实验5 定时器

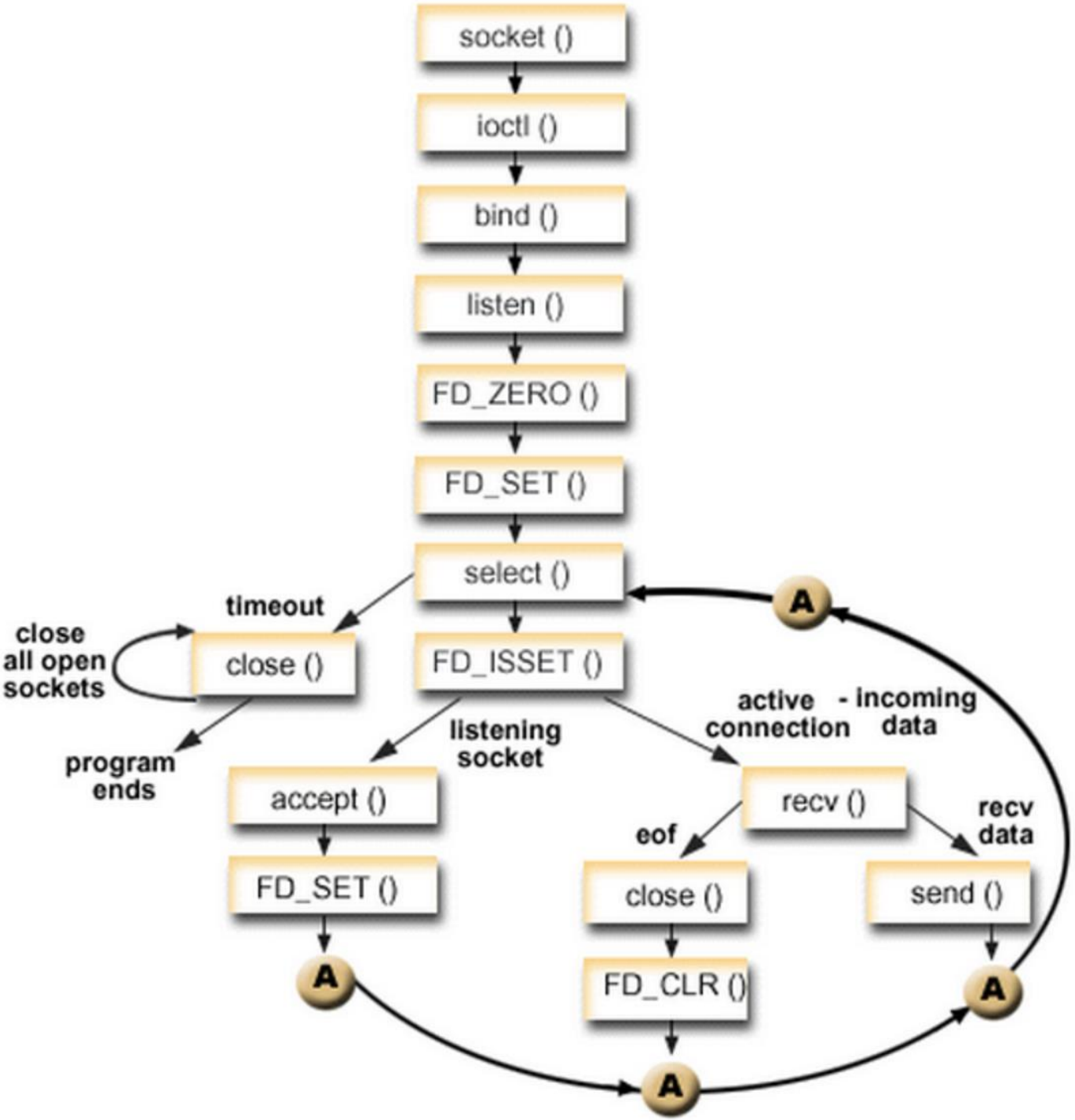
互联网络程序设计实验

实验目标

- 网络协议用定时器
- 定时引擎
  - Timer IO
  - Select/Epoll
  - timerfd
- 设计方案
  - 链表
  - 二叉树
  - 时间轮

