抖音面试：说说延迟任务的调度算法?

Netty 框架是以性能著称的框架，因此在它的框架中使用了大量提升性能的机制，例如 Netty 用于实现延迟队列的时间轮调度算法就是一个典型的例子。使用时间轮调度算法可以实现海量任务新增和取消任务的时间度为 O(1)，那么什么是时间轮调度算法呢？接下来我们一起来看。

1.延迟任务实现

在 Netty 中，我们需要使用 HashedWheelTimer 类来实现延迟任务，例如以下代码：

```
public class DelayTaskExample {
    public static void main(String[] args) {
        System.out.println("程序启动时间：" + LocalDateTime.now());
        NettyTask();
    }

    private static void NettyTask() {
        // 创建延迟任务实例
        HashedWheelTimer timer = new HashedWheelTimer(3, // 间隔时间
                TimeUnit.SECONDS, // 间隔时间单位
                100); // 时间轮中的槽数
        // 创建任务
        TimerTask task = new TimerTask() {
            @Override
            public void run(Timeout timeout) throws Exception {
                System.out.println("执行任务时间：" + LocalDateTime.now());
            }
        };
        // 将任务添加到延迟队列中
        timer.newTimeout(task, 0, TimeUnit.SECONDS);
    }
}
```

以上程序的执行结果如下：

> 程序启动时间：2024-06-04T10:16:23.033
>
>
> 执行任务时间：2024-06-04T10:16:26.118

从上述执行结果可以看出，我们使用 HashedWheelTimer 实现了延迟任务的执行。

2.时间轮调度算法

那么问题来了，HashedWheelTimer 是如何实现延迟任务的？什么是时间轮调度算法？

查看 HashedWheelTimer 类的源码会发现，其实它是底层是通过时间轮调度算法来实现的，以下是 HashedWheelTimer 核心实现源码（HashedWheelTimer 的创建源码）如下：

```
private static HashedWheelBucket[] createWheel(int ticksPerWheel) {
    // 省略其他代码
    ticksPerWheel = normalizeTicksPerWheel(ticksPerWheel);
    HashedWheelBucket[] wheel = new
HashedWheelBucket[ticksPerWheel];
    for (int i = 0; i < wheel.length; i ++) {
        wheel[i] = new HashedWheelBucket();
    }
    return wheel;
}
private static int normalizeTicksPerWheel(int ticksPerWheel) {
    int normalizedTicksPerWheel = 1;
    while (normalizedTicksPerWheel < ticksPerWheel) {
        normalizedTicksPerWheel <<= 1;
    }
    return normalizedTicksPerWheel;
}
private static final class HashedWheelBucket {
    private HashedWheelTimeout head;
    private HashedWheelTimeout tail;
```

```
    // 省略其他代码
}

```

在 HashedWheelTimer 中，使用了 HashedWheelBucket 数组实现时间轮的概念，每个 HashedWheelBucket 表示时间轮中一个 slot（时间槽），HashedWheelBucket 内部是一个双向链表结构，双向链表的每个节点持有一个 HashedWheelTimeout 对象，HashedWheelTimeout 代表一个定时任务，每个 HashedWheelBucket 都包含双向链表 head 和 tail 两个 HashedWheelTimeout 节点，这样就可以实现不同方向进行链表遍历，如下图所示：

时间轮算法的设计思想就来源于钟表，如上图所示，时间轮可以理解为一种环形结构，像钟表一样被分为多个 slot 槽位。每个 slot 代表一个时间段，每个 slot 中可以存放多个任务，使用的是链表结构保存该时间段到期的所有任务。时间轮通过一个时针随着时间一个个 slot 转动，并执行 slot 中的所有到期任务。

任务的添加是根据任务的到期时间进行取模，然后将任务分布到不同的 slot 中。如上图所示，时间轮被划分为 8 个 slot，每个 slot 代表 1s，当前时针指向 2 时，假如现在需要调度一个 3s 后执行的任务，应该加入 2+3=5 的 slot 中；如果需要调度一个 12s 以后的任务，需要等待时针完整走完一圈 round 零 4 个 slot，需要放入第 (2+12)%8=6 个 slot。

那么当时针走到第 6 个 slot 时，怎么区分每个任务是否需要立即执行，还是需要等待下一圈 round，甚至更久时间之后执行呢？所以我们需要把 round 信息保存在任务中。例如图中第 6 个 slot 的链表中包含 3 个任务，第一个任务 round=0，需要立即执行；第二个任务 round=1，需要等待 1*8=8s 后执行；第三个任务 round=2，需要等待 2*8=8s 后执行。所以当时针转动到对应 slot 时，只执行 round=0 的任务，slot 中其余任务的 round 应当减 1，等待下一个 round 之后执行。

可以看出时间轮有点类似 HashMap，如果多个任务如果对应同一个 slot，处理冲突的方法采用的是拉链法。在任务数量比较多的场景下，适当增加时间轮的 slot 数量，可以减少时针转动时遍历的任务个数。

时间轮定时器最大的优势就是，任务的新增和取消都是 O(1) 时间复杂度，而且只需要一个线程就可以驱动时间轮进行工作。

课后思考

Netty 中的时间轮调度算法有什么缺点?

参考 & 鸣谢

《Netty核心原理剖析与RPC实践》

> 本文已收录到我的面试小站 [www.javacn.site](http://cxyroad.com/”https://www.javacn.site”)，其中包含的内容有：Redis、JVM、并发、并发、MySQL、Spring、Spring MVC、Spring Boot、Spring Cloud、MyBatis、设计模式、消息队列等模块。

原文链接: https://juejin.cn/post/7376582861661126697