I/O 事件处理 processSelectedKeys 和异步任务队列处理 runAllTasks 的逻辑。所以在存在异步任务的场景，NioEventLoop 会优先保证 CPU 能够及时处理异步任务。

当 NioEventLoop 线程的不存在异步任务，即任务队列为空，返回的是 SELECT 策略, 就会调用 select(boolean oldWakenUp) 方法，接下来我们看看 select() 内部是如何实现的：

```
    private void select(boolean oldWakenUp) throws IOException {
        Selector selector = this.selector;
```

```
try {
    int selectCnt = 0;
    long currentTimeNanos = System.nanoTime();
    long selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);
    long normalizedDeadlineNanos = selectDeadLineNanos - initialNanoTime();
    if (nextWakeupTime != normalizedDeadlineNanos) {
        nextWakeupTime = normalizedDeadlineNanos;
    }
    for (;;) {
        // ------ 1. 检测 select 阻塞操作是否超过截止时间 ------
        long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L)
        if (timeoutMillis <= 0) {
            if (selectCnt == 0) {
                selector.selectNow();
                selectCnt = 1;
            }
            break;
        }
        // ------ 2. 轮询过程中如果有任务产生，中断本次轮询
        if (hasTasks() && wakenUp.compareAndSet(false, true)) {
            selector.selectNow();
            selectCnt = 1;
            break;
        }
        // ------ 3. select 阻塞等待获取 I/O 事件 ------
        int selectedKeys = selector.select(timeoutMillis);
        selectCnt ++;
        if (selectedKeys != 0 || oldWakenUp || wakenUp.get() || hasTasks() || h
```

```
                break;
            }
            if (Thread.interrupted()) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Selector.select() returned prematurely because "
                            "Thread.currentThread().interrupt() was called. Use " +
                            "NioEventLoop.shutdownGracefully() to shutdown the NioE
                }
                selectCnt = 1;
                break;
            }
            // ------ 4. 解决臭名昭著的 JDK epoll 空轮询 Bug ------
            long time = System.nanoTime();
            if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeN
                selectCnt = 1;
            } else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
                    selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
                selector = selectRebuildSelector(selectCnt);
                selectCnt = 1;
                break;
            }
            currentTimeNanos = time;
        }
        if (selectCnt > MIN_PREMATURE_SELECTOR_RETURNS) {
            if (logger.isDebugEnabled()) {
                logger.debug("Selector.select() returned prematurely {} times in a
                        selectCnt - 1, selector);
            }
```

```
            }
        } catch (CancelledKeyException e) {
            if (logger.isDebugEnabled()) {
                logger.debug(CancelledKeyException.class.getSimpleName() + " raised by
                        selector, e);
            }
        }
    }
```

Netty 为了解决臭名昭著的 JDK epoll 空轮询 Bug，造成整个 select() 方法是相对比较复杂的，我把它划分成四个部分逐一拆解来看。

第一步，检测 select 阻塞操作是否超过截止时间。 在进入无限循环之前，Netty 首先记录了当前时间 currentTimeNanos 以及定时任务队列中最近待执行任务的执行时间 selectDeadLineNanos，Netty 中定时任务队列是按照延迟时间从小到大进行排列的，通过调用 delayNanos(currentTimeNanos) 方法可以获得第一个待执行定时任务的延迟时间。然后代码会进入无限循环。首先判断 currentTimeNanos 是否超过 selectDeadLineNanos 0.5ms 以上，如果超过说明当前任务队列中有定时任务需要立刻执行，所以此时会退出无限循环。退出之前如果从未执行过 select 操作，那么会立即一次非阻塞的 selectNow 操作。那么这里有一个疑问，为什么会留出 0.5ms 的时间窗口呢？在任务队列为空的情况下，可能 select 操作没有获得到任何 I/O 事件就立即停止阻塞返回。