LINUX中的IO复用

- select
  - select　　函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪（有数据可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。


socket ()
ioctl ()
bind ()
listen ()
FD_ZERO ()
FD_SET ()
select ()
timeout
close all open sockets
close ()
program ends
FD_ISSET ()
listening socket
accept ()
FD_SET ()
A
active connection - incoming data
recv ()
eof
recv data
close ()
send ()
FD_CLR ()
A
A
A

LINUX中的IO复用

- poll
  - poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

实验要求

- 设计时间轮方案，实现并提交实验结果

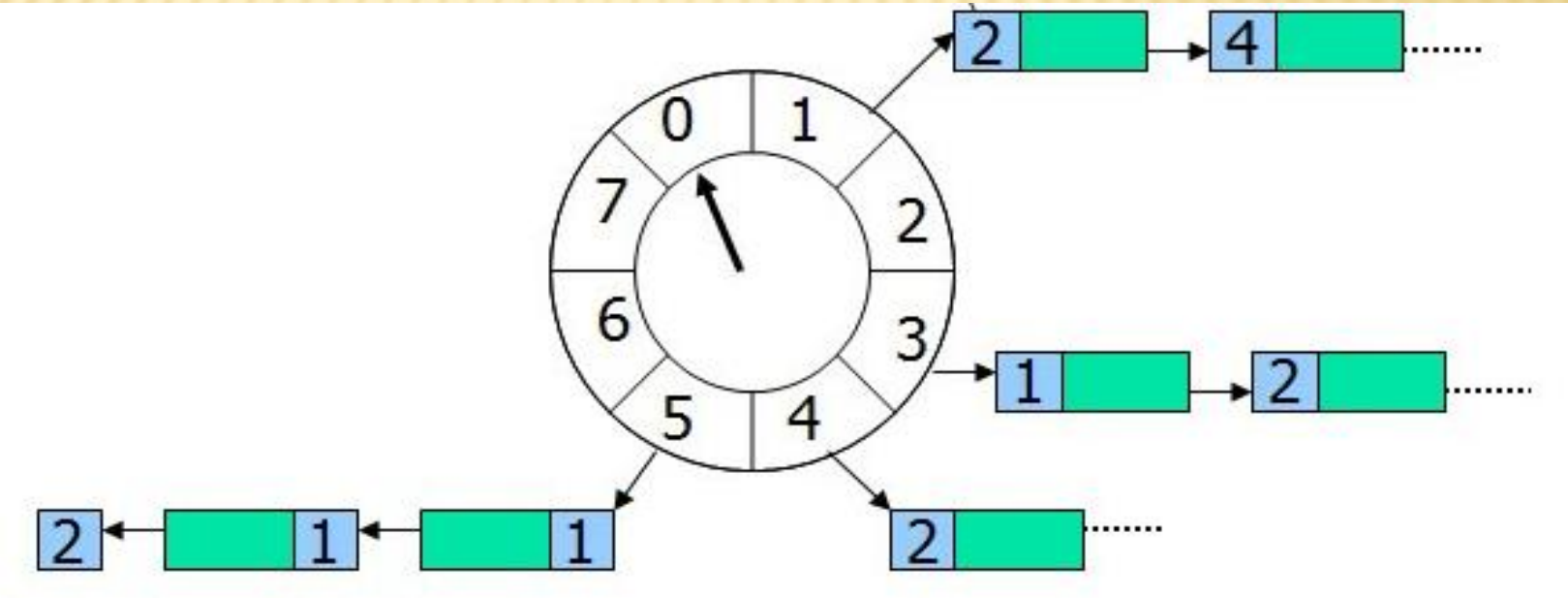
基本原理

- 设想有个时间钟，每隔一段时间（如100ms）有个滴答，每个滴答处对应一个list，存放了一些定时任务。当时间走到一个滴答处时就触发其对应的任务list。每个任务根据其超时时间放到相应滴答的list中，当一个任务list被滴答查看时，list中的所有任务的超时时间都被更新，并重新放入到时间轮盘中合适的位置。

