

我是一段不羁的公告！

记得给芬芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/oneMail>

<https://github.com/YunaiV/ruoyi-vue-pro>

• NETTY

## 精尽 Netty 源码解析 —— Util 之 HashedWheelTimer

笔者先把 Netty 主要的内容写完，所以关于 HashedWheelTimer 的分享，先放在后续的计划里。

老芬芳：其实是因为，自己想去研究下 Service Mesh，所以先简单收个小尾。

当然，良心如我，还是为对这块感兴趣的胖友，先准备好了一篇不错的文章：

- 德胜 《Netty工具类HashedWheelTimer源码走读(一)》
- 德胜 《Netty工具类HashedWheelTimer源码走读(二)》
- 德胜 《Netty工具类HashedWheelTimer源码走读(三)》
- Zacad 《netty源码解读之时间轮算法实现-HashedWheelTimer》

为避免可能 《netty源码解读之时间轮算法实现-HashedWheelTimer》 被作者删除，笔者这里先复制一份作为备份。

## 666. 备份

### 前因

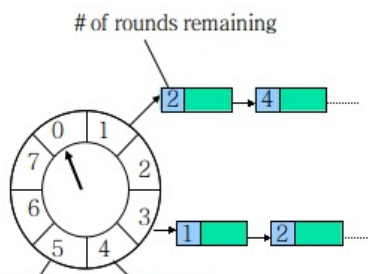
由于netty动辄管理100w+的连接，每一个连接都会有很多超时任务。比如发送超时、心跳检测间隔等，如果每一个定时任务都启动一个 Timer，不仅低效，而且会消耗大量的资源。

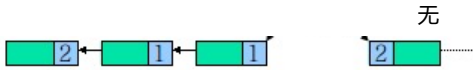
### 解决方案

根据George Varghese 和 Tony Lauck 1996 年的论文：[Hashed and Hierarchical Timing Wheels: data structures to efficiently implement a timer facility](#)。提出了一种定时轮的方式来管理和维护大量的 Timer 调度。

### 原理

时间轮其实就是一种环形的数据结构，可以想象成时钟，分成很多格子，一个格子代码一段时间（这个时间越短，Timer 的精度越高）。并用一个链表报错在该格子上的到期任务，同时一个指针随着时间一格一格转动，并执行相应格子中的到期任务。任务通过 取摸 决定放入那个格子。如下图所示：





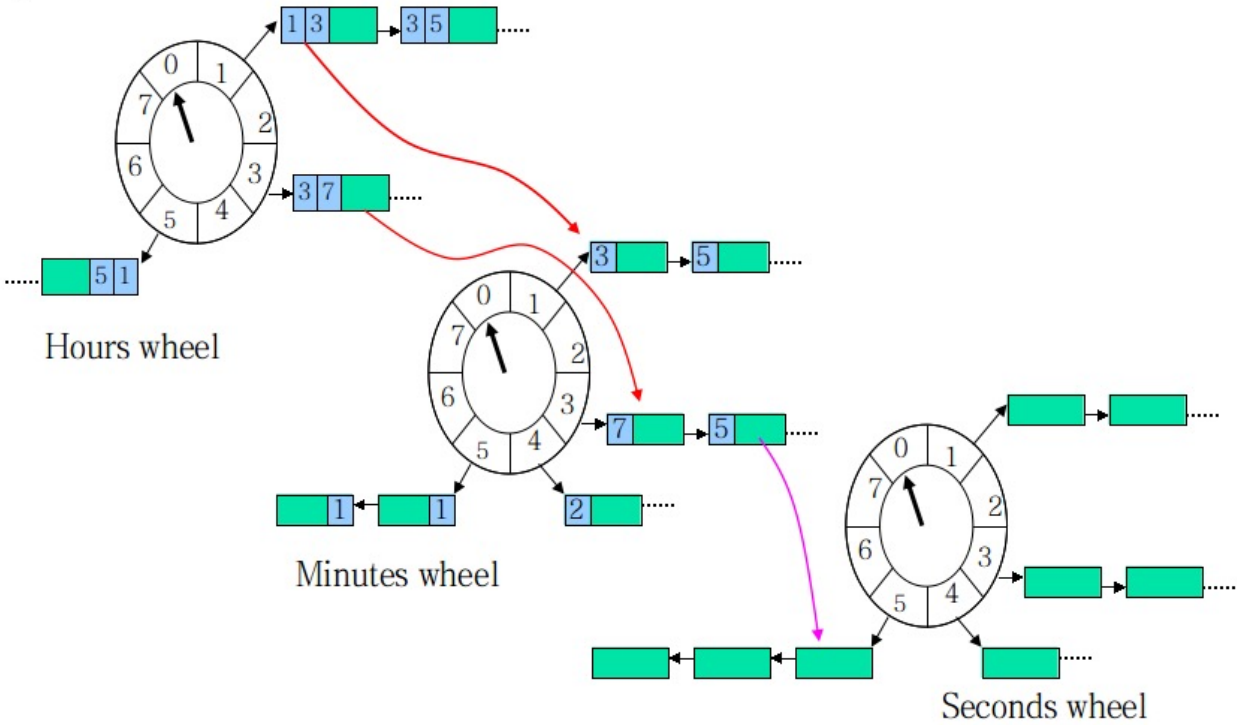
以上图为例，假设一个格子是1秒，则整个wheel能表示的时间段为8s，假如当前指针指向2，此时需要调度一个3s后执行的任务，显然应该加入到(2+3=5)的方格中，指针再走3次就可以执行了；如果任务要在10s后执行，应该等指针走完一个round零2格再执行，因此应放入4，同时将round (1) 保存到任务中。检查到期任务时应当只执行round为0的，格子上其他任务的round应减1。