其中有一点容易混淆，Netty 的任务队列包括普通任务、定时任务以及尾部任务，hasTask() 判断的是普通任务队列和尾部队列是否为空，而 delayNanos(currentTimeNanos) 方法获取的是定时任务的延迟时间。

第二步，轮询过程中及时处理产生的任务。 Netty 为了保证任务能够及时执行，会立即一次非阻塞的 selectNow 操作后，立即跳出循环回到事件循环的主流程，确保接下来能够优先执行 runAllTasks。

第三步，select 阻塞等待获取 I/O 事件。 执行 select 阻塞操作，说明任务队列已经为空，而且第一个待执行定时任务还没有到达任务执行的截止时间，需要阻塞等待 timeoutMillis 的超时时间。假设一种极端情况，如果定时任务的截止时间非常久，那么 select 操作岂不是会一直阻塞造成 Netty 无法工作？所以 Netty 在外部线程添加任务的时候，可以唤醒 select 阻塞操作，具体源码如下：

```
// SingleThreadEventExecutor#execute
```

```
public void execute(Runnable task) {
          // 省略其他代码
    if (!addTaskWakesUp && wakesUpForTask(task)) {
        wakeup(inEventLoop);
    }
}
// NioEventLoop#wakeup
protected void wakeup(boolean inEventLoop) {
    // 如果是外部线程，设置 wakenUp 为true，则唤醒 select 阻塞操作
    if (!inEventLoop && wakenUp.compareAndSet(false, true)) {
        selector.wakeup();
    }
}
```

selector.wakeup() 操作的开销是非常大的，所以 Netty 并不是每次都直接调用，在每次调用之前都会先执行 wakenUp.compareAndSet(false, true)，只有设置成功之后才会执行 selector.wakeup() 操作。

第四步，解决臭名昭著的 JDK epoll 空轮询 Bug。 在之前的课程中已经初步介绍了 Netty 的解决方案，在这里结合整体 select 操作我们再做一次回顾。实际上 Netty 并没有从根源上解决该问题，而是巧妙地规避了这个问题。Netty 引入了计数变量 selectCnt，用于记录 select 操作的次数，如果事件轮询时间小于 timeoutMillis，并且在该时间周期内连续发生超过 SELECTOR_AUTO_REBUILD_THRESHOLD（默认512） 次空轮询，说明可能触发了 epoll 空轮询 Bug。Netty 通过重建新的 Selector 对象，将异常的 Selector 中所有的 SelectionKey 会重新注册到新建的 Selector，重建完成之后异常的 Selector 就可以废弃了。

NioEventLoop 轮询 I/O 事件 select 的过程已经讲完了，我们简单总结 select 过程所做的事情。select 操作也是一个无限循环，在事件轮询之前检查任务队列是否为空，确保任务队列中待执行的任务能够及时执行。如果任务队列中已经为空，然后执行 select 阻塞操作获取等待获取 I/O 事件。Netty 通过引入计数器变量，并统计在一定时间窗口内 select 操作的执行次数，识别出可能存在异常的 Selector 对象，然后采用重建 Selector 的方式巧妙地避免了 JDK epoll 空轮询的问题。

处理 I/O 事件

通过 select 过程我们已经获取到准备就绪的 I/O 事件，接下来就需要调用 processSelectedKeys() 方法处理 I/O 事件。在开始处理 I/O 事件之前，Netty 通过 ioRatio 参数控制 I/O 事件处理和任务处理的时间比例，默认为 ioRatio = 50。如果 ioRatio = 100，表示每次都处理完 I/O 事件后，会执行所有的 task。如果 ioRatio < 100，也会优先处理完 I/O 事件，再处理异步任务队列。所以不论如何 processSelectedKeys() 都是先执行的，接下来跟进下 processSelectedKeys() 的源码：

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

处理 I/O 事件时有两种选择，一种是处理 Netty 优化过的 selectedKeys，另外一种是正常的处理逻辑。根据是否设置了 selectedKeys 来判断使用哪种策略，这两种策略使用的 selectedKeys 集合是不一样的。Netty 优化过的 selectedKeys 是 SelectedSelectionKeySet 类型，而正常逻辑使用的是 JDK HashSet 类型。下面我们逐一介绍两种策略的实现。

