文章目录

— EventLoop（七）之 EventLoop 处

（六）之 EventLoop 处理普通任务》，分享【处理**定时任务**】的部分。

竭尽 Netty 源码解析 —— EventLoop（三）之 EventLoop 初始化》并未分享，文先写这部分内容。

sk

tureTask ，实现 ScheduledFuture、PriorityQueueNode 接口，继承 PromiseTask 抽象类，Netty 定时任务。

> 老艿艿：也有文章喜欢把“定时任务”叫作“调度任务”，意思是相同的，本文统一使用“定时任务”。

2.1 静态属性

```
/**
 * 任务序号生成器，通过 AtomicLong 实现递增发号
 */
private static final AtomicLong nextTaskId = new AtomicLong();
/**
 * 定时任务时间起点
 */
private static final long START_TIME = System.nanoTime();
```

- `nextTaskId` 静态属性，任务序号生成器，通过 AtomicLong 实现**递增**发号。
- `START_TIME` 静态属性，定时任务时间**起点**。在 ScheduledFutureTask 中，定时任务的执行时间，都是基于 `START_TIME` 做**相对**时间。😈 至于为什么使用相对时间？笔者暂时没有搞清楚。
  - 笔者也搜索了下和 `System.nanoTime()` 相关的内容，唯一能看的是《System.nanoTime() 的隐患》，但是应该不是这个原因。
  - 和我的大表弟普架交流了一波，他的理解是：

文章目录

了系统时间也没关系

要多长时间

个是相对 START_TIME 来算的。代码如下：

```
TIME;
```

到它。

获得任务执行时间，这个也是相对 START_TIME 来算的。代码如下：

```
/**
 * @param delay 延迟时长，单位：纳秒
 * @return 获得任务执行时间，也是相对 {@link #START_TIME} 来算的。
 *          实际上，返回的结果，会用于 {@link #deadlineNanos} 字段
 */
static long deadlineNanos(long delay) {
    long deadlineNanos = nanoTime() + delay;
    // Guard against overflow 防御性编程
    return deadlineNanos < 0 ? Long.MAX_VALUE : deadlineNanos;
}
```

2.4 构造方法

```
/**
 * 任务编号
 */
private final long id = nextTaskId.getAndIncrement();
/**
 * 任务执行时间，即到了该时间，该任务就会被执行
 */
private long deadlineNanos;
/**
 * 任务执行周期
 *
 * =0 - 只执行一次
 * >0 - 按照计划执行时间计算
```

文章目录


```
gtuu0123/article/details/6040159

d rate, <0 - repeat with fixed delay */

_QUEUE;

r executor,
long nanoTime) {
e, result), nanoTime);

r executor,
noTime, long period) {

ption("period: 0 (expected: != 0)");

}

ScheduledFutureTask(
        AbstractScheduledEventExecutor executor,
        Callable<V> callable, long nanoTime) {
    super(executor, callable);
    deadlineNanos = nanoTime;
    periodNanos = 0;
}
```

- 每个字段比较简单，胖友看上面的注释。

2.5 delayNanos

`#delayNanos(...)` 方法，获得距离指定时间，还要多久可执行。代码如下：

```
/**
 * @return 距离当前时间，还要多久可执行。若为负数，直接返回 0
 */
public long delayNanos() {
    return Math.max(0, deadlineNanos() - nanoTime());
}

/**
 * @param currentTimeNanos 指定时间
 * @return 距离指定时间，还要多久可执行。若为负数，直接返回 0
 */
```

