- NETTY

精尽 Netty 源码解析 —— EventLoop（四）之 EventLoop 运行

1. 概述

本文我们分享 EventLoop 的**运行**相关代码的实现。

因为 EventLoop 的**运行**主要是通过 NioEventLoop 的 `#run()` 方法实现，考虑到内容相对的完整性，在《精尽 Netty 源码解析 —— EventLoop（三）之 EventLoop 初始化》一文中，我们并未分享 NioEventLoop 的**初始化**，所以本文也会分享这部分的内容。

OK，还是老样子，自上而下的方式，一起来看看 NioEventLoop 的代码实现。

> 老芳芳，本文的重点在「2.9 run」和「2.12 select」中。

2. NioEventLoop

`io.netty.channel.nio.NioEventLoop`，继承 SingleThreadEventLoop 抽象类，NIO EventLoop 实现类，实现对注册到其中的 Channel 的就绪的 IO 事件，和对用户提交的任务进行处理。

2.1 static

在 `static` 代码块中，初始化了 NioEventLoop 的静态属性们。代码如下：

```
/**
 * TODO 1007 NioEventLoop cancel
 */
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.noK

/**
                                                    tor 对象的功能

                                                    ELECTOR_RETURNS = 3;
```

文章目录

```
Selector 对象

EBUILD_THRESHOLD;

vel";
tyUtil.get(key);

d(new PrivilegedAction<Void>() {

ey, "");
                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.selectorAutoRebuildThreshol
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。详细解析，见《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》。
- `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。详细解析，见 http://bugs.sun.com/view_bug.do?bug_id=6427854 和 https://github.com/netty/netty/issues/203 。
- `<2>` 处，初始化 `SELECTOR_AUTO_REBUILD_THRESHOLD` 属性。默认 512 。

2.2 构造方法

```
/**
 * The NIO {@link Selector}.
```

现，经过优化。

Keys;

or 对象

```
private final SelectorProvider provider;

/**
 * Boolean that controls determines if a blocked Selector.select should
 * break out of its selection process. In our case we use a timeout for
 * the select method and the select method will block for that time unless
 * waken up.
 *
 * 唤醒标记。因为唤醒方法 {@link Selector#wakeup()} 开销比较大，通过该标识，减少调用。
 *
 * @see #wakeup(boolean)
 * @see #run()
 */
private final AtomicBoolean wakenUp = new AtomicBoolean();
/**
 * Select 策略
 *
 * @see #select(boolean)
 */
private final SelectStrategy selectStrategy;
/**
 * 处理 Channel 的就绪的 IO 事件，占处理任务的总时间的比例
 */
private volatile int ioRatio = 50;
/**
 * 取消 SelectionKey 的数量
 *
 * TODO 1007 NioEventLoop cancel
 */
private int cancelledKeys;
/**
 * 是否需要再次 select Selector 对象
 *
 * TODO 1007 NioEventLoop cancel
 */
private boolean needsToSelectAgain;
```

xecutor executor, SelectorProvider selectorProvider,
jectedExecutionHandler rejectedExecutionHandler) {
LT_MAX_PENDING_TASKS, rejectedExecutionHandler);

```
'selectorProvider");

'selectStrategy");

openSelector();

wrappedSelector;
```

- Selector 相关：
  - `unwrappedSelector` 属性，未包装的 NIO Selector 对象。
  - `selector` 属性，包装的 NIO Selector 对象。Netty 对 NIO Selector 做了优化。详细解析，见《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》。
  - `selectedKeys` 属性，注册的 NIO SelectionKey 集合。Netty 自己实现，经过优化。详细解析，见《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》。
  - `provider` 属性，NIO SelectorProvider 对象，用于创建 NIO Selector 对象。
  - 在 `<1>` 处，调用 `#openSelector()` 方法，创建 NIO Selector 对象。
- `wakenUp` 属性，唤醒标记。因为唤醒方法 `Selector#wakeup()` 开销比较大，通过该标识，减少调用。详细解析，见「2.8 wakeup」。
- `selectStrategy` 属性，Select 策略。详细解析，见「2.10 SelectStrategy」。
- `ioRatio` 属性，在 NioEventLoop 中，会三种类型的任务：1) Channel 的就绪的 IO 事件；2) 普通任务；3) 定时任务。而 `ioRatio` 属性，处理 Channel 的就绪的 IO 事件，占处理任务的总时间的比例。
- 取消 SelectionKey 相关：
  - `cancelledKeys` 属性， 取消 SelectionKey 的数量。TODO 1007 NioEventLoop cancel
  - `needsToSelectAgain` 属性，是否需要再次 select Selector 对象。TODO 1007 NioEventLoop cancel

2.3 openSelector

`#openSelector()` 方法，创建 NIO Selector 对象。

考虑到让本文更专注在 EventLoop 的逻辑，并且不影响对本文的理解，所以暂时不讲解它的具体实现。详细解析，见《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》。

2.4 rebuildSelector

`#rebuildSelector()` 方法，重建 NIO Selector 对象。

考虑到让本文更专注在 EventLoop 的逻辑，并且不影响对本文的理解，所以暂时不讲解它的具体实现。详细解析，见《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》。

2.5 newTaskQueue

`#newTaskQueue(int maxPendingTasks)` 方法，创建任务队列。代码如下：

文章目录