- 为了方便讨论，约定一些术语。
- 时间轮有两个属性， frequence, wheelSize。
- 时间轮每隔一段时间有个滴答，每个滴答，轮子的指针会顺时针转动一次。这段时间的长度为 TimeWheel 的时间粒度，我们称之为 frequence, 也就是每过frequence ms 轮子的指针会转动一次。

- 每个时间轮都有一个轮子大小， 我们称之为wheelSize.
- 时间轮每一项都对应一个list， 这个list中的每一项都是一个定时器， 称为Timer。

- 如图所示，我们维护了一个wheelSize为8的TimeWheel。我们设置 frequence为100ms，当前指针指向第0项，每过100ms，指针顺时针转动1格。轮子转动一圈的时间为frequence * wheelSize = 8 * 100ms = 800ms.



- Timewheel的每一项维护一个timer队列。假设指针指向在第0项时，用户申请一个时间间隔为300ms的timer时，timerwheel 会将这个timer放入到第3项(图中的两个timer会变成3个timer)，当指针经过300ms后指向第3项时，会将第3项对应的list （3个timer)全部删除。
- 所以timeWheel 可以在o(1)常数时间内实现维护定时器。

二叉树

- 在计算机科学中，二叉树（英语：Binary tree）是每个节点最多只有两个分支（即不存在分支度大于2的节点）的树结构。通常分支被称作“左子树”或“右子树”。二叉树的分支具有左右次序，不能随意颠倒。

- 二叉树的第i层至多拥有2^(i - 1)个节点；深度为k的二叉树至多总共有2^k - 1个节点（定义根节点所在深度 k0=0），而总计拥有节点数符合的，称为“满二叉树”；深度为k有n个节点的二叉树，当且仅当其中的每一节点，都可以和同样深度k的满二叉树，序号为1到n的节点一对一对应时，称为完全二叉树。对任何一棵非空的二叉树T，如果其叶片（终端节点）数为n0，分支度为2的节点数为n2，则n0=n2 + 1。

二叉树

- 与普通树不同，普通树的节点个数至少为1，而二叉树的节点个数可以为0；普通树节点的最大分支度没有限制，而二叉树节点的最大分支度为2；普通树的节点无左、右次序之分，而二叉树的节点有左、右次序之分。

- 二叉树通常作为数据结构应用，典型用法是对节点定义一个标记函数，将一些值与每个节点相关联。这样标记的二叉树就可以实现二叉搜索树和二叉堆，并应用于高效率的搜索和排序。

链表

- 链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而顺序表相应的时间复杂度分别是O(logn)和O(1)。

- 使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表

- 在计算机科学中，链表作为一种基础的数据结构可以用来生成其它类型的数据结构。链表通常由一连串节点组成，每个节点包含任意的实例数据（data fields）和一或两个用来指向上一个/或下一个节点的位置的链接（"links"）。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的访问往往要在不同的排列顺序中转换。而链表是一种自我指示数据类型，因为它包含指向另一个相同类型的数据的指针（链接）。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

实现方式一：简单的TIMEWHEEL

- 直接使用timewheel的原理实现定时器。这么做的缺点是占用大量内存。
- 比如在frequence= 100ms 时，wheelSize = 1000时，轮子转一圈需要 100s，也就是可以插入的Timer最大是100s，想插入更大的Timer只能增大 wheelSize. 但若是申请一个 2^32 次方秒的时间间隔，就要申请很大的空间了。

实现方式二

- 第一种方法只能允许插入最大(frequence * wheelSize / 1000)秒的Timer, 更大的Timer不许插入了。这非常不灵活。我们对方法1进行修改，每个插入TimeWheel的Timer多加一个属性，rotationCount.
- rotationCount =