1. processSelectedKeysPlain

首先看下正常的处理逻辑 processSelectedKeysPlain 的源码：

```
private void processSelectedKeysPlain(Set<SelectionKey> selectedKeys) {
    if (selectedKeys.isEmpty()) {
        return;
    }
    Iterator<SelectionKey> i = selectedKeys.iterator();
    for (;;) {
        final SelectionKey k = i.next();
        final Object a = k.attachment();
        i.remove();
```

```
            if (a instanceof AbstractNioChannel) {
                // I/O 事件由 Netty 负责处理
                processSelectedKey(k, (AbstractNioChannel) a);
            } else {
                // 用户自定义任务
                @SuppressWarnings("unchecked")
                NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
                processSelectedKey(k, task);
            }
            if (!i.hasNext()) {
                break;
            }
            if (needsToSelectAgain) {
                selectAgain();
                selectedKeys = selector.selectedKeys();
                if (selectedKeys.isEmpty()) {
                    break;
                } else {
                    i = selectedKeys.iterator();
                }
            }
        }
    }
```

Netty 会遍历依次处理已经就绪的 SelectionKey，SelectionKey 上面可以挂载 attachment。再根据 attachment 属性可以判断 SelectionKey 的类型，SelectionKey 的类型可能是 AbstractNioChannel 和 NioTask，这两种类型对应的处理方式也是不同的，AbstractNioChannel 类型由 Netty 框架负责处理，NioTask 是用户自定义的 task，一般不会是这种类型。我们着重看下 AbstractNioChannel 的处理场景，跟进 processSelectedKey() 的源码：

```
private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {
    final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();
    if (!k.isValid()) { // 检查 Key 是否合法
        final EventLoop eventLoop;
        try {
            eventLoop = ch.eventLoop();
        } catch (Throwable ignored) {
            return;
        }
        if (eventLoop != this || eventLoop == null) {
            return;
        }
        unsafe.close(unsafe.voidPromise()); // Key 不合法，直接关闭连接
        return;
    }
    try {
        int readyOps = k.readyOps();
        // 处理连接事件
        if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
            int ops = k.interestOps();
            ops &= ~SelectionKey.OP_CONNECT;
            k.interestOps(ops);
            unsafe.finishConnect();
        }
        // 处理可写事件
        if ((readyOps & SelectionKey.OP_WRITE) != 0) {
            ch.unsafe().forceFlush();
        }
```

```
            // 处理可读事件
            if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || re
                unsafe.read();
            }
        } catch (CancelledKeyException ignored) {
            unsafe.close(unsafe.voidPromise());
        }
    }
```

从上述源码可知，processSelectedKey 一共处理了 OP_CONNECT、OP_WRITE、OP_READ 三个事件，我们分别了解下这三个事件的处理过程。

OP_CONNECT 连接建立事件。表示 TCP 连接建立成功, Channel 处于 Active 状态。处理 OP_CONNECT 事件首先将该事件从事件集合中清除，避免事件集合中一直存在连接建立事件，然后调用 unsafe.finishConnect() 方法通知上层连接已经建立。可以跟进 unsafe.finishConnect() 的源码发现会底层调用的 pipeline().fireChannelActive() 方法，这时会产生一个 Inbound 事件，然后会在 Pipeline 中进行传播，依次调用 ChannelHandler 的 channelActive() 方法，通知各个 ChannelHandler 连接建立成功。