```
nt maxPendingTasks) {
ask()
X_VALUE ? PlatformDependent.<Runnable>newMpscQueue()
        : PlatformDependent.<Runnable>newMpscQueue(maxPendingT
```

.) 方法，创建 mpsc 队列。我们来看看代码注释对 mpsc 队列的描

```
afe to use for multiple producers (different threads) and a s
```

- mpsc 是 multiple producers and a single consumer 的缩写。
- mpsc 是对**多**线程生产任务，**单**线程消费任务的消费，恰好符合 NioEventLoop 的情况。
- 详细解析，见后续文章。当然，着急的胖友，可以先看看 《原理剖析（第 012 篇）Netty 之无锁队列 MpscUnboundedArrayQueue 原理分析》 。

2.6 pendingTasks

`#pendingTasks()` 方法，获得待执行的任务数量。代码如下：

> 该方法覆写父类的该方法。

```
@Override
public int pendingTasks() {
    // As we use a MpscQueue we need to ensure pendingTasks() is only executed from within the EventLo
    // otherwise we may see unexpected behavior (as size() is only allowed to be called by a single co
    // See https://github.com/netty/netty/issues/5297
    if (inEventLoop()) {
        return super.pendingTasks();
    } else {
        return submit(pendingTasksCallable).syncUninterruptibly().getNow();
    }
}
```

- 因为 MpscQueue 仅允许单消费，所以获得队列的大小，仅允许在 EventLoop 的线程中调用。

2.7 setIoRatio

`#setIoRatio(int ioRatio)` 方法，设置 `ioRatio` 属性。代码如下：

```
/**
 * Sets the percentage of the desired amount of time spent for I/O in the event loop.  The default val
```

```
p will try to spend the same amount of time for I/O as for non
```

```
on("ioRatio: " + ioRatio + " (expected: 0 < ioRatio <= 100)");
```

线程。代码如下：

```
p) {
ndSet(false, true)) { // <2>
        selector.wakeup(); // <1>
    }
}
```

- <1> 处，因为 NioEventLoop 的线程阻塞，主要是调用 Selector#select(long timeout) 方法，阻塞等待有 Channel 感兴趣的 IO 事件，或者超时。所以需要调用 Selector#wakeup() 方法，进行唤醒 Selector 。
- <2> 处，因为 Selector#wakeup() 方法的唤醒操作是开销比较大的操作，并且每次重复调用相当于重复唤醒。所以，通过 wakenUp 属性，通过 CAS 修改 false => true ，保证有且仅有进行一次唤醒。
- 当然，详细的解析，可以结合 「2.9 run」 一起看，这样会更加清晰明了。

2.9 run

#run() 方法，NioEventLoop 运行，处理任务。**这是本文最重要的方法。**代码如下：

```
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
                       select(...)'. (BAD)
                       s waken up between 'selector.select(...)' and
                       Up.get()) { ... }'. (OK)
```

```
case, 'wakenUp' is set to true and the
lector.select(...)' will wake up immediately.
p' is set to false again in the next round,
areAndSet(false, true)' will fail, and therefore
o wake up the Selector will fail, too, causing
 'selector.select(...)' call to block
.