  Timer设定的时间 / (frequence * wheelSize)

- 每次指针经过自己时，rotationCount减1，rotationCount =0时再次遇到经过就删除。
- 这种方法成功地使timewheel 能够插入任意时间的timer 。但不能保证维护时的O(1)复杂度，因为轮子的每个Timer 可能在删除前访问多次。
- 由均摊分析知：时间复杂度为O(sum of rotationCount / total timers)
- 也就是说rotationCount 之和越小，执行会越快。

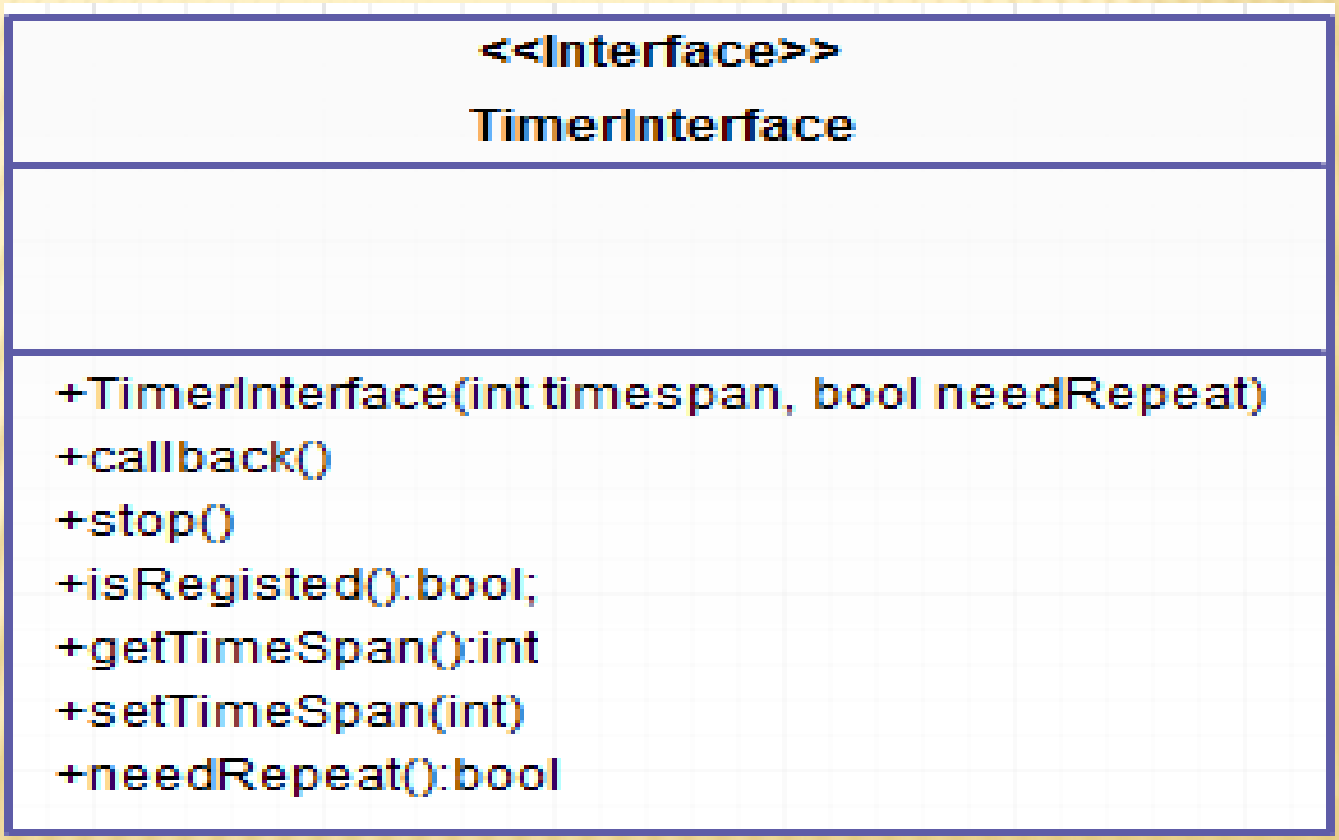
实现方法三：水表算法

- 这种算法是目前比较优秀的timewheel实现，这也是linux下的定时器实现的算法。简单来讲就是维护m个timewheel. 第一个轮子每个tick转动一次，第2个轮子在第一个轮子转动一圈后转动一次，第3个轮子在第2个轮子转动一圈后转动一次，依次类推。