

定时轮算法及其实现

HashedWheelTimer定时轮算法被广泛使用，netty、dubbo甚至是操作系统Linux中都有其身影，用于管理及维护大量Timer调度算法。

一个HashedWheelTimer是环形结构，类似一个时钟，分为很多槽，一个槽代表一个时间间隔，每个槽使用双向链表存储定时任务，指针周期性的跳动，跳动到一个槽位，就执行该槽位的定时任务。

The diagram illustrates the HashedWheelTimer structure. It features a central circular wheel divided into eight slots, numbered 0 through 7. An arrow points to slot 0, which is labeled '# of rounds remaining'. Each slot contains a linked list of tasks. Slot 0 has a task with 2 rounds remaining, which points to a task with 4 rounds remaining. Slot 3 has a task with 1 round remaining, which points to a task with 2 rounds remaining. Slot 5 has a task with 2 rounds remaining, which points to a task with 1 round remaining, which in turn points to a task with 1 round remaining. The text '云教程中心' is visible in the bottom right corner of the diagram area.

图 1: 时间轮算法

实现

定时轮的具体实现中，按照职责不同，可分为 时钟引擎、时钟槽、定时任务 3个主要角色，为透彻理解其实现，行文有穿插， 这一部分抹掉了具体实现语言的特性。

定时任务——HashedWheelTimeout

在具体实现中定时任务HashedWheelTimeout扮演着双重角色，既是双向链表的节点，同时也是实际调度任务TimerTask的容器， 其由引擎在滴答运行起始时刻使用&取hash装入对应的时钟槽。

关键属性

HashedWheelTimeout next,prev : 当前定时任务在链表中的前驱和后继引用

TimerTask task : 实际被调度的任务

long deadline : 该时间是相对于引擎的startTime的，由公式 `currentTime + delay - startTime` 得到，时间单位一般为纳秒

delay: 任务在提交时给出的相对当前的滞后执行时间

currentTime: 当前内核时间

int state: 定时任务当前所处状态, INIT⁰—初始态, CANCELLED¹—已被取消, EXPIRED²—已过期

NOTE

状态的标记是在发起 `expire` 或 `cancel` 操作的起始瞬间完成的

HashedWheelTimeout 本身支持的操作并不多, 如下:

1. `remove`: 调用所属时钟槽的 `remove` 将自身从中移除, 若尚未被装入槽中, 需要额外对所属定时轮的 `pendingTimeouts` 执行 `-1` 处理
2. `expire`: 任务到期, 驱动 `TimerTask` 运行
3. `cancel`: 任务被提交方取消, 取消的任务被装入定时轮的 `cancelledTimeouts` 队列中, 待引擎进入下一个滴答时刻调用其 `remove` 移除自身

时钟槽——HashedWheelBucket

时钟槽实际上就是一个用于缓存和管理定时任务的双向链表容器^{每一个节点也即一个定时任务}。它持有链表的首尾两个节点, 由于每个节点均持有前驱和后继的引用, 因此利用链表的这个特性可以完成如下操作:

1. `addTimeout`: 新增尾节点, 添加任务
2. `pollTimeout`: 移除首节点, 取出头部任务
3. `remove`: 根据引用移除指定节点, 被移除任务已被处理或被取消, 对所属定时轮的 `pendingTimeouts` 相应 `-1` 处理
4. `clearTimeouts`: 循环调用 `pollTimeout` 获取到所有未超时或者未被取消的任务
5. `expireTimeouts`: 从首节点开始循环遍历槽中所有节点, 调用 `remove` 取出到期任务运行 `expire` 或直接移除 `remove` 被取消的任务, 对其它正常任务的剩余轮数执行 `-1` 操作

时钟引擎——HashedWheelTimer

时钟引擎有节律地周期性运作, 总是根据当前时钟滴答选定对应的时钟槽, 从链表头部开始迭代, 对每一个任务计算出其是否属于当前时钟周期, 属于则取出运行, 否则便将对剩下时钟周期数执行减一操作。

另外，引擎维持着两个缓存定时任务的阻塞队列，其中一个用于接受外界断断续续地投递进来的，另外一个则用于缓存那些主动取消的，引擎需要在滴答开始期间 先行将他们装入对应的时钟槽或从中移除他们。

关键属性

`Queue<HashedWheelTimeout> timeouts、cancelledTimeouts`：队列，用于缓存外界主动提交或取消的任务

`int workerState`：定时轮当前所处状态：

NOTE | 状态值

`init0`——初始态

`started1`——已开始运行

`shutdown2`——已结束运行

`startTime`：当前定时轮正式开始调度任务的时间，此后所有提交的定时任务 第一个任务提交的开始，引擎就开始正式执行了，均以该时间点作为起点

`ticks`：滴答，由时钟引擎维护，是步长为1的单调递增计数，也即 `ticks+=1`

`ticksDuration`：滴答时长，每轮询一个特定时钟槽代表走完一个滴答时长

`pendingTimeouts` 当前定时轮实时任务剩余数

`n`：时钟轮槽数为`n`，不一定和期望达到的槽数一致，取大于且最靠近的2的幂次方值，其计算公式为 $n=2^x$

`mask`：掩码，`mask = n - 1`，执行 `ticks & mask` 便能定位到对应位置的时钟槽，效果上相当于 `ticks % (mask + 1)`，由`n`这个2的幂次方保证