文章目录


```
meNanos) {
) - (currentTimeNanos - START_TIME));


 TimeUnit.NANOSECONDS);
```

```
();


nternal()) {

all();

l(result);
            }
        } else {
            // 判断任务并未取消
            // check if is done as it may was cancelled
            if (!isCancelled()) {
                // 执行任务
                task.call();
                if (!executor().isShutdown()) {
                    // 计算下次执行时间
                    long p = periodNanos;
                    if (p > 0) {
                        deadlineNanos += p;
                    } else {
                        deadlineNanos = nanoTime() - p;
                    }
                    // 判断任务并未取消
                    if (!isCancelled()) {
                        // 重新添加到任务队列，等待下次定时执行
                        // scheduledTaskQueue can never be null as we lazy init it before submit t
                        Queue<ScheduledFutureTask<?>> scheduledTaskQueue =
                                ((AbstractScheduledEventExecutor) executor()).scheduledTaskQueue;
                        assert scheduledTaskQueue != null;
                        scheduledTaskQueue.add(this);
                    }
                }
            }
        }
    // 发生异常，通知任务执行失败
    } catch (Throwable cause) {
```

文章目录

;

。

在执行任务会略有不同。当然，大体是相同的。

时任务。

cellableInternal() 方法，设置任务不可取消。具体的方法实现，我

() 方法，执行任务。

essInternal(V result) 方法，回调通知注册在定时任务上的监听器。

reTask 继承了 PromiseTask 抽象类。

时任务。

ncelled() 方法，判断任务是否已经取消。这一点，和【第 7 行】的代

后续关于 Promise 的文章中分享。

l() 方法，执行任务。

。

。不同的执行 fixed 方式，计算方式不同。其中【第 25 行】的 -

负得正来计算。另外，这块会修改定时任务的 deadlineNanos 属性，

。

列 scheduledTaskQueue 中，等待下次定时执行。

Task#setFailureInternal(Throwable cause) 方法，回调通知注册在

2.7 cancel

有两个方法，可以取消定时任务。代码如下：

```
@Override
public boolean cancel(boolean mayInterruptIfRunning) {
    boolean canceled = super.cancel(mayInterruptIfRunning);
    // 取消成功，移除出定时任务队列
    if (canceled) {
        ((AbstractScheduledEventExecutor) executor()).removeScheduled(this);
    }
    return canceled;
}

// 移除任务
boolean cancelWithoutRemove(boolean mayInterruptIfRunning) {
    return super.cancel(mayInterruptIfRunning);
}
```

- 差别在于，是否 调用 AbstractScheduledEventExecutor#removeScheduled(ScheduledFutureTask) 方法，从定时任务队列移除自己。

2.8 compareTo

#compareTo(Delayed o) 方法，用于队列(ScheduledFutureTask 使用 PriorityQueue 作为**优先级队列**)排序。代码如下：

文章目录


```
heduledFutureTask<?>) o;
eadlineNanos();
```

序。

`#priorityQueueIndex(...)` 方法，获得或设置 `queueIndex` 属性。代码如下：

```
@Override
public int priorityQueueIndex(DefaultPriorityQueue<?> queue) { // 获得
    return queueIndex;
}

@Override
public void priorityQueueIndex(DefaultPriorityQueue<?> queue, int i) { // 设置
    queueIndex = i;
}
```

- 因为 ScheduledFutureTask 实现 PriorityQueueNode 接口，所以需要实现这两个方法。

3. AbstractScheduledEventExecutor

`io.netty.util.concurrent.AbstractScheduledEventExecutor` ，继承 AbstractEventExecutor 抽象类，**支持定时任务**的 EventExecutor 的抽象类。

3.1 构造方法

```
/**
 * 定时任务队列
 */
PriorityQueue<ScheduledFutureTask<?>> scheduledTaskQueue;
```

文章目录


```
tor() {

tor(EventExecutorGroup parent) {
```

列。

务队列。若未初始化，则进行创建。代码如下：

```
uledFutureTask<?>> SCHEDULED_FUTURE_TASK_COMPARATOR =
eTask<?>>() {

ledFutureTask<?> o1, ScheduledFutureTask<?> o2) {
2); //
```

```
PriorityQueue<ScheduledFutureTask<?>> scheduledTaskQueue() {
    if (scheduledTaskQueue == null) {
        scheduledTaskQueue = new DefaultPriorityQueue<ScheduledFutureTask<?>>(
                SCHEDULED_FUTURE_TASK_COMPARATOR,
                // Use same initial capacity as java.util.PriorityQueue
                11);
    }
    return scheduledTaskQueue;
}
```