- **OP_WRITE，可写事件。**表示上层可以向 Channel 写入数据，通过执行 ch.unsafe().forceFlush() 操作，将数据冲刷到客户端，最终会调用 javaChannel 的 write() 方法执行底层写操作。
- **OP_READ，可读事件。**表示 Channel 收到了可以被读取的新数据。Netty 将 READ 和 Accept 事件进行了统一的封装，都通过 unsafe.read() 进行处理。unsafe.read() 的逻辑可以归纳为几个步骤：从 Channel 中读取数据并存储到分配的 ByteBuf；调用 pipeline.fireChannelRead() 方法产生 Inbound 事件，然后依次调用 ChannelHandler 的 channelRead() 方法处理数据；调用 pipeline.fireChannelReadComplete() 方法完成读操作；最终执行 removeReadOp() 清除 OP_READ 事件。

我们再次回到 processSelectedKeysPlain 的主流程，接下来会判断 needsToSelectAgain 决定是否需要重新轮询。如果 needsToSelectAgain == true，会调用 selectAgain() 方法进行重新轮询，该方法会将 needsToSelectAgain 再次置为 false，然后调用 selectorNow() 后立即返回。

我们回顾一下 Reactor 线程的主流程，会发现每次在处理 I/O 事件之前，needsToSelectAgain 都会被设置为 false，那么在什么场景下 needsToSelectAgain 会

再次设置为 true 呢？我们通过查找变量的引用，最后定位到 AbstractChannel#doDeregister。该方法的作用是将 Channel 从当前注册的 Selector 对象中移除，方法内部可能会把 needsToSelectAgain 设置为 true，具体源码如下：

```
protected void doDeregister() throws Exception {
    eventLoop().cancel(selectionKey());
}
void cancel(SelectionKey key) {
    key.cancel();
    cancelledKeys ++;
    // 当取消的 Key 超过默认阈值 256，needsToSelectAgain 设置为 true
    if (cancelledKeys >= CLEANUP_INTERVAL) {
        cancelledKeys = 0;
        needsToSelectAgain = true;
    }
}
```

当 Netty 在处理 I/O 事件的过程中，如果发现超过默认阈值 256 个 Channel 从 Selector 对象中移除后，会将 needsToSelectAgai 设置为 true，重新做一次轮询操作，从而确保 keySet 的有效性。

2. processSelectedKeysOptimized

介绍完正常的 I/O 事件处理 processSelectedKeysPlain 之后，回过头我们再来分析 Netty 优化的 processSelectedKeysOptimized 就会轻松很多，Netty 是否采用 SelectedSelectionKeySet 类型的优化策略由 DISABLE_KEYSET_OPTIMIZATION 参数决定。那么到底 SelectedSelectionKeySet 是如何进行优化的呢？我们继续跟进下 processSelectedKeysOptimized 的源码：

```
private void processSelectedKeysOptimized() {
    for (int i = 0; i < selectedKeys.size; ++i) {
        final SelectionKey k = selectedKeys.keys[i];
        selectedKeys.keys[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);
            }
            if (needsToSelectAgain) {
                selectedKeys.reset(i + 1);
                selectAgain();
                i = -1;
            }
        }
    }
```

可以发现 processSelectedKeysOptimized 与 processSelectedKeysPlain 的代码结构非常相似，其中最重要的一点就是 selectedKeys 的遍历方式是不同的，所以还是需要看下 SelectedSelectionKeySet 的源码一探究竟。

```
final class SelectedSelectionKeySet extends AbstractSet<SelectionKey> {
    SelectionKey[] keys;
    int size;
    SelectedSelectionKeySet() {
        keys = new SelectionKey[1024];
    }
    @Override
    public boolean add(SelectionKey o) {
        if (o == null) {
            return false;
```

```
        }
        keys[size++] = o;
        if (size == keys.length) {
            increaseCapacity();
        }
        return true;
    }

    // 省略其他代码
}
```

因为 SelectedSelectionKeySet 内部使用的是 SelectionKey 数组，所以 processSelectedKeysOptimized 可以直接通过遍历数组取出 I/O 事件，相比 JDK HashSet 的遍历效率更高。SelectedSelectionKeySet 内部通过 size 变量记录数据的逻辑长度，每次执行 add 操作时，会把对象添加到 SelectionKey[] 尾部。当 size 等于 SelectionKey[] 的真实长度时，SelectionKey[] 会进行扩容。相比于 HashSet，SelectionKey[] 不需要考虑哈希冲突的问题，所以可以实现 O(1) 时间复杂度的 add 操作。