是不是很像java中的 HashMap 。其实就是 HashMap 的哈希拉链算法，只不过多了指针转动与一些定时处理的逻辑。所以其相关的操作和 HashMap 也一致：

- 添加任务：O(1)
- 删除/取消任务：O(1)
- 过期/执行任务：最差情况为O(n)->也就是当 HashMap 里面的元素全部hash冲突，退化为一条链表的情况。平均O(1)->显然，格子越多，每个格子上的链表就越短，这里需要权衡时间与空间。

多层时间轮

如果任务的时间跨度很大，数量很大，单层的时间轮会造成任务的 round 很大，单个格子的链表很长。这时候可以将时间轮分层，类似于时钟的时分秒3层。如下图所示：



但是个人认为，多层的时间轮造成的算法复杂度的进一步提升。单层时间轮只需增加每一轮的格子就能解决链表过长的问题。因此，更倾向使用单层的时间轮，netty4中时间轮的实现也是单层的。

netty时间轮的实现-HashedWheelTimer

简单使用示例

1.引入netty依赖

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.4.Final</version>
</dependency>
```

2.示例代码

示例1:

```
@Test
public void test1() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    HashedWheelTimer hashedWheelTimer = new HashedWheelTimer(100, TimeUnit.MILLISECONDS);

    System.out.println("start:" + LocalDateTime.now().format(formatter));

    hashedWheelTimer.newTimeout(timeout -> {
        System.out.println("task :" + LocalDateTime.now().format(formatter));
    }, 3, TimeUnit.SECONDS);
    Thread.sleep(5000);
}
```

输出为:

start:2016-11-30 05:56:35

task :2016-11-30 05:56:38

示例2:

```
@Test
public void test2() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    HashedWheelTimer hashedWheelTimer = new HashedWheelTimer(100, TimeUnit.MILLISECONDS);

    System.out.println("start:" + LocalDateTime.now().format(formatter));

    hashedWheelTimer.newTimeout(timeout -> {
        Thread.sleep(3000);
        System.out.println("task1:" + LocalDateTime.now().format(formatter));
    }, 3, TimeUnit.SECONDS);


    hashedWheelTimer.newTimeout(timeout -> System.out.println("task2:" + LocalDateTime.now().format(
        formatter)), 4, TimeUnit.SECONDS);

    Thread.sleep(10000);
}
```

输出:

start:2016-12-01 08:32:37

task1:2016-12-01 08:32:43

task2:2016-12-01 08:32:43

可以看到，当前一个任务执行时间过长的时候，会影响后续任务的到期执行时间的。也就是说其中的任务是串行执行的。所以，要求里面的任务都要短平快。

HashedWheelTimer源码之构造函数

```

public HashedWheelTimer(
    ThreadFactory threadFactory, // 用来创建worker线程
    long tickDuration, // tick的时长, 也就是指针多久转一格
    TimeUnit unit, // tickDuration的时间单位
    int ticksPerWheel, // 一圈有几格
    boolean leakDetection // 是否开启内存泄露检测
) {