roblem, we wake up the selector again if wakenUp
iately after selector.select(...).
ient in that it wakes up the selector for both
e (BAD - wake-up required) and the second case
-up required).
                    // 唤醒。原因，见上面中文注释
                    if (wakenUp.get()) {
                        selector.wakeup();
                    }
                    // fall through
                default:
            }

            // TODO 1007 NioEventLoop cancel 方法
            cancelledKeys = 0;
            needsToSelectAgain = false;

            final int ioRatio = this.ioRatio;
            if (ioRatio == 100) {
                try {
                    // 处理 Channel 感兴趣的就绪 IO 事件
                    processSelectedKeys();
                } finally {
                    // 运行所有普通任务和定时任务，不限制时间
                    // Ensure we always run tasks.
                    runAllTasks();
                }
            } else {
                final long ioStartTime = System.nanoTime();
                try {
                    // 处理 Channel 感兴趣的就绪 IO 事件
                    processSelectedKeys();
                } finally {
                    // 运行所有普通任务和定时任务，限制时间
                    // Ensure we always run tasks.
                    final long ioTime = System.nanoTime() - ioStartTime;
                    runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
                }
            }
        } catch (Throwable t) {
            handleLoopException(t);
```



```
关闭
en if the loop processing threw an exception.
```

```
()) {
```

- 第 3 行：“死”循环，直到 NioEventLoop 关闭，即【第 78 至 89 行】的代码。
- 第 5 行：调用 `SelectStrategy#calculateStrategy(IntSupplier selectSupplier, boolean hasTasks)` 方法，获得使用的 select 策略。详细解析，胖友先跳到 「2.10 SelectStrategy」 中研究。😈 看完回来。
  - 我们知道 `SelectStrategy#calculateStrategy(...)` 方法，有 3 种返回的情况。
  - 第 6 至 7 行：第一种， `SelectStrategy.CONTINUE` ，默认实现下，不存在这个情况。