引擎内核——Worker

时钟引擎实际上分为对外接口和调度运行两部分，可以想象内核就是一个引擎的心脏起搏器驱动着定时轮的运行，完成任务的调度，实现上对应一个工作线程，为方便 理解，先单独阐述他们所依赖的内核状态。

内核状态

对于任何引擎来说，状态机是其关键组成，因此状态值的控制对其而言是至关重要，因而将这部分作为单独的部分阐述。内核状态有定时轮维护管理，对外提供的接口都要借助它实现。初始时便为init状态，当引擎被设计成不可复活时，便不存在 `init/started/shutdown` → `init` 这样的迁移过程。

NOTE

`start()`

init → started

于引擎的整个生命周期而言，这个状态的迁移过程只允许发生一次，实现中会结合`startTime`做防御性保护，直到整个过程完成为止

started → started

表示引擎已经被启用，一般直接忽略，否则也会等效于什么也不做

shutdown → started

定时轮一般被设计为不能复活的，这种情况下该过程是不允许发生的，属于外界调用方的越界行为

NOTE

`stop()`

init → shutdown

尚未开始就进入终结状态，一般发生在对定时轮已经完成初始化，但尚未给其提交任务或调用过 `start()` 操作。但还有另外一种特殊的情形，在多个线程对同一定时轮进行操作时发生争用，一个线程在另外一个线程刚开始进入 `start()` 操作时，调用了`stop()`

started → shutdown

正常的引擎关闭操作，一旦进入该过程，会持续到引擎完全终止，对应到实现上就是结束Work线程

shutdown → shutdown

该迁移过程没有实际语义，一般直接跳过

外部接口

这部分内容实际上已经在内核状态一节已经有过具体阐述

start：用于定时轮开启引擎，但外界不一定需要调用此方法启用定时轮，因为外界每次调用 `newTimeout()` 提交任务时，定时轮都会主动调用该接口，以确保引擎已经处于运行状态。

stop：完成定时轮引擎的关闭过程，返回未被处理的定时任务

`Timeout newTimeout(TimerTask task, long delay, TimeUnit unit)`：用于向引擎提交任务，在任务被正式加入`timeouts`队列之前：1) 定时轮 会首先调用 `start()` 确保引擎已经启动；然后为加入的`Timeout`计算出`deadline`值。

调度运行

有了以上的分析，对定时轮的任务调度也就不难理解了，简单而言就是周期性的执行滴答操作，对应如下几个操作：

1. 等待进入滴答周期
2. 时钟转动，滴答周期开始：
 - a. 将外界主动取消的装载在`cancelledTimeouts`队列的任务逐个移除
 - b. 将外界提交的装载在`timeouts`队列的任务逐个载入对应的时钟槽里
3. 根据当前`tick`定位对应时钟槽，执行其中的定时任务
4. 检测引擎内核状态是否已经被终止，若未被终止，则循环执行上述操作，否则往下继续执行
5. 将下述方式获取到未被处理的任务加入`unprocessedTimeouts`队列：
 - a. 遍历时钟槽调用 `clearTimeouts()`
 - b. 对`timeouts`队列中未被加入槽中循环调用 `poll()`
6. 移除最后一个滴答周期后加入到`cancelledTimeouts`队列任务

IMPORTANT

相邻两个滴答周期的开始时间理论上来说是等距的，但是结束时间则会随该周期所需处理任务的数目及时长有所变化。因而引擎剩下的休眠时间需要使用如下公式获得：

$$\text{tickDuration} * (\text{tick} + 1) - (\text{currentTime} - \text{startTime})$$

定时轮在dubbo中的应用

实际上，定时轮算法并不直接用于等周期性的执行某些提交任务，向其提交的任务只会到期执行一次，但具体应用中，会利用每次任务的执行，调用 `newTimeout()` 提交`Timer`所引用的当前任务，使其在若干单位时间后重新继续执行。这样做的好处是，如果诸如IO等耗时任务，甚至是某些原因导致的当下执行任务卡住比较 长时间，后面不会有同样的任务不断提交进来，而导致任务堆积至无法处理。可见这里说的额周期性任务不是严格固定 每x单位时间 执行一次的任务。

Dubbo中对定时轮的应用主要体现在如下几个方面：

1. 失败重试

- a. 注册 **Register**
- b. 取消注册 **Unregister**
- c. 订阅 **Subscribe**
- d. 取消订阅 **Unsubscribe**

2. 周期任务

- a. 心跳 **Heartbeat**
- b. 重连 **Reconnect**
- c. 下线 **CloseChannel**

IMPORTANT

定时轮用单一的线程去管理触发Task的运行，Task执行期间，不能直接抛异常，否则会导致整个定时轮引擎的奔溃而使得提交的后续任务无法执行。Task的模式如下：

JAVA

```
try {  
    if (sthCheck()) {  
        logger.warn("Sth happended");  
        doBuz();  
    }  
} catch (Throwable t) {  
    logger.warn("Exception when do sth ", t);  
}
```