    // 一些参数校验
    if (threadFactory == null) {
        throw new NullPointerException("threadFactory");
    }
    if (unit == null) {
        throw new NullPointerException("unit");
    }
    if (tickDuration <= 0) {
        throw new IllegalArgumentException("tickDuration must be greater than 0: " + tickDuration);
    }
    if (ticksPerWheel <= 0) {
        throw new IllegalArgumentException("ticksPerWheel must be greater than 0: " + ticksPerWheel);
    }

    // 创建时间轮基本的数据结构, 一个数组。长度为不小于ticksPerWheel的最小2的n次方
    wheel = createWheel(ticksPerWheel);
    // 这是一个标示符, 用来快速计算任务应该呆的格子。
    // 我们知道, 给定一个deadline的定时任务, 其应该呆的格子=deadline%wheel.length.但是%操作是个相对耗时的操
    // 因为一圈的长度为2的n次方, mask = 2^n-1后低位将全部是1, 然后deadline&mask == deadline%wheel.length
    // java中的HashMap也是使用这种处理方法
    mask = wheel.length - 1;

    // 转换成纳秒处理
    this.tickDuration = unit.toNanos(tickDuration);

    // 校验是否存在溢出。即指针转动的时间间隔不能太长而导致tickDuration*wheel.length>Long.MAX_VALUE
    if (this.tickDuration >= Long.MAX_VALUE / wheel.length) {
        throw new IllegalArgumentException(String.format(
            "tickDuration: %d (expected: 0 < tickDuration in nanos < %d",
            tickDuration, Long.MAX_VALUE / wheel.length));
    }
    // 创建worker线程
    workerThread = threadFactory.newThread(worker);

    // 这里默认是启动内存泄露检测: 当HashedWheelTimer实例超过当前cpu可用核数*4的时候, 将发出警告
    leak = leakDetection || !workerThread.isDaemon() ? leakDetector.open(this) : null;
}

```

再来看下 createWheel 的代码:

```

private static HashedWheelBucket[] createWheel(int ticksPerWheel) {
    // 一些参数校验
    if (ticksPerWheel <= 0) {
        throw new IllegalArgumentException(
            "ticksPerWheel must be greater than 0: " + ticksPerWheel);
    }
}

```

```

    }
    if (ticksPerWheel > 1073741824) {
        throw new IllegalArgumentException(
            "ticksPerWheel may not be greater than 2^30: " + ticksPerWheel);
    }

    // 初始化ticksPerWheel的值为不小于ticksPerWheel的最小2的n次方
    ticksPerWheel = normalizeTicksPerWheel(ticksPerWheel);
    // 初始化wheel数组
    HashedWheelBucket[] wheel = new HashedWheelBucket[ticksPerWheel];
    for (int i = 0; i < wheel.length; i++) {
        wheel[i] = new HashedWheelBucket();
    }
    return wheel;
}

```

normalizeTicksPerWheel() 的代码:

```

// 初始化ticksPerWheel的值为不小于ticksPerWheel的最小2的n次方
private static int normalizeTicksPerWheel(int ticksPerWheel) {
    int normalizedTicksPerWheel = 1;
    while (normalizedTicksPerWheel < ticksPerWheel) {
        normalizedTicksPerWheel <<= 1;
    }
    return normalizedTicksPerWheel;
}

```

这里其实不建议使用这种方式, 因为当ticksPerWheel的值很大的时候, 这个方法会循环很多次, 方法执行时间不稳定, 效率也不够。推荐使用java8 HashMap的做法:

```

private int normalizeTicksPerWheel(int ticksPerWheel) {
    // 这里参考java8 hashmap的算法, 使推算的过程固定
    int n = ticksPerWheel - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    // 这里1073741824 = 2^30, 防止溢出
    return (n < 0) ? 1 : (n >= 1073741824) ? 1073741824 : n + 1;
}

```

## HashedWheelTimer源码之启动、停止与添加任务

start() 启动时间轮的方法:

```

// 启动时间轮。这个方法其实不需要显示的主动调用, 因为在添加定时任务(newTimeout()方法)的时候会自动调用此方法。
// 这个是合理的设计, 因为如果时间轮里根本没有定时任务, 启动时间轮也是空耗资源
public void start() {
    // 判断当前时间轮的状态, 如果是初始化, 则启动worker线程, 启动整个时间轮; 如果已经启动则略过; 如果是已经停止
    // 这里是一个Lock Free的设计。因为可能有多个线程调用启动方法, 这里使用AtomicIntegerFieldUpdater原子的更新
    switch (WORKER_STATE_UPDATER.get(this)) {

```

```

        case WORKER_STATE_INIT:
            if (WORKER_STATE_UPDATER.compareAndSet(this, WORKER_STATE_INIT, WORKER_STATE_STARTED)) {
                workerThread.start();
            }
            break;
        case WORKER_STATE_STARTED:
            break;
        case WORKER_STATE_SHUTDOWN:
            throw new IllegalStateException("cannot be started once stopped");
        default:
            throw new Error("Invalid WorkerState");
    }

    // 等待worker线程初始化时间轮的启动时间
    while (startTime == 0) {
        try {
            startTimeInitialized.await();
        } catch (InterruptedException ignore) {
            // Ignore - it will be ready very soon.
        }
    }
}

```

AtomicIntegerFieldUpdater是JUC里面的类，原理是利用反射进行原子操作。有比AtomicInteger更好的性能和更低得内存占用。跟踪这个类的github 提交记录，可以看到更详细的原因

stop() 停止时间轮的方法：

```

public Set<Timeout> stop() {
    // worker线程不能停止时间轮，也就是加入的定时任务，不能调用这个方法。
    // 不然会有恶意的定时任务调用这个方法而造成大量定时任务失效
    if (Thread.currentThread() == workerThread) {
        throw new IllegalStateException(
            HashedWheelTimer.class.getSimpleName() +
                ".stop() cannot be called from " +
                TimerTask.class.getSimpleName());
    }
    // 尝试CAS替换当前状态为“停止：2”。如果失败，则当前时间轮的状态只能是“初始化：0”或者“停止：2”。直接将当前状态
    if (!WORKER_STATE_UPDATER.compareAndSet(this, WORKER_STATE_STARTED, WORKER_STATE_SHUTDOWN)) {
        // workerState can be 0 or 2 at this moment - let it always be 2.
        WORKER_STATE_UPDATER.set(this, WORKER_STATE_SHUTDOWN);

        if (leak != null) {
            leak.close();
        }

        return Collections.emptySet();
    }

    // 终端worker线程
    boolean interrupted = false;
    while (workerThread.isAlive()) {
        workerThread.interrupt();
        try {

```

```

        workerThread.join(100);
    } catch (InterruptedException ignored) {
        interrupted = true;
    }
}

// 从中断中恢复
if (interrupted) {
    Thread.currentThread().interrupt();
}

if (leak != null) {
    leak.close();
}
// 返回未处理的任务
return worker.unprocessedTimeouts();
}

```

`newTimeout()` 添加定时任务：

```

public Timeout newTimeout(TimerTask task, long delay, TimeUnit unit) {
    // 参数校验
    if (task == null) {
        throw new NullPointerException("task");
    }
    if (unit == null) {
        throw new NullPointerException("unit");
    }
    // 如果时间轮没有启动，则启动
    start();

    // Add the timeout to the timeout queue which will be processed on the next tick.
    // During processing all the queued HashedWheelTimeouts will be added to the correct HashedWheelBu
    // 计算任务的deadline
    long deadline = System.nanoTime() + unit.toNanos(delay) - startTime;
    // 这里定时任务不是直接加到对应的格子中，而是先加入到一个队列里，然后等到下一个tick的时候，会从队列里取出最
    HashedWheelTimeout timeout = new HashedWheelTimeout(this, task, deadline);
    timeouts.add(timeout);
    return timeout;
}

```

这里使用的Queue不是普通java自带的Queue的实现，而是使用JCTool—一个高性能的的并发Queue实现包。

## HashedWheelTimer源码之HashedWheelTimeout

HashedWheelTimeout 是一个定时任务的内部包装类，双向链表结构。会保存定时任务到期执行的任务、deadline、round 等信息。