  - 第 8 至 44 行：第二种， `SelectStrategy.SELECT` ，进行 Selector **阻塞** select 。
    - 第 11 行：重置 `wakeUp` 标识为 `false` ，并返回修改前的值。
    - 第 11 行：调用 `#select(boolean oldWakeUp)` 方法，选择(查询)任务。直接看这个方法不能完全表达出该方法的用途，所以详细解析，见 「2.12 select」 。
    - 第 41 至 44 行：若唤醒标识 `wakeup` 为 `true` 时，调用 `Selector#wakeup()` 方法，唤醒 Selector 。可能看到此处，很多胖友会和我一样，一脸懵逼。实际上，**耐下性子**，答案在上面的**英文注释**中。笔者来简单解析下：
      - 1）在 `wakenUp.getAndSet(false)` 和 `#select(boolean oldWakeUp)` 之间，在标识 `wakeUp` 设置为 `false` 时，在 `#select(boolean oldWakeUp)` 方法中，正在调用 `Selector#select(...)` 方法，处于**阻塞**中。
      - 2）此时，有另外的线程调用了 `#wakeup()` 方法，会将标记 `wakeUp` 设置为 `true` ，并**唤醒** `Selector#select(...)` 方法的阻塞等待。
      - 3）标识 `wakeUp` 为 `true` ，所以再有另外的线程调用 `#wakeup()` 方法，都无法唤醒 `Selector#select(...)` 。为什么呢? 因为 `#wakeup()` 的 CAS 修改 `false => true` 会**失败**，导致无法调用 `Selector#wakeup()` 方法。
      - 解决方式：所以在 `#select(boolean oldWakeUp)` 执行完后，增加了【第 41 至 44 行】来解决。
      - 😈😈😈 整体比较绕，胖友结合实现代码 + 英文注释，再好好理解下。
  - 第 46 行：第三种， `>= 0` ，已经有可以处理的任务，直接向下。
- 第 49 至 51 行：TODO 1007 NioEventLoop cancel 方法
- 第 53 至 74 行：根据 `ioRatio` 的配置不同，分成**略有差异**的 2 种：
  - 第一种， `ioRatio` 为 100 ，则**不考虑**时间占比的分配。
    - 第 57 行：调用 `#processSelectedKeys()` 方法，处理 Channel 感兴趣的就绪 IO 事件。详细解析，见 《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》 。
    - 第 58 至 62 行：调用 `#runAllTasks()` 方法，运行所有普通任务和定时任务，**不限制时间**。详细解析，见 《精尽 Netty 源码解析 —— EventLoop（五）之 EventLoop 处理 IO 事件》 。
  - 第二种， `ioRatio` 为 `< 100` ，则**考虑**时间占比的分配。
    - 第 64 行：记录当前时间。
    - 第 67 行：和【第 57 行】的代码**一样**。
    - 第 71 至 72 行：🙂 比较巧妙的方式，是不是和胖友之前认为的不太一样。它是以 `#processSelectedKeys()` 方法的执行时间作为**基准**，计算 `#runAllTasks(long timeoutNanos)` 方法可执行的时间。
    - 第 72 行：调用 #runAllTasks(long timeoutNanos)` 方法，运行所有普通任务和定时任务，**限制时间**。

`eLoopException(Throwable t)` 方法，处理异常。代码如下：

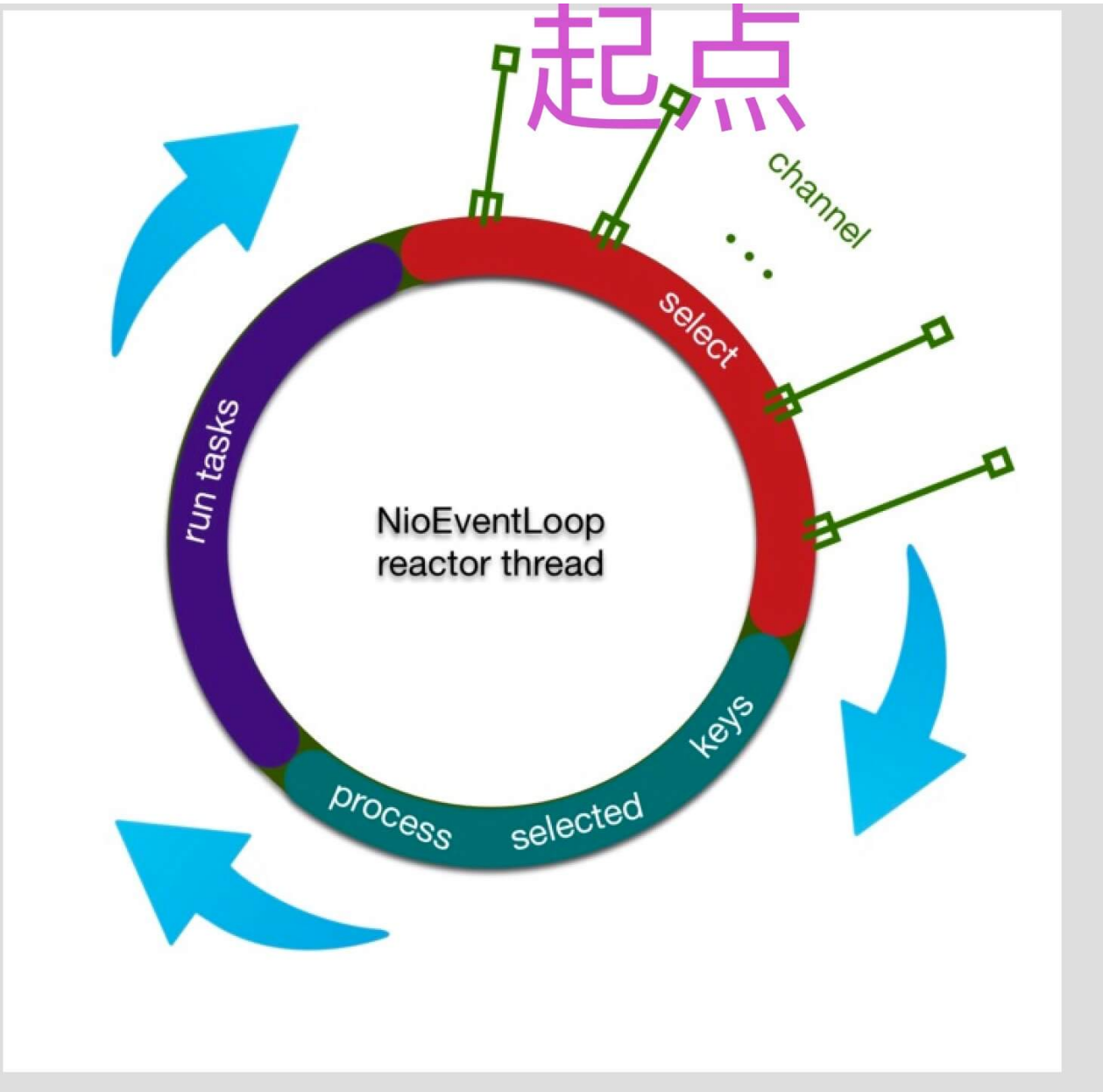
```
on(Throwable t) {
 in the selector loop.", t);