- 创建的队列是 `io.netty.util.internal.DefaultPriorityQueue` 类型。具体的代码实现，本文先不解析。在这里，我们只要知道它是一个**优先级**队列，通过 `SCHEDULED_FUTURE_TASK_COMPARATOR` 来比较排序 ScheduledFutureTask 的任务优先级(顺序)。
- `SCHEDULED_FUTURE_TASK_COMPARATOR` 的具体实现，是调用 「2.8 compareTo」 方法来实现，所以队列**首个**任务，就是**第一个**需要执行的定时任务。

3.3 nanoTime

`#nanoTime()` **静态**方法，获得当前时间。代码如下：

```
protected static long nanoTime() {
    return ScheduledFutureTask.nanoTime();
}
```

- 在方法内部，会调用 「2.2 nanoTime」 方法。

3.4 schedule

文章目录

V> task) 方法，提交定时任务。代码如下：

```
 ScheduledFutureTask<V> task) {


);

到定时任务队列


add(task);
```

时任务到队列中。

调用 #schedule(final ScheduledFutureTask<V> task) 方法，分别创建

```
e(Callable<V> callable, long delay, TimeUnit unit) {
    ObjectUtil.checkNotNull(callable, "callable");
    ObjectUtil.checkNotNull(unit, "unit");
    if (delay < 0) {
        delay = 0;
    }
    // 无视，已经废弃
    validateScheduled0(delay, unit);

    return schedule(new ScheduledFutureTask<V>(
            this, callable, ScheduledFutureTask.deadlineNanos(unit.toNanos(delay))));
}

@Override
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUn
    ObjectUtil.checkNotNull(command, "command");
    ObjectUtil.checkNotNull(unit, "unit");
    if (initialDelay < 0) {
        throw new IllegalArgumentException(
                String.format("initialDelay: %d (expected: >= 0)", initialDelay));
    }
    if (period <= 0) {
        throw new IllegalArgumentException(
                String.format("period: %d (expected: > 0)", period));
    }
    // 无视，已经废弃
    validateScheduled0(initialDelay, unit);
    validateScheduled0(period, unit);
```

文章目录


```
reTask<Void>(
lable(command, null), // Runnable => Callable
ineNanos(unit.toNanos(initialDelay)), unit.toNanos(period)));



hFixedDelay(Runnable command, long initialDelay, long delay, Time
"command");
it");

ption(
lDelay: %d (expected: >= 0)", initialDelay));


ption(
 %d (expected: > 0)", delay));


unit);


reTask<Void>(
lable(command, null), // Runnable => Callable
ineNanos(unit.toNanos(initialDelay)), -unit.toNanos(delay)));
```

- 每个方法，前面都是校验参数的代码，重点是在最后对 `#schedule(final ScheduledFutureTask<V> task)` 方法的调用。

3.5 removeScheduled

`#removeScheduled(final ScheduledFutureTask<?> task)` 方法，移除出定时任务队列。代码如下：

```
final void removeScheduled(final ScheduledFutureTask<?> task) {
    if (inEventLoop()) {
        // 移除出定时任务队列
        scheduledTaskQueue().removeTyped(task);
    } else {
        // 通过 EventLoop 的线程，移除出定时任务队列
        execute(new Runnable() {
            @Override
            public void run() {
                removeScheduled(task);
            }
        });
    }
}
```

- 必须在 EventLoop 的线程中，**才能**移除出定时任务队列。

3.6 hasScheduledTasks

可执行的定时任务。代码如下：

```
d task is ready for processing.

asks() {
eduledTaskQueue = this.scheduledTaskQueue;
中，移除该任务
ask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek

cheduledTask.deadlineNanos() <= nanoTime();
```