```

private static final class HashedWheelTimeout implements Timeout {

    // 定义定时任务的3个状态：初始化、取消、过期
    private static final int ST_INIT = 0;

```



```
private static final int ST_CANCELLED = 1;
private static final int ST_EXPIRED = 2;
// 用来CAS方式更新定时任务状态
private static final AtomicIntegerFieldUpdater<HashedWheelTimeout> STATE_UPDATER;

static {
    AtomicIntegerFieldUpdater<HashedWheelTimeout> updater =
        PlatformDependent.newAtomicIntegerFieldUpdater(HashedWheelTimeout.class, "state");
    if (updater == null) {
        updater = AtomicIntegerFieldUpdater.newUpdater(HashedWheelTimeout.class, "state");
    }
    STATE_UPDATER = updater;
}

// 时间轮引用
private final HashedWheelTimer timer;
// 具体到期需要执行的任务
private final TimerTask task;
private final long deadline;

@SuppressWarnings({"unused", "FieldMayBeFinal", "RedundantFieldInitialization" })
private volatile int state = ST_INIT;

// 离任务执行的轮数，当将次任务加入到格子中是计算该值，每过一轮，该值减一。
long remainingRounds;

// 双向链表结构，由于只有worker线程会访问，这里不需要synchronization / volatile
HashedWheelTimeout next;
HashedWheelTimeout prev;

// 定时任务所在的格子
HashedWheelBucket bucket;

HashedWheelTimeout(HashedWheelTimer timer, TimerTask task, long deadline) {
    this.timer = timer;
    this.task = task;
    this.deadline = deadline;
}

@Override
public Timer timer() {
    return timer;
}

@Override
public TimerTask task() {
    return task;
}

@Override
public boolean cancel() {
    // 这里只是修改状态为ST_CANCELLED，会在下次tick时，在格子中移除
    if (!compareAndSetState(ST_INIT, ST_CANCELLED)) {
        return false;
    }
}
```

```
    }  
    // 加入到时间轮的待取消队列，并在每次tick的时候，从相应格子中移除。  
    timer.cancelledTimeouts.add(this);  
    return true;  
}  
  
// 从格子中移除自身  
void remove() {  
    HashedWheelBucket bucket = this.bucket;  
    if (bucket != null) {  
        bucket.remove(this);  
    }  
}  
  
public boolean compareAndSetState(int expected, int state) {  
    return STATE_UPDATER.compareAndSet(this, expected, state);  
}  
  
public int state() {  
    return state;  
}  
  
@Override  
public boolean isCancelled() {  
    return state() == ST_CANCELLED;  
}  
  
@Override  
public boolean isExpired() {  
    return state() == ST_EXPIRED;  
}  
  
// 过期并执行任务  
public void expire() {  
    if (!compareAndSetState(ST_INIT, ST_EXPIRED)) {  
        return;  
    }  
  
    try {  
        task.run(this);  
    } catch (Throwable t) {  
        if (logger.isWarnEnabled()) {  
            logger.warn("An exception was thrown by " + TimerTask.class.getSimpleName() + '.', t);  
        }  
    }  
}  
  
// 略过toString()  
}
```

## HashedWheelTimer源码之HashedWheelBucket

HashedWheelBucket 用来存放HashedWheelTimeout, 结构类似于LinkedList。提供了 expireTimeouts(long deadline) 方法来过期并执行格子中的定时任务

```
private static final class HashedWheelBucket {
    // 指向格子中任务的首尾
    private HashedWheelTimeout head;
    private HashedWheelTimeout tail;

    // 基础的链表添加操作
    public void addTimeout(HashedWheelTimeout timeout) {
        assert timeout.bucket == null;
        timeout.bucket = this;
        if (head == null) {
            head = tail = timeout;
        } else {
            tail.next = timeout;
            timeout.prev = tail;
            tail = timeout;
        }
    }

    // 过期并执行格子中的到期任务, tick到该格子的时候, worker线程会调用这个方法, 根据deadline和remainingRound
    public void expireTimeouts(long deadline) {
        HashedWheelTimeout timeout = head;

        // 遍历格子中的所有定时任务
        while (timeout != null) {
            boolean remove = false;
            if (timeout.remainingRounds <= 0) { // 定时任务到期
                if (timeout.deadline <= deadline) {
                    timeout.expire();
                } else {
                    // 如果round数已经为0, deadline却>当前格子的deadline, 说放错格子了, 这种情况应该不会出现
                    throw new IllegalStateException(String.format(
                        "timeout.deadline (%d) > deadline (%d)", timeout.deadline, deadline));
                }
                remove = true;
            } else if (timeout.isCancelled()) {
                remove = true;
            } else { // 没有到期, 轮数-1
                timeout.remainingRounds --;
            }
            // 先保存next, 因为移除后next将被设置为null
            HashedWheelTimeout next = timeout.next;
            if (remove) {
                remove(timeout);
            }
            timeout = next;
        }
    }