mediate failures that lead to

{
```

闭

- 总的来说， #run() 的执行过程，就是如下一张图：



2.10 SelectStrategy

select)策略接口。代码如下：

```
d follow.

etried, no blocking select to follow directly.
```

```
    /**
     * The {@link SelectStrategy} can be used to steer the outcome of a potential select
     * call.
     *
     * @param selectSupplier The supplier with the result of a select result.
     * @param hasTasks true if tasks are waiting to be processed.
     * @return {@link #SELECT} if the next step should be blocking select {@link #CONTINUE} if
     *         the next step should be to not select but rather jump back to the IO loop and try
     *         again. Any value >= 0 is treated as an indicator that work needs to be done.
     */
    int calculateStrategy(IntSupplier selectSupplier, boolean hasTasks) throws Exception;

}
```

- `calculateStrategy(IntSupplier selectSupplier, boolean hasTasks)` 接口方法有 **3** 种返回的情况：
  - `SELECT` ， `-1` ，表示使用阻塞 **select** 的策略。
  - `CONTINUE` ， `-2` ，表示需要进行重试的策略。实际上，默认情况下，不会返回 `CONTINUE` 的策略。
  - `>= 0` ，表示不需要 select ，目前已经有可以执行的任务了。

2.10.1 DefaultSelectStrategy

`io.netty.channel.DefaultSelectStrategy` ，实现 SelectStrategy 接口，默认选择策略实现类。代码如下：

```
ents SelectStrategy {

= new DefaultSelectStrategy();

lier selectSupplier, boolean hasTasks) throws Exception {
.get() : SelectStrategy.SELECT;

}
```

- 当 `hasTasks` 为 `true` ，表示当前已经有任务，所以调用 `IntSupplier#get()` 方法，返回当前 Channel 新增的 IO 就绪事件的数量。代码如下：

```
private final IntSupplier selectNowSupplier = new IntSupplier() {
    @Override
    public int get() throws Exception {
        return selectNow();
    }
};
```

  - `io.netty.util.IntSupplier` ，代码如下：

```
public interface IntSupplier {

    /**
     * Gets a result.
     *
     * @return a result
     */
    int get() throws Exception;

}
```

    - 类似 Java 自带的 `Callable<Int>` 。
  - IntSupplier 在 NioEventLoop 中的实现为 `selectNowSupplier` 属性。在它的内部会调用 `#selectNow()` 方法。详细解析，见「2.11 selectNow」。
  - 实际上，这里不调用 `IntSupplier#get()` 方法，也是可以的。只不过考虑到，可以通过 `#selectNow()` 方法，**无阻塞**的 select Channel 是否有感兴趣的就绪事件。
- 当 `hasTasks` 为 `false` 时，直接返回 `SelectStrategy.SELECT` ，进行**阻塞** select Channel 感兴趣的就绪 IO 事件。

2.11 selectNow

`#selectNow()` 方法，代码如下：

<1>

d <2>

方法，立即(**无阻塞**)返回 Channel 新增的感兴趣的就绪 IO 事件数量。

，调用 `Selector#wakeup()` 方法，唤醒 Selector 。因为 `<1>` 处的
lector 的唤醒，所以需要进行**复原**。有一个冷知道，可能有胖友不知道：

> 注意，如果有其它线程调用了 `#wakeup()` 方法，但当前没有线程阻塞在 `#select()` 方法上，下个调用 `#select()` 方法的线程会立即被唤醒。
> 😈 有点神奇。

2.12 select

`#select(boolean oldWakenUp)` 方法，选择(查询)任务。**这是本文最重要的方法。**代码如下：