那么 SelectedSelectionKeySet 是什么时候生成的呢？通过查找 SelectedSelectionKeySet 的引用定位到 NioEventLoop#openSelector 方法，摘录核心源码片段如下：

```
private SelectorTuple openSelector() {
    // 省略其他代码
    final SelectedSelectionKeySet selectedKeySet = new SelectedSelectionKeySet();
    Object maybeException = AccessController.doPrivileged(new PrivilegedAction<Obje
        @Override
        public Object run() {
            try {
                Field selectedKeysField = selectorImplClass.getDeclaredField("selec
                Field publicSelectedKeysField = selectorImplClass.getDeclaredField(
                if (PlatformDependent.javaVersion() >= 9 && PlatformDependent.hasUn
```

```
                    long selectedKeysFieldOffset = PlatformDependent.objectFieldOff
                    long publicSelectedKeysFieldOffset =
                            PlatformDependent.objectFieldOffset(publicSelectedKeysF
                    if (selectedKeysFieldOffset != -1 && publicSelectedKeysFieldOff
                        PlatformDependent.putObject(
                                unwrappedSelector, selectedKeysFieldOffset, selecte
                        PlatformDependent.putObject(
                                unwrappedSelector, publicSelectedKeysFieldOffset, s
                        return null;
                    }
                }
                // 省略其他代码
            } catch (NoSuchFieldException e) {
                return e;
            } catch (IllegalAccessException e) {
                return e;
            }
        }
    });
    // 省略其他代码
}
```

Netty 通过反射的方式，将 Selector 对象内部的 selectedKeys 和 publicSelectedKeys 替换为 SelectedSelectionKeySet，原先 selectedKeys 和 publicSelectedKeys 这两个字段都是 HashSet 类型。这真是很棒的一个小技巧，对于 JDK 底层的优化一般是很少见的，Netty 在细节优化上追求极致的精神值得我们学习。

到这里，Reactor 线程主流程的第二步。处理 I/O 事件 processSelectedKeys 已经讲完了，简单总结一下 processSelectedKeys 的要点。处理 I/O 事件时有两种选择，一种是处理 Netty 优化过的 selectedKeys，另外一种是正常的处理逻辑，两种策略的处理逻辑是相似的，都是通过获取 SelectionKey 上挂载的 attachment 判断 SelectionKey 的类型，不同的

SelectionKey 的类型又会调用不同的处理方法，然后通过 Pipeline 进行事件传播。Netty 优化过的 selectedKeys 是使用数组存储的 SelectionKey，相比于 JDK 的 HashSet 遍历效率更高效。processSelectedKeys 还做了更多的优化处理，如果发现超过默认阈值 256 个 Channel 从 Selector 对象中移除后，会重新做一次轮询操作，以确保 keySet 的有效性。

处理异步任务队列

继续分析 Reactor 线程主流程的最后一步，处理异步任务队列 runAllTasks。为什么 Netty 能够保证 Channel 的操作都是线程安全的呢？这要归功于 Netty 的任务机制。下面我们从任务添加和任务执行两个方面介绍 Netty 的任务机制。

- **任务添加**

NioEventLoop 内部有两个非常重要的异步任务队列，分别为普通任务队列和定时任务队列。NioEventLoop 提供了 execute() 和 schedule() 方法用于向不同的队列中添加任务，execute() 用于添加普通任务，schedule() 方法用于添加定时任务。