周期任务

在dubbo中每一个连接被表征为一个Channel通道，dubbo节点间建立连接相互通信，单个节点需要维护和多个连入节点的连接：①通过持续发送心跳检测以保持连接 ②对超过了一定时间段处于空闲状态的连接进行下线处理；③对已经掉线非下线处理的Channel进行重连处理。

基本的步骤如下：

1. 滴答运行时Task通过回调获得当前节点的所有连入Channel
2. 对没有被关闭的节点执行实际的任务操作，比如心跳
3. 通过volatile的可见性保证属性检测当前任务是否被取消，是返回，否继续
4. 若定时轮是否还在运行，则使用其提供的 newTimeout() 提交一个新的Task

以下结合源码进行分析：

JAVA

```
public abstract class AbstractTimerTask implements TimerTask {
    /**
     * 该属性比较关键，真正执行Task操作的是定时轮所持有的线程，而唤起``cancel()``操作的是提交任务的其它线程
     */
    protected volatile boolean cancel = false;

    public void cancel() {
        this.cancel = true;
    }

    ....//省略部分代码

    private void reput(Timeout timeout, Long tick) {
        if (timeout == null || tick == null) {
            throw new IllegalArgumentException();
        }

        if (cancel) {
            return;
        }

        Timer timer = timeout.timer();
        if (timer.isStop() || timeout.isCancelled()) {
            return;
        }

        timer.newTimeout(timeout.task(), tick, TimeUnit.MILLISECONDS);
    }

    @Override
    public void run(Timeout timeout) throws Exception {
        Collection<Channel> c = channelProvider.getChannels();
        for (Channel channel : c) {
            if (channel.isClosed()) {
                continue;
            }
            doTask(channel);
        }
        reput(timeout, tick);
    }

    protected abstract void doTask(Channel channel);

    interface ChannelProvider {
        Collection<Channel> getChannels();
    }
}
```


从以上代码可以看出，利用定时轮实现间隔时间任务的模式比较固定，如下：

JAVA

```
//保证立马被定时轮获知任务已被取消
protected volatile boolean cancel = false;

public void run(Timeout timeout) throws Exception {
    //执行任务业务逻辑
    doTask()
    //重新
    reput(timeout, tick);
}

private void reput(Timeout timeout, Long tick) {

    if (cancel) {
        return;
    }

    //确认定时轮还处于运行状态
    Timer timer = timeout.timer();
    if (timer.isStop() || timeout.isCancelled()) {
        return;
    }

    //向定时轮重新提交一个新的__Task__，指定tick单位时间后执行
    timer.newTimeout(timeout.task(), tick,
        TimeUnit.MILLISECONDS);
}
```

NOTE

周期性任务中对每一个Channel所做得事情比较简单，实际上是在满足条件的情况调用channel的指定操作

1. 心跳—— channel.send(req)

JAVA

```
Request req = new Request();
req.setVersion(Version.getProtocolVersion());
//双边通讯
req.setTwoWay(true);
//事件类型：心跳
req.setEvent(Request.HEARTBEAT_EVENT);
```


1. 重连—— `((Client) channel).reconnect()`

2. 下线—— `channel.close()`

失败重试

网络情况的的复杂多变性，使得一件原本在单机上很轻易的事情，分布式应用中，为确保某类型的操作能发生可能需要重试多次。除了catch到异常后进行重试和对重试次数有规定外，和上述的周期任务实现几乎一样。模式如下：

```

/**
 * times of retry.
 * retry task is execute in single thread so that the times is not need volatile.
 */
//重试次数并没有被申明为volatile, 原因是该变量只会被定时轮引擎中的工作线程所使用到, 投递任务的那个线程
//并没有直接接触
private int times = 1;
....
protected void reput(Timeout timeout, long tick) {
    if (timeout == null) {
        throw new IllegalArgumentException();
    }

    Timer timer = timeout.timer();
    if (timer.isStop() || timeout.isCancelled() || isCancel()) {
        return;
    }
    times++;
    timer.newTimeout(timeout.task(), tick, TimeUnit.MILLISECONDS);
}

@Override
public void run(Timeout timeout) throws Exception {
    if (timeout.isCancelled() || timeout.timer().isStop() || isCancel()) {
        // other thread cancel this timeout or stop the timer.
        return;
    }
    if (times > retryTimes) {
        // reach the most times of retry.
        logger.warn("Final failed to execute task " + taskName + ", url: "
            + url + ", retry " + retryTimes + " times.");

        return;
    }
    if (logger.isInfoEnabled()) {
        logger.info(taskName + " : " + url);
    }
    try {
        doRetry(url, registry, timeout);
    } catch (Throwable t) {

        // Ignore all the exceptions and wait for the next retry
        logger.warn("Failed to execute task " + taskName + ", url: "
            + url + ", waiting for again, cause:" + t.getMessage(), t);

        // reput this task when catch exception.
        reput(timeout, retryPeriod);
    }
}

protected abstract void doRetry(URL url, FailbackRegistry registry, Timeout timeout);

```