    // 基础的链表移除node操作
    public void remove(HashedWheelTimeout timeout) {
        HashedWheelTimeout next = timeout.next;
```

```

// remove timeout that was either processed or cancelled by updating the linked-list
if (timeout.prev != null) {
    timeout.prev.next = next;
}
if (timeout.next != null) {
    timeout.next.prev = timeout.prev;
}

if (timeout == head) {
    // if timeout is also the tail we need to adjust the entry too
    if (timeout == tail) {
        tail = null;
        head = null;
    } else {
        head = next;
    }
} else if (timeout == tail) {
    // if the timeout is the tail modify the tail to be the prev node.
    tail = timeout.prev;
}

// null out prev, next and bucket to allow for GC.
timeout.prev = null;
timeout.next = null;
timeout.bucket = null;
}

/**
 * Clear this bucket and return all not expired / cancelled {@link Timeout}s.
 */
public void clearTimeouts(Set<Timeout> set) {
    for (;;) {
        HashedWheelTimeout timeout = pollTimeout();
        if (timeout == null) {
            return;
        }
        if (timeout.isExpired() || timeout.isCancelled()) {
            continue;
        }
        set.add(timeout);
    }
}

// 链表的poll操作
private HashedWheelTimeout pollTimeout() {
    HashedWheelTimeout head = this.head;
    if (head == null) {
        return null;
    }
    HashedWheelTimeout next = head.next;
    if (next == null) {
        tail = this.head = null;
    } else {
        this.head = next;
        next.prev = null;
    }
}

```

```

    }

    // null out prev and next to allow for GC.
    head.next = null;
    head.prev = null;
    head.bucket = null;
    return head;
}
}

```

## HashedWheelTimer源码之Worker

Worker 是时间轮的核心线程类。tick的转动，过期任务的处理都是在这个线程中处理的。

```

private final class Worker implements Runnable {
    private final Set<Timeout> unprocessedTimeouts = new HashSet<Timeout>();

    private long tick;

    @Override
    public void run() {
        // 初始化startTime.只有所有任务的deadline都是想对于这个时间点
        startTime = System.nanoTime();
        // 由于System.nanoTime()可能返回0，甚至负数。并且0是一个标示符，用来判断startTime是否被初始化，所以当
        if (startTime == 0) {
            startTime = 1;
        }

        // 唤醒阻塞在start()的线程
        startTimeInitialized.countDown();

        // 只要时间轮的状态为WORKER_STATE_STARTED，就循环的“转动”tick，循环判断响应格子中的到期任务
        do {
            // waitNextTick方法主要是计算下次tick的时间，然后sleep到下次tick
            // 返回值就是System.nanoTime() - startTime，也就是Timer启动后到这次tick，所过去的时间
            final long deadline = waitNextTick();
            if (deadline > 0) { // 可能溢出或者被中断的时候会返回负数，所以小于等于0不管
                // 获取tick对应的格子索引
                int idx = (int) (tick & mask);
                // 移除被取消的任务
                processCancelledTasks();
                HashedWheelBucket bucket =
                    wheel[idx];
                // 从任务队列中取出任务加入到对应的格子中
                transferTimeoutsToBuckets();
                // 过期执行格子中的任务
                bucket.expireTimeouts(deadline);
                tick++;
            }
        } while (WORKER_STATE_UPDATER.get(HashedWheelTimer.this) == WORKER_STATE_STARTED);

        // 这里应该是时间轮停止了，清除所有格子中的任务，并加入到未处理任务列表，以供stop()方法返回
        for (HashedWheelBucket bucket: wheel) {

```

```

        bucket.clearTimeouts(unprocessedTimeouts);
    }
    // 将还没有加入到格子中的待处理定时任务队列中的任务取出，如果是未取消的任务，则加入到未处理任务队列中，
    for (;;) {
        HashedWheelTimeout timeout = timeouts.poll();
        if (timeout == null) {
            break;
        }
        if (!timeout.isCancelled()) {
            unprocessedTimeouts.add(timeout);
        }
    }
    // 处理取消的任务
    processCancelledTasks();
}