```
1: private void select(boolean oldWakenUp) throws IOException {
2:     // 记录下 Selector 对象
3:     Selector selector = this.selector;
4:     try {
5:         // select 计数器
6:         int selectCnt = 0; // cnt 为 count 的缩写
7:         // 记录当前时间，单位：纳秒
8:         long currentTimeNanos = System.nanoTime();
9:         // 计算 select 截止时间，单位：纳秒。
10:        long selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);
11:
12:        for (;;) {
13:            // 计算本次 select 的超时时长，单位：毫秒。
14:            // + 500000L 是为了四舍五入
15:            // / 1000000L 是为了纳秒转为毫秒
16:            long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;
17:            // 如果超时时长，则结束 select
18:            if (timeoutMillis <= 0) {
19:                if (selectCnt == 0) { // 如果是首次 select ，selectNow 一次，非阻塞
20:                    selector.selectNow();
21:                    selectCnt = 1;
22:                }
23:                break;
24:            }
```


ted when wakenUp value was true, the task didn't get a chance
we need to check task queue again before executing select oper
k might be pended until select operation was timed out.

```
ntil idle timeout if IdleStateHandler existed in pipeline.

Up.compareAndSet(false, true)) {
非阻塞
);
器

annel 是否有就绪的 IO 事件
ctor.select(timeoutMillis);
```

```
44:             // 结束 select ，如果满足下面任一一个条件
45:             if (selectedKeys != 0 || oldWakenUp || wakenUp.get() || hasTasks() || hasScheduledTas
46:                 // - Selected something,
47:                 // - waken up by user, or
48:                 // - the task queue has a pending task.
49:                 // - a scheduled task is ready for processing
50:                 break;
51:             }
52:             // 线程被打断。一般情况下不会出现，出现基本是 bug ，或者错误使用。
53:             if (Thread.interrupted()) {
54:                 // Thread was interrupted so reset selected keys and break so we not run into a b
55:                 // As this is most likely a bug in the handler of the user or it's client library
56:                 // also log it.
57:                 //
58:                 // See https://github.com/netty/netty/issues/2426
59:                 if (logger.isDebugEnabled()) {
60:                     logger.debug("Selector.select() returned prematurely because " +
61:                             "Thread.currentThread().interrupt() was called. Use " +
62:                             "NioEventLoop.shutdownGracefully() to shutdown the NioEventLoop.");
63:                 }
64:                 selectCnt = 1;
65:                 break;
66:             }
67:
68:             // 记录当前时间
69:             long time = System.nanoTime();
70:             // 符合 select 超时条件，重置 selectCnt 为 1
71:             if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos) {
72:                 // timeoutMillis elapsed without anything selected.
73:                 selectCnt = 1;
74:             // 不符合 select 超时的提交，若 select 次数到达重建 Selector 对象的上限，进行重建
75:             } else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
76:                     selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
77:                 // The selector returned prematurely many times in a row.
78:                 // Rebuild the selector to work around the problem.
79:                 logger.warn("Selector.select() returned prematurely {} times in a row; rebuilding
```

象

```
对象
ector;