这样做可以复杂度折中为o(m) ，又因为m通常很小，可以将复杂度看做 o(1). Linux 下 的实现时使用了5个轮子，每个timewheel的大小分别为 256, 64, 64, 64, 64. 可以表示2^32 以内的Timer。

实验步骤：接口设计

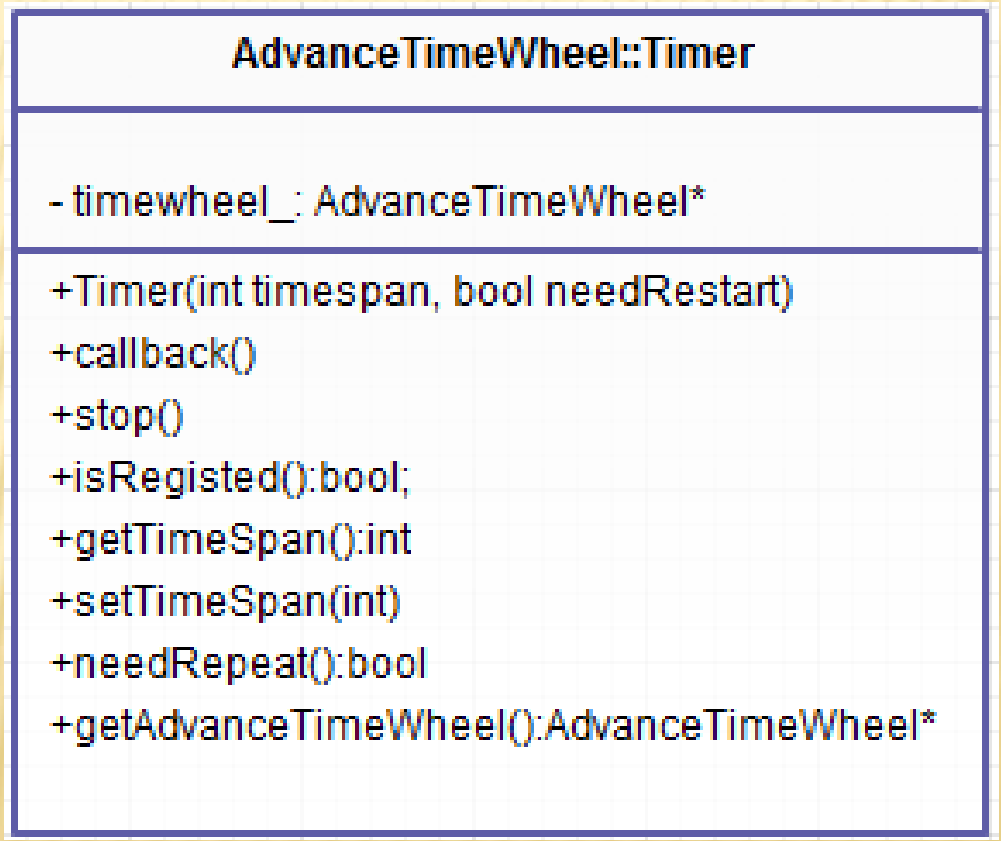
- 定时器接口，是一个抽象类。给定一个时间间隔，从当前时间开始计时，时间到了以后(when expired)，执行callback()业务逻辑函数。
- 共有两种定时器:
  - needRepeat= true时，构造出一个repeated-timer，不停地以timespan为周期执行callback()函数，直到显式地调用stop() 从时间轮中取出。
  - needRepeat= false时，构造出一个one-shot timer，执行完一次callback()之后自动释放。只执行一次业务逻辑后自动从TimeWheel中删除。