首先我们看下如何添加普通任务。NioEventLoop 继承自 SingleThreadEventExecutor，SingleThreadEventExecutor 提供了 execute() 用于添加普通任务，源码如下：

```
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    boolean inEventLoop = inEventLoop();
    addTask(task);
    if (!inEventLoop) {
        startThread();
        if (isShutdown()) {
            boolean reject = false;
            try {
                if (removeTask(task)) {
                    reject = true;
                }
```

```
            } catch (UnsupportedOperationException e) {
            }
            if (reject) {
                reject();
            }
        }
    }
    if (!addTaskWakesUp && wakesUpForTask(task)) {
        wakeup(inEventLoop);
    }
}
protected void addTask(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    if (!offerTask(task)) {
        reject(task);
    }
}
final boolean offerTask(Runnable task) {
    if (isShutdown()) {
        reject();
    }
    return taskQueue.offer(task);
}
```

我们一步步跟进 addTask(task)，发现最后是将任务添加到了 taskQueue，SingleThreadEventExecutor 中 taskQueue 就是普通任务队列。taskQueue 默认使用的是 Mpsc Queue，可以理解为多生产者单消费者队列，关于 Mpsc Queue 我们

会有一节课程单独介绍，在这里不详细展开。此外，在任务处理的场景下，inEventLoop() 始终是返回 true，始终都是在 Reactor 线程内执行，既然在 Reactor 线程内都是串行执行，可以保证线程安全，那为什么还需要 Mpsc Queue 呢？我们继续往下看。

这里举一种很常见的场景，比如在 RPC 业务线程池里处理完业务请求后，可以根据用户请求拿到关联的 Channel，将数据写回客户端。那么对于外部线程调用 Channel 的相关方法 Netty 是如何操作的呢？我们一直跟进下 channel.write() 的源码：

```
// #AbstractChannel#write
public ChannelFuture write(Object msg) {
    return pipeline.write(msg);
}
// AbstractChannelHandlerContext#write
private void write(Object msg, boolean flush, ChannelPromise promise) {
    // 省略其他代码
    final AbstractChannelHandlerContext next = findContextOutbound(flush ?
            (MASK_WRITE | MASK_FLUSH) : MASK_WRITE);
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) { // Reactor 线程内部调用
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            next.invokeWrite(m, promise);
        }
    } else { // 外部线程调用会走到该分支
        final AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        }  else {
            task = WriteTask.newInstance(next, m, promise);
```

```
            }
            if (!safeExecute(executor, task, promise, m)) {
                task.cancel();
            }
        }
    }
    // AbstractChannelHandlerContext#safeExecute
    private static boolean safeExecute(EventExecutor executor, Runnable runnable, Chann
        try {
            executor.execute(runnable);
            return true;
        } catch (Throwable cause) {
            try {
                promise.setFailure(cause);
            } finally {
                if (msg != null) {
                    ReferenceCountUtil.release(msg);
                }
            }
            return false;
        }
    }
```

如果是 Reactor 线程发起调用 channel.write() 方法，inEventLoop() 返回 true，此时直接在 Reactor 线程内部直接交由 Pipeline 进行事件处理。如果是外部线程调用，那么会走到 else 分支，此时会将写操作封装成一个 WriteTask，然后通过 safeExecute() 执行，可以发现 safeExecute() 就是调用的 SingleThreadEventExecutor#execute() 方法，最终会将任务添加到 taskQueue 中。因为多个外部线程可能会并发操作同一个 Channel，这时候 Mpsc Queue 就可以保证线程的安全性。

接下来我们再分析定时任务的添加过程。与普通任务类似，定时任务也会有 Reactor 线程内和外部线程两种场景，我们直接跟进到 AbstractScheduledEventExecutor#schedule() 源码的深层，发现如下核心代码：

```
private <V> ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {
    if (inEventLoop()) { // Reactor 线程内部
        scheduledTaskQueue().add(task.setId(nextTaskId++));
    } else { // 外部线程
        executeScheduledRunnable(new Runnable() {
            @Override
            public void run() {
                scheduledTaskQueue().add(task.setId(nextTaskId++));
            }
        }, true, task.deadlineNanos());
    }
    return task;
}
PriorityQueue<ScheduledFutureTask<?>> scheduledTaskQueue() {
    if (scheduledTaskQueue == null) {
        scheduledTaskQueue = new DefaultPriorityQueue<ScheduledFutureTask<?>>(
                SCHEDULED_FUTURE_TASK_COMPARATOR,
                11);
    }
    return scheduledTaskQueue;
}
void executeScheduledRunnable(Runnable runnable,
                                        @SuppressWarnings("unused") boolean isAddit
                                        @SuppressWarnings("unused") long deadlineNa
    execute(runnable);
```

```
}
```

