# 定时器

## Timer

### 概念

定时计划任务功能在Java中主要使用的就是Timer对象，它在内部使用多线程的方式进行处理，所以它和多线程技术还是有非常大的关联的。在JDK中Timer类主要负责计划任务的功能，也就是在指定的时间开始执行某一个任务，但封装任务的类却是TimerTask类。

通过继承 TimerTask 类 并实现 run() 方法来自定义要执行的任务：

```
public class Mytask extends TimerTask {
    @Override
    public void run()
    {
        System.out.println("Hello World!");
    }
}
```

通过执行Timer.schedule() 在执行时间运行任务：

```
public static void main(String[] args) throws ParseException {
    Timer timer = new Timer();
    //延迟2秒后执行一次
    timer.schedule(new Mytask(), 2000);
}
```

### 注意事项

- 创建一个 Timer 对象就是新启动了一个线程，但是这个新启动的线程，并不是守护线程，它一直在后台运行，通过如下 可以将新启动的 Timer 线程设置为守护线程。

  ```
  private static Timer timer=new Timer(true);
  ```

- 提前：当计划时间早于当前时间，则任务立即被运行。
- 延迟：TimerTask 是以队列的方式一个一个被顺序运行的，所以执行的时间和你预期的时间可能不一致，因为前面的任务可能消耗的时间较长，则后面的任务运行的时间会被延迟。延迟的任务具体开始的时间，就是依据前面任务的"结束时间"
- 周期性运行：Timer.schedule(TimerTask task,long delay, long period) 从 当前延迟时间开始每隔 period 毫秒执行一次任务

### Timer的cancel() 和 TimerTask的cancel() 的区别？

前面提到任务的执行是以对列的方式一个个被顺序执行的，TimerTask.cancel() 指的是 把当前任务从任务对列里取消。Timer.cancel() 值的是把当前任务队列里的所有任务都取消。值得注意的是，Timer 的cancel()有时并不一定会停止执行计划任务，而是正常执行。这是因为Timer类中的cancel()方法有时并没有争抢到queue锁，所以TimerTask类中的任务继续正常执行。