// 将newTimeout()方法中加入到待处理定时任务队列中的任务加入到指定的格子中
private void transferTimeoutsToBuckets() {
    // 每次tick只处理10w个任务，以免阻塞worker线程
    for (int i = 0; i < 100000; i++) {
        HashedWheelTimeout timeout = timeouts.poll();
        // 如果没有任务了，直接跳出循环
        if (timeout == null) {
            break;
        }
        // 还没有放入到格子中就取消了，直接略过
        if (timeout.state() == HashedWheelTimeout.ST_CANCELLED) {
            continue;
        }

        // 计算任务需要经过多少个tick
        long calculated = timeout.deadline / tickDuration;
        // 计算任务的轮数
        timeout.remainingRounds = (calculated - tick) / wheel.length;

        //如果任务在timeouts队列里面放久了，以至于已经过了执行时间，这个时候就使用当前tick，也就是放到当前
        final long ticks = Math.max(calculated, tick); // Ensure we don't schedule for past.
        int stopIndex = (int) (ticks & mask);

        // 将任务加入到响应的格子中
        HashedWheelBucket bucket = wheel[stopIndex];
        bucket.addTimeout(timeout);
    }
}

// 将取消的任务取出，并从格子中移除
private void processCancelledTasks() {
    for (;;) {
        HashedWheelTimeout timeout = cancelledTimeouts.poll();
        if (timeout == null) {
            // all processed
            break;
        }
        try {

```

```

        timeout.remove();
    } catch (Throwable t) {
        if (logger.isWarnEnabled()) {
            logger.warn("An exception was thrown while process a cancellation task", t);
        }
    }
}

/**
 * calculate goal nanoTime from startTime and current tick number,
 * then wait until that goal has been reached.
 * @return Long.MIN_VALUE if received a shutdown request,
 * current time otherwise (with Long.MIN_VALUE changed by +1)
 */
//sleep, 直到下次tick到来, 然后返回该次tick和启动时间之间的时长
private long waitForNextTick() {
    //下次tick的时间点, 用于计算需要sleep的时间
    long deadline = tickDuration * (tick + 1);

    for (;;) {
        // 计算需要sleep的时间, 之所以加999999后再除1000000, 是为了保证足够的sleep时间
        // 例如: 当deadline - currentTime=2000002的时候, 如果不加999999, 则只睡了2ms,
        // 而2ms其实是未到达deadline这个时间点的, 所有为了使上述情况能sleep足够的时间, 加上999999后, 会多
        final long currentTime = System.nanoTime() - startTime;
        long sleepTimeMs = (deadline - currentTime + 999999) / 1000000;

        if (sleepTimeMs <= 0) {
            // 以下为个人理解: (如有错误, 欢迎大家指正)
            // 这里的意思应该是从时间轮启动到现在经过太长的时间(跨度大于292年...), 以至于让long装不下, 都
            if (currentTime == Long.MIN_VALUE) {
                return -Long.MAX_VALUE;
            } else {
                return currentTime;
            }
        }

        // Check if we run on windows, as if thats the case we will need
        // to round the sleepTime as workaround for a bug that only affect
        // the JVM if it runs on windows.
        //
        // See https://github.com/netty/netty/issues/356
        if (PlatformDependent.isWindows()) { // 这里是因为windows平台的定时调度最小单位为10ms, 如果不是
            sleepTimeMs = sleepTimeMs / 10 * 10;
        }

        try {
            Thread.sleep(sleepTimeMs);
        } catch (InterruptedException ignored) {
            // 调用HashedWheelTimer.stop()时优雅退出
            if (WORKER_STATE_UPDATER.get(HashedWheelTimer.this) == WORKER_STATE_SHUTDOWN) {
                return Long.MIN_VALUE;
            }
        }
    }
}

```

```
    }  
}  
  
public Set<Timeout> unprocessedTimeouts() {  
    return Collections.unmodifiableSet(unprocessedTimeouts);  
}  
}
```

## 总结

通过阅读源码，学到了很多之前不知道的知识点和注意事项。比如：

1. 操作数字型要考虑溢出问题
2. System.nanoTime() 返回值
3. Atomic\*FieldUpdater类的运用
4. 一些代码设计方式
5. 不断优化性能，Lock Less代替Lock；Lock Free代替Lock Less
6. JCTool高性能队列的使用