- 可以将Repeated-timer扩展为执行特定n次数的Timer。只需要在callback()中增加一个静态计数器变量，满足n次后在run()方法内执行stopTimer(timer)即可。
- isRegisted() 函数：用来判断该Timer是否已经注册到timewheel上运行。
- getTimeSpan()/setTimeSpan()：用来查询或设置该定时器当前的超时时间。参数以毫秒为单位。
- needRepeat()：用来判断是否这个定时器是repeated-timer.


<<Interface>>
TimerInterface
+TimerInterface(int timespan, bool needRepeat)
+callback()
+stop()
+isRegisted():bool;
+getTimeSpan():int
+setTimeSpan(int)
+needRepeat():bool

- 该接口对应的实现类为:

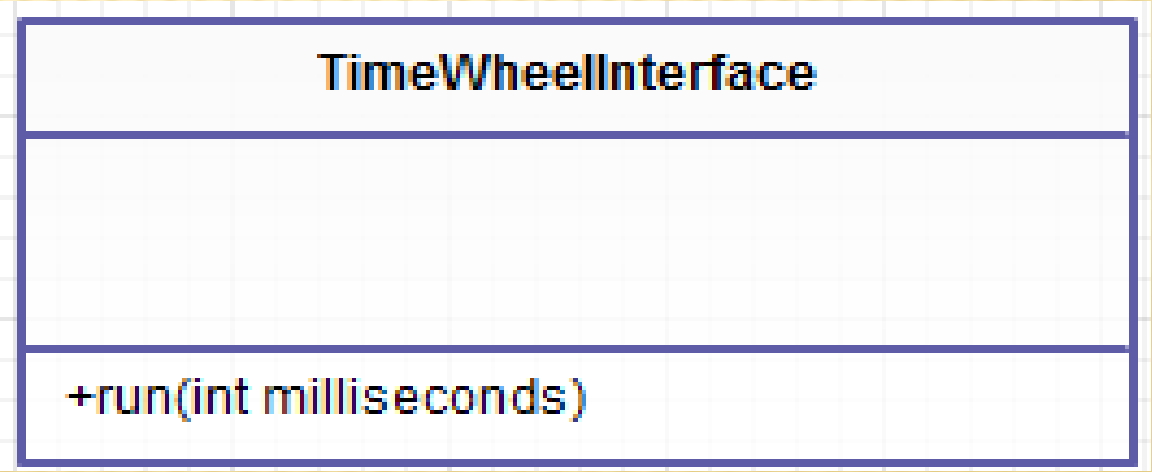
AdvanceTimeWheel::Timer



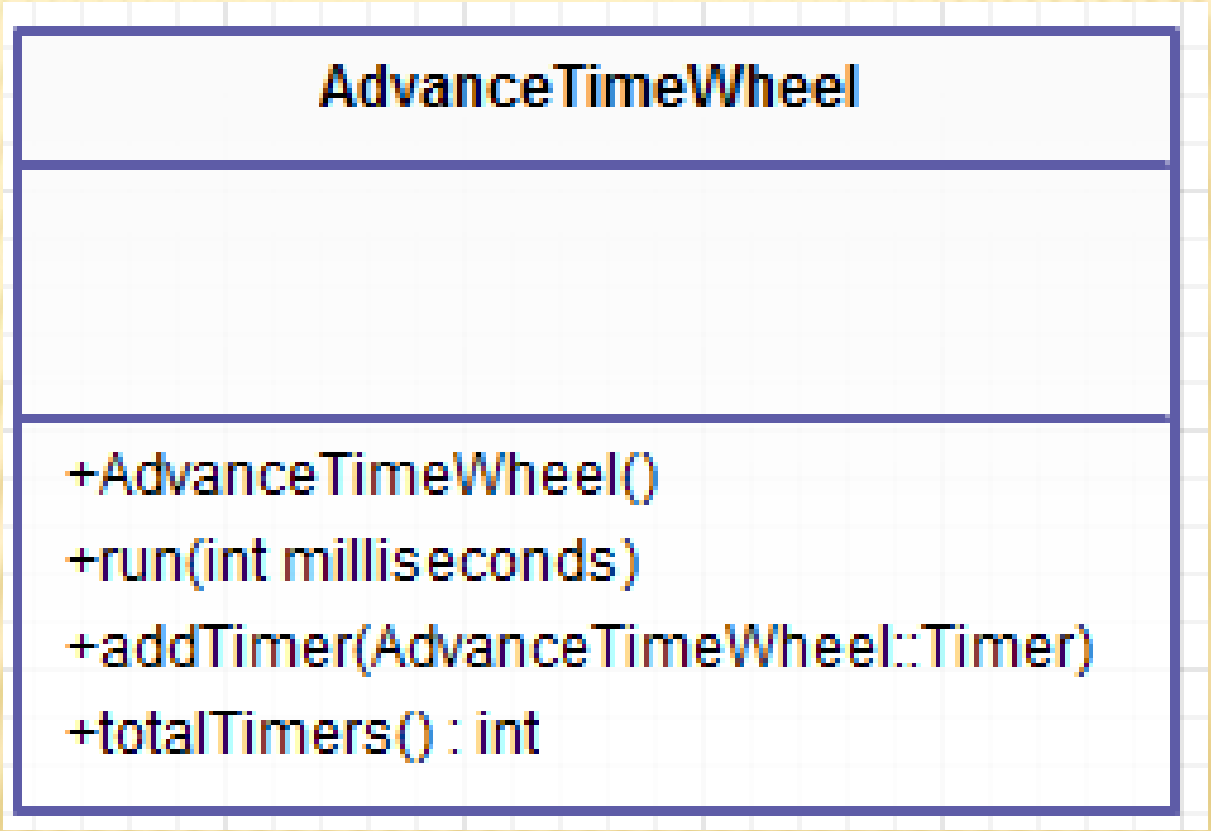
- 用户负责派生AdvanceTimeWheel::Timer类，并覆盖其callback()方法。该方法用来表示具体的业务逻辑。在Timer expired后，回调callback()方法。
- 注意：除了具体的事务逻辑外，也可以在callback()中向AdvanceTimeWheel添加新的Timer，或停止旧的Timer，你也可以把自己从AdvanceTimeWheel中删除。

管理定时器的数据结构

- TimeWheelInterface是一个虚类，具体的实现类实现其run()方法。
- 共有两个TimeWheelInterface的实现类，TimeWheel和AdvanceTimeWheel。分别对应两种不同的时间轮的实现。