AbstractScheduledEventExecutor 中 scheduledTaskQueue 就是定时任务队列，可以看到 scheduledTaskQueue 的默认实现是优先级队列 DefaultPriorityQueue，这样可以方便队列中的任务按照时间进行排序。但是 DefaultPriorityQueue 是非线程安全的，如果是 Reactor 线程内部调用，因为是串行执行，所以不会有线程安全问题。如果是外部线程添加定时任务，我们发现 Netty 把添加定时任务的操作又再次封装成一个任务交由 executeScheduledRunnable() 处理，而 executeScheduledRunnable() 中又再次调用了普通任务的 execute() 的方法，巧妙地借助普通任务场景中 Mpsc Queue 解决了外部线程添加定时任务的线程安全问题。

- **任务执行**

介绍完 Netty 中不同任务的添加过程，回过头我们再来分析 Reactor 线程是如何执行这些任务的呢？通过 Reactor 线程主流程的分析，我们知道处理异步任务队列有 runAllTasks() 和 runAllTasks(long timeoutNanos) 两种实现，第一种会处理所有任务，第二种是带有超时时间来处理任务。之所以设置超时时间是为了防止 Reactor 线程处理任务时间过长而导致 I/O 事件阻塞，我们着重分析下 runAllTasks(long timeoutNanos) 的源码：

```
protected boolean runAllTasks(long timeoutNanos) {
    fetchFromScheduledTaskQueue(); // 1. 合并定时任务到普通任务队列
    // 2. 从普通任务队列中取出任务并处理
    Runnable task = pollTask();
    if (task == null) {
        afterRunningAllTasks();
        return false;
    }
    // 计算任务处理的超时时间
    final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;
    long runTasks = 0;
    long lastExecutionTime;
    for (;;) {
        safeExecute(task); // 执行任务
        runTasks ++;
```

```
            // 每执行 64 个任务检查一下是否超时
            if ((runTasks & 0x3F) == 0) {
                lastExecutionTime = ScheduledFutureTask.nanoTime();
                if (lastExecutionTime >= deadline) {
                    break;
                }
            }
            task = pollTask(); // 继续取出下一个任务
            if (task == null) {
                lastExecutionTime = ScheduledFutureTask.nanoTime();
                break;
            }
        }
        // 3. 收尾工作
        afterRunningAllTasks();
        this.lastExecutionTime = lastExecutionTime;
        return true;
    }
```

异步任务处理 runAllTasks 的过程可以分为三步：合并定时任务到普通任务队列，然后从普通任务队列中取出任务并处理，最后进行收尾工作。我们分别看看三个步骤都是如何实现的。

第一步，合并定时任务到普通任务队列，对应的实现是 fetchFromScheduledTaskQueue() 方法。

```
    private boolean fetchFromScheduledTaskQueue() {
        if (scheduledTaskQueue == null || scheduledTaskQueue.isEmpty()) {
            return true;
        }
```

```
        long nanoTime = AbstractScheduledEventExecutor.nanoTime();
        for (;;) {
            Runnable scheduledTask = pollScheduledTask(nanoTime); // 从定时任务队列中取出
            if (scheduledTask == null) {
                return true;
            }
            if (!taskQueue.offer(scheduledTask)) {
                // 如果普通任务队列已满，把定时任务放回
                scheduledTaskQueue.add((ScheduledFutureTask<?>) scheduledTask);
                return false;
            }
        }
    }
    protected final Runnable pollScheduledTask(long nanoTime) {
        assert inEventLoop();
        Queue<ScheduledFutureTask<?>> scheduledTaskQueue = this.scheduledTaskQueue;
        ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == null ? null : sche
        // 如果定时任务的 deadlineNanos 小于当前时间就取出
        if (scheduledTask == null || scheduledTask.deadlineNanos() - nanoTime > 0) {
            return null;
        }
        scheduledTaskQueue.remove();
        return scheduledTask;
    }
```