个定时任务。不会从队列中，移除该任务。代码如下：

```
duledTask() {
eduledTaskQueue = this.scheduledTaskQueue;

    return scheduledTaskQueue.peek();
}
```

3.8 nextScheduledTaskNano

`#nextScheduledTaskNano()` 方法，获得定时任务队列，距离当前时间，还要多久可执行。

- 若队列**为空**，则返回 `-1` 。
- 若队列**非空**，若为负数，直接返回 0 。实际等价，ScheduledFutureTask#delayNanos() 方法。

代码如下：

```
/**
 * Return the nanoseconds when the next scheduled task is ready to be run or {@code -1} if no task is
 */
protected final long nextScheduledTaskNano() {
    Queue<ScheduledFutureTask<?>> scheduledTaskQueue = this.scheduledTaskQueue;
    // 获得队列首个定时任务。不会从队列中，移除该任务
    ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek
    if (scheduledTask == null) {
        return -1;
    }
    // 距离当前时间，还要多久可执行。若为负数，直接返回 0 。实际等价，ScheduledFutureTask#delayNanos() 方法。
    return Math.max(0, scheduledTask.deadlineNanos() - nanoTime());
}
```

文章目录

定时间内，定时任务队列**首个**可执行的任务，并且从队列中移除。代码如下：

```
dTask() {
()); // 当前时间


is ready to be executed with the given {@code nanoTime}.
 to retrieve the correct {@code nanoTime}.

dTask(long nanoTime) {


eduledTaskQueue = this.scheduledTaskQueue;
中，移除该任务
ask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek
    }

    // 在指定时间内，则返回该任务
    if (scheduledTask.deadlineNanos() <= nanoTime) {
        scheduledTaskQueue.remove(); // 移除任务
        return scheduledTask;
    }
    return null;
}
```

3.10 cancelScheduledTasks

`#cancelScheduledTasks()` 方法，取消定时任务队列的所有任务。代码如下：

```
/**
 * Cancel all scheduled tasks.
 * <p>
 * This method MUST be called only when {@link #inEventLoop()} is {@code true}.
 */
protected void cancelScheduledTasks() {
    assert inEventLoop();

    // 若队列为空，直接返回
    PriorityQueue<ScheduledFutureTask<?>> scheduledTaskQueue = this.scheduledTaskQueue;
    if (isNullOrEmpty(scheduledTaskQueue)) {
        return;
```

文章目录


```
heduledTasks = scheduledTaskQueue.toArray(new ScheduledFutureTask
: scheduledTasks) {
e);


ndexes();


Queue<ScheduledFutureTask<?>> queue) {
mpty();
```

xecutor

) 之 EventLoop 处理普通任务》 中，有个
将定时任务队列 `scheduledTaskQueue` 到达可执行的任务，添加到任务队

```
private boolean fetchFromScheduledTaskQueue() {
    // 获得当前时间
    long nanoTime = AbstractScheduledEventExecutor.nanoTime();
    // 获得指定时间内，定时任务队列**首个**可执行的任务，并且从队列中移除。
    Runnable scheduledTask  = pollScheduledTask(nanoTime);
    // 不断从定时任务队列中，获得
    while (scheduledTask != null) {
        // 将定时任务添加到 taskQueue 中。若添加失败，则结束循环，返回 false ，表示未获取完所有课执行的定时任
        if (!taskQueue.offer(scheduledTask)) {
            // 将定时任务添加回 scheduledTaskQueue 中
            // No space left in the task queue add it back to the scheduledTaskQueue so we pick it up
            scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);
            return false;
        }
        // 获得指定时间内，定时任务队列**首个**可执行的任务，并且从队列中移除。
        scheduledTask  = pollScheduledTask(nanoTime);
    }
    // 返回 true ，表示获取完所有可执行的定时任务
    return true;
}
```

- 代码比较简单，胖友看下笔者的详细代码注释。哈哈哈

666. 彩蛋

没有彩蛋，简单水文一篇。

文章目录