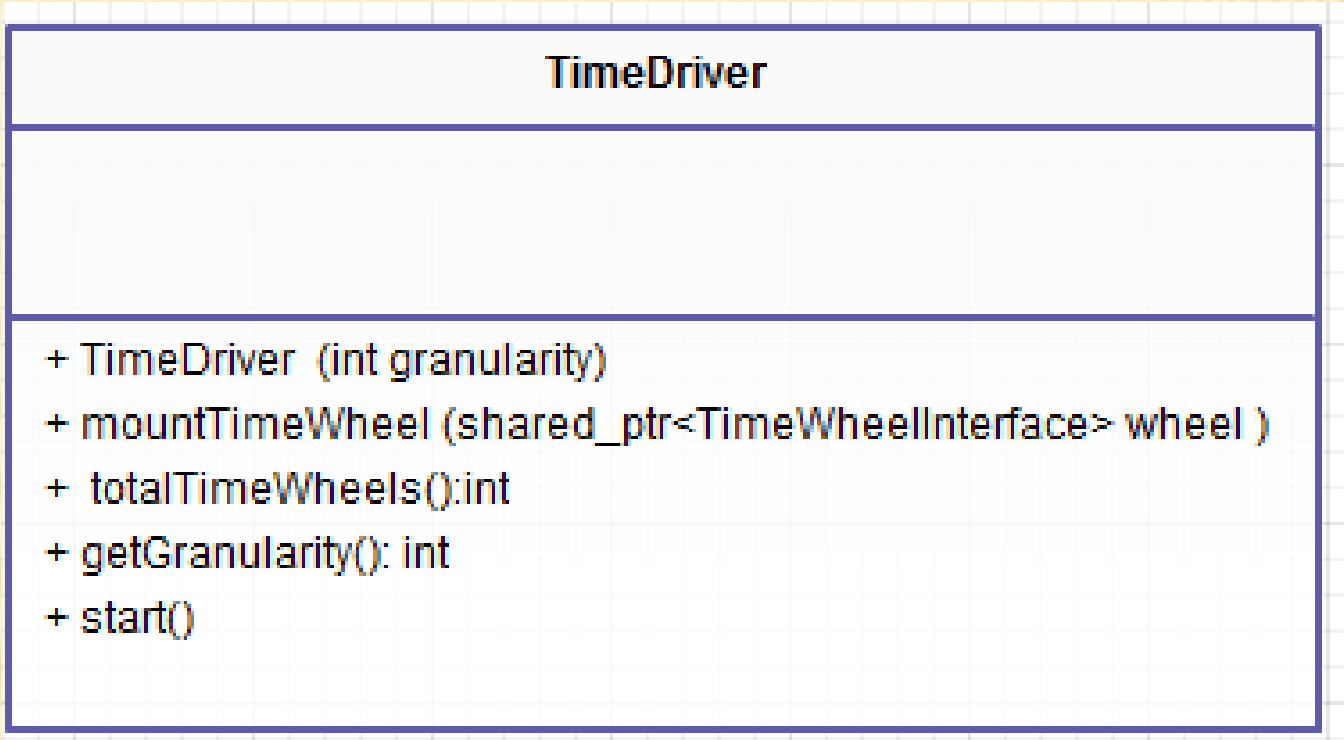


- 该接口对应的实现是AdvanceTimeWheel.
- 该类是时间轮的水表算法的实现，接口简单，用户不需要调整参数就可以获得很好的性能。
- void addTimer(AdvanceTimeWheel::Timer&) 可以在O(m)时间复杂度向timewheel中添加一个新的timer。
- int totalTimers(): 返回当前的定时器的个数， 注意这里复杂度是o(n)的。频繁使用会影响性能。
- void run(int milliseconds): 驱动时间轮运行millisecdonds 毫秒。


AdvanceTimeWheel
+AdvanceTimeWheel()
+run(int milliseconds)
+addTimer(AdvanceTimeWheel::Timer)
+totalTimers() : int

时间轮的驱动类

- 实现时使用内核定时器进行驱动。一个TimeDriver可以使用mountTimeWheel()函数来挂载很多个timewheel. 然后可以同时驱动多个timewheel进行转动。
- 它是一个抽象类，所有的实现类派生TimeDriver,实现start()方法
- 用户可以根据需求选择合适的TimeDriver实现类。



- 构造函数为 explicit TimeDriver(long granularity = DEFAULT_GRANULARITY)；granularity 时间粒度，以毫秒为单位， 默认是 DEFAULT_GRANULARITY，1ms。

- mountTimeWheel()： 挂载timewheel, 允许TimeDriver同时挂载并驱动多个TimeWheel。
- int totalWheels()： 返回当前挂载了多少个TimeWheel。
- long getGranularity()： 返回该TimeDriver自身的时间粒度。
- start()：用户负责调用此函数。 调用此函数后，TimeDriver开始驱动所有的timewheel开始计时。

The End