## scheduleAtFixedRate 和 schedule 区别

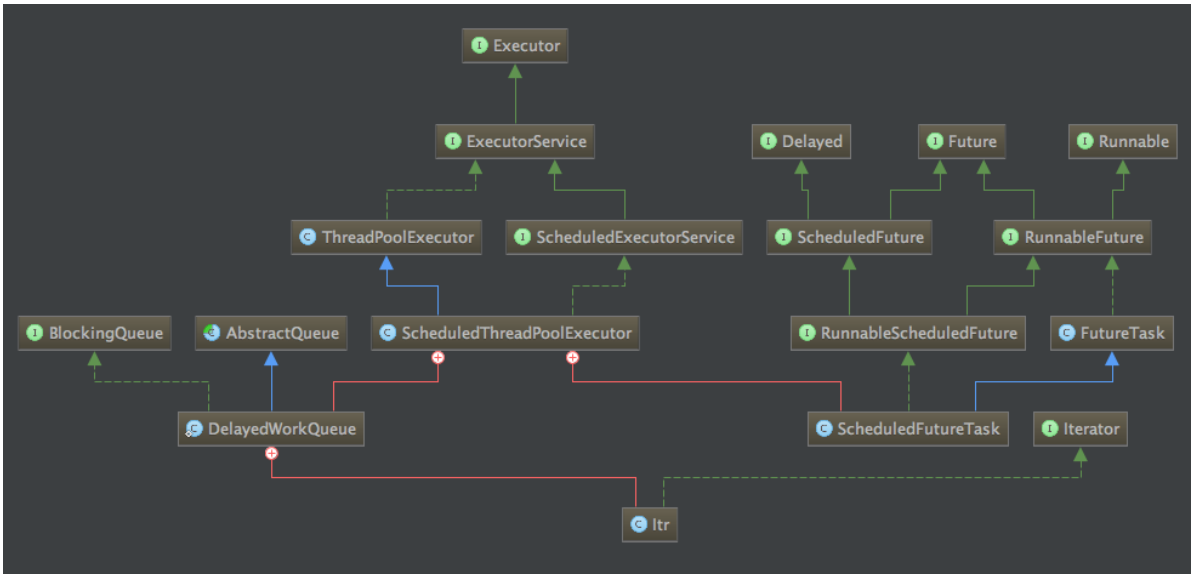
### 相同点

- 方法schedule 和方法 scheduleAtFixedRate 都会按顺序执行，所以不用考虑非线程安全的情况。
- 方法schedule 和方法 scheduleAtFixedRate 如果执行任务的时间没有被延迟，那么下一次任务的执行时间参考的是上一次的任务的"开始"时的时间来计算的。
- 方法schedule 和方法 scheduleAtFixedRate 如果执行任务的时间被延迟了，那么下一次任务的执行时间参考的是上一次任务"结束"时的时间来计算。

### 区别

方法schedule 和方法 scheduleAtFixedRate 在使用上基本没什么差别，就是 scheduleAtFixedRate 具有追赶执行性，什么意思呢？就是如果任务 在周期性运行过程中被打断了，scheduleAtFixedRate 会尝试把之前落下的任务补上运行。而schedule就不管了，接着运行接下来的任务就行了。

## ScheduledExecutorService是什么？




## schedule()方法

```
public ScheduledFuture<?> schedule(Runnable command,
                                   long delay,
                                   TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    RunnableScheduledFuture<?> t = decorateTask(command,
        new ScheduledFutureTask<Void>(command, null,
                                      triggerTime(delay, unit)));
    delayedExecute(t);
    return t;
}
public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                                       long delay,
                                       TimeUnit unit) {
    if (callable == null || unit == null)
        throw new NullPointerException();
```

```
    RunnableScheduledFuture<V> t = decorateTask(callable,
        new ScheduledFutureTask<V>(callable,
                                   triggerTime(delay, unit)));
    delayedExecute(t);
    return t;
}
```

- 起到延迟执行的作用；
- 多次提交任务时，后面任务延迟执行的时间是否准确，与**线程池的大小**和**上一个任务执行耗时**两个因素有关。
- 提交任务的先后顺序与实际执行的顺序无关，而是与延迟时间有关。

## scheduleAtFixedRate

该方法设置了执行周期，下一次执行时间相当于是上一次的执行时间加上period，它是采用已固定的频率来执行任务：

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    if (period <= 0)
        throw new IllegalArgumentException();
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
                                      null,
                                      triggerTime(initialDelay, unit),
                                      unit.toNanos(period));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    sft.outerTask = t;
    delayedExecute(t);
    return t;
}
```

- 此方法用于周期性执行任务
- 当任务耗时长于周期，那么下一个周期任务将在上一个执行完毕之后马上执行。
- 当任务耗时短于周期，那么正常周期性执行。

## scheduleWithFixedDelay

该方法设置了执行周期，与scheduleAtFixedRate方法不同的是，下一次执行时间是**上一次任务执行完**的系统时间加上period，因而具体执行时间不是固定的，但周期是固定的，是采用相对固定的延迟来执行任务：

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                 long initialDelay,
                                                 long delay,
                                                 TimeUnit unit) {
    if (command == null || unit == null)
        throw new NullPointerException();
    if (delay <= 0)
        throw new IllegalArgumentException();
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command,
```

```
                                       null,
                                       triggerTime(initialDelay, unit),
                                       unit.toNanos(-delay));
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    sft.outerTask = t;
    delayedExecute(t);
    return t;
}
```

- 此方法用于周期性执行
- 无论上一个方法耗时多长，下一个方法都会等到上一个方法执行完毕之后，再经过delay的时间才执行。

## schedule方法的实现原理