定时任务只有满足截止时间 deadlineNanos 小于当前时间，才可以取出合并到普通任务。由于定时任务是按照截止时间 deadlineNanos 从小到大排列的，所以取出的定时任务不满足合并条件，那么定时任务队列中剩下的所有任务都不会满足条件，合并操作完成并退出。

第二步，从普通任务队列中取出任务并处理，可以回过头再看 runAllTasks(long

timeoutNanos) 第二部分的源码，我已经用注释标明。真正处理任务的 safeExecute() 非常简单，就是直接调用的 Runnable 的 run() 方法。因为异步任务处理是有超时时间的，所以 Netty 采取了定时检测的策略，每执行 64 个任务的时候就会检查一下是否超时，这也是出于对性能的折中考虑，如果异步队列中有大量的短时间任务，每一次执行完都检测一次超时性能会有所降低。

第三步，收尾工作，对应的是 afterRunningAllTasks() 方法实现。

```
protected void afterRunningAllTasks() {
    runAllTasksFrom(tailTasks);
}
protected final boolean runAllTasksFrom(Queue<Runnable> taskQueue) {
    Runnable task = pollTaskFrom(taskQueue);
    if (task == null) {
        return false;
    }
    for (;;) {
        safeExecute(task);
        task = pollTaskFrom(taskQueue);
        if (task == null) {
            return true;
        }
    }
}
```

这里的尾部队列 tailTasks 相比于普通任务队列优先级较低，可以理解为是收尾任务，在每次执行完 taskQueue 中任务后会去获取尾部队列中任务执行。可以看出 afterRunningAllTasks() 就是把尾部队列 tailTasks 里的任务以此取出执行一遍。尾部队列并不常用，一般用于什么场景呢？例如你想对 Netty 的运行状态做一些统计数据，例如任务循环的耗时、占用物理内存的大小等等，都可以向尾部队列添加一个收尾任务完成统计数据的实时更新。

到这里，Netty 处理异步任务队列的流程就讲完了，再做一个简单的总结。异步任务主要分

为普通任务和定时任务两种，在任务添加和任务执行时，都需要考虑 Reactor 线程内和外部线程两种情况。外部线程添加定时任务时，Netty 巧妙地借助普通任务的 Mpsc Queue 解决多线程并发操作时的线程安全问题。Netty 执行任务之前会将满足条件的定时任务合并到普通任务队列，由普通任务队列统一负责执行，并且每执行 64 个任务的时候就会检查一下是否超时。

总结

Reactor 线程模型是 Netty 最核心的内容，本节课我也花了大量的篇幅对其进行讲解。NioEventLoop 作为 Netty Reactor 线程的实现，它的设计原理是非常精妙的，值得我们反复阅读和思考。我们始终需要记住 NioEventLoop 的无限循环中所做的三件事：轮询 I/O 事件，处理 I/O 事件，处理异步任务队列。

关于 Netty Reactor 线程模型经常会遇到几个高频的面试问题，读完本节课之后你是否都已经清楚了呢？

- Netty 的 NioEventLoop 是如何实现的？它为什么能够保证 Channel 的操作是线程安全的？
- Netty 如何解决 JDK epoll 空轮询 Bug？
- NioEventLoop 是如何实现无锁化的？

欢迎你在评论区留言，期待看到你分享关于 Reactor 线程模型更多的认识和思考。

上一页 下一页