populate selectedKeys.
次，非阻塞
);
为 1

;
```

```
 98:         if (selectCnt > MIN_PREMATURE_SELECTOR_RETURNS) {
 99:             if (logger.isDebugEnabled()) {
100:                 logger.debug("Selector.select() returned prematurely {} times in a row for Select
101:             }
102:         }
103:     } catch (CancelledKeyException e) {
104:         if (logger.isDebugEnabled()) {
105:             logger.debug(CancelledKeyException.class.getSimpleName() + " raised by a Selector {}
106:         }
107:         // Harmless exception - log anyway
108:     }
109: }
```

- 第 3 行：获得使用的 Selector 对象，不需要每次访问使用 `volatile` 修饰的 `selector` 属性。
- 第 6 行：获得 select 操作的计数器。主要用于记录 Selector 空轮询次数，所以每次在正在轮询完成(例如：轮询超时)，则重置 `selectCnt` 为 1 。
- 第 8 行：记录当前时间，单位：纳秒。
- 第 10 行：计算 select 操作的截止时间，单位：纳秒。
  - `#delayNanos(currentTimeNanos)` 方法返回的为下一个定时任务距离现在的时间，如果不存在定时任务，则默认返回 1000 ms 。该方法的详细解析，见后续文章。
- 第 12 行：“死”循环，直到符合如下**任一**一种情况后**结束**：
  1. select 操作超时，对应【第 18 至 24 行】。
  2. 若有新的任务加入，对应【第 26 至 37 行】。
  3. 查询到任务或者唤醒，对应【第 45 至 51 行】。
  4. 线程被异常打断，对应【第 52 至 66 行】。
  5. 发生 NIO 空轮询的 Bug 后重建 Selector 对象后，对应【第 75 至 93 行】。
- 第 16 行：计算本次 select 的**超时时长**，单位：毫秒。因为【第 40 行】的 `Selector#select(timeoutMillis)` 方法，可能因为**各种情况结束**，所以需要循环，并且每次**重新**计算超时时间。至于 `+ 500000L` 和 `/ 1000000L` 的用途，看下代码注释。
- 第 17 至 24 行：如果超过 select 超时时长，则结束 select 。
  - 第 19 至 21 行：如果是首次 select ，则调用 `Selector#selectNow()` 方法，获得**非阻塞**的 Channel 感兴趣的就绪的 IO 事件，并重置 `selectCnt` 为 1 。
- 第 26 至 37 行：若有新的任务加入。这里实际要分成两种情况：
  - 第一种，提交的任务的类型是 NonWakeupRunnable ，那么它并不会调用 `#wakeup()` 方法，原因胖友自己看 `#execute(Runnable task)` 思考下。Netty 在 `#select()` 方法的设计上，**能尽快执行任务**。此时如果标记 直接结束 select 。
  - Runnable ，那么在 `#run()` 方法的【第 8 至 11 行】的 了一次 `#wakeup()` 方法，那么因为

wakeUp 设置为 false ，所以就能满足 hasTasks() &&
的条件。
注释 So we need to check task queue again before
don't, the task might be pended until select
？这是为什么呢？因为 Selector 被提前 wakeup 了，所以下一次

Selector#selectNow() 方法，**非阻塞**的获取一次 Channel 新增

t/f44f3e7926f1676315ae86d0f18bdd9b95681d9f 。
illis) 方法，**阻塞** select ，获得 Channel 新增的就绪的 IO 事件的数

结束 select ：
l 新增的就绪的 IO 事件，所以结束 select ，很好理解。
2. oldWakenUp || wakenUp.get() 时，表示 Selector 被唤醒，所以结束 select 。
3. hasTasks() || hasScheduledTasks() ，表示有普通任务或定时任务，所以结束 select 。
4. 那么剩余的情况，主要是 select **超时**或者发生**空轮询**，即【第 68 至 93 行】的代码。

- 第 52 至 66 行：线程被打断。一般情况下不会出现，出现基本是 **bug** ，或者错误使用。感兴趣的胖友，可以看看 https://github.com/netty/netty/issues/2426 。
- 第 69 行：记录当前时间。
  - 第 70 至 73 行：若满足 time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos ，说明到达此处时，Selector 是**超时** select ，那么是**正常**的，所以重置 selectCnt 为 1 。
  - 第 74 至 93 行：不符合 select 超时的提交，若 select 次数到达重建 Selector 对象的上限，进行重建。**这就是 Netty 判断发生 NIO Selector 空轮询的方式**，N (默认 512)次 select 并未阻塞超时这么长，那么就认为发生 NIO Selector 空轮询。过多的 NIO Selector 将会导致 CPU 100% 。
    - 第 82 行：调用 #rebuildSelector() 方法，重建 Selector 对象。
    - 第 84 行：**重新**获得使用的 Selector 对象。
    - 第 86 至 90 行：同【第 20 至 21 行】的代码。
    - 第 92 行：结束 select 。
- 第 95 行：记录新的当前时间，用于【第 16 行】，**重新**计算本次 select 的超时时长。

666. 彩蛋

总的来说还是比较简单的，比较困难的，在于对标记 wakeup 的理解。真的是，细思极恐！！！感谢在理解过程中，闪电侠和大表弟普架的帮助。

推荐阅读文章：

- 闪电侠 《netty 源码分析之揭开 reactor 线程的面纱（一）》
- Hypercube 《自顶向下深入分析 Netty（四）–EventLoop-2》

> 老艿艿：全文的 NIO Selector 空轮询，指的是 epoll cpu 100% 的 bug 。