零声教育出品 Mark 老师 QQ：2548898954

定时器应用

- 心跳检测

  keepalive

  应用层发送心跳包
- 技能冷却
- 倒计时
- 其他需要延时处理的功能

定时器触发方式

> 对于服务端来说，驱动服务端业务逻辑的事件包括网络事件、定时事件、以及信号事件；通常网络事件和定时事件会进行协同处理；定时器触发形式通常有两种：
>
> - 利用 IO 多路复用系统调用最后一个参数（超时），来触发检测定时器；
> - 利用 timerfd，将定时检测作为 IO 多路复用当中的事件进行处理；

```
// 网络事件和定时事件在一个线程中处理
while (!quit) {
    int timeout = get_nearest_timer() - now();
    if (timeout < 0) timeout = -1;
    int nevent = epoll_wait(epfd, ev, nev, timeout);
    for (int i=0; i<nevent; i++) {
        // ... 处理网络事件
```

```
    }
    // 处理定时事件
    update_timer();
}

// 网络事件和定时事件在不同线程中处理
void * thread_timer(void *thread_param) {
    init_timer();
    while (!quit) {
        update_timer();
        sleep(t);
    }
    clear_timer();
    return NULL;
}
pthread_create(&pid, NULL, thread_timer,
&thread_param);
```

timerfd

```
#include <sys/timerfd.h>
// 创建一个定时器文件描述符，可以像网络 IO 一样，将这个 fd
交由 IO 多路复用来管理
int timerfd_create(int clockid, int flags);
// 设置一个触发时间，IO 多路复用将会检测这个过期事件，然后
通知应用程序定时事件就绪
int timerfd_settime(int fd, int flags,
                    const struct itimerspec
*new_value,
                    struct itimerspec
*old_value);
```

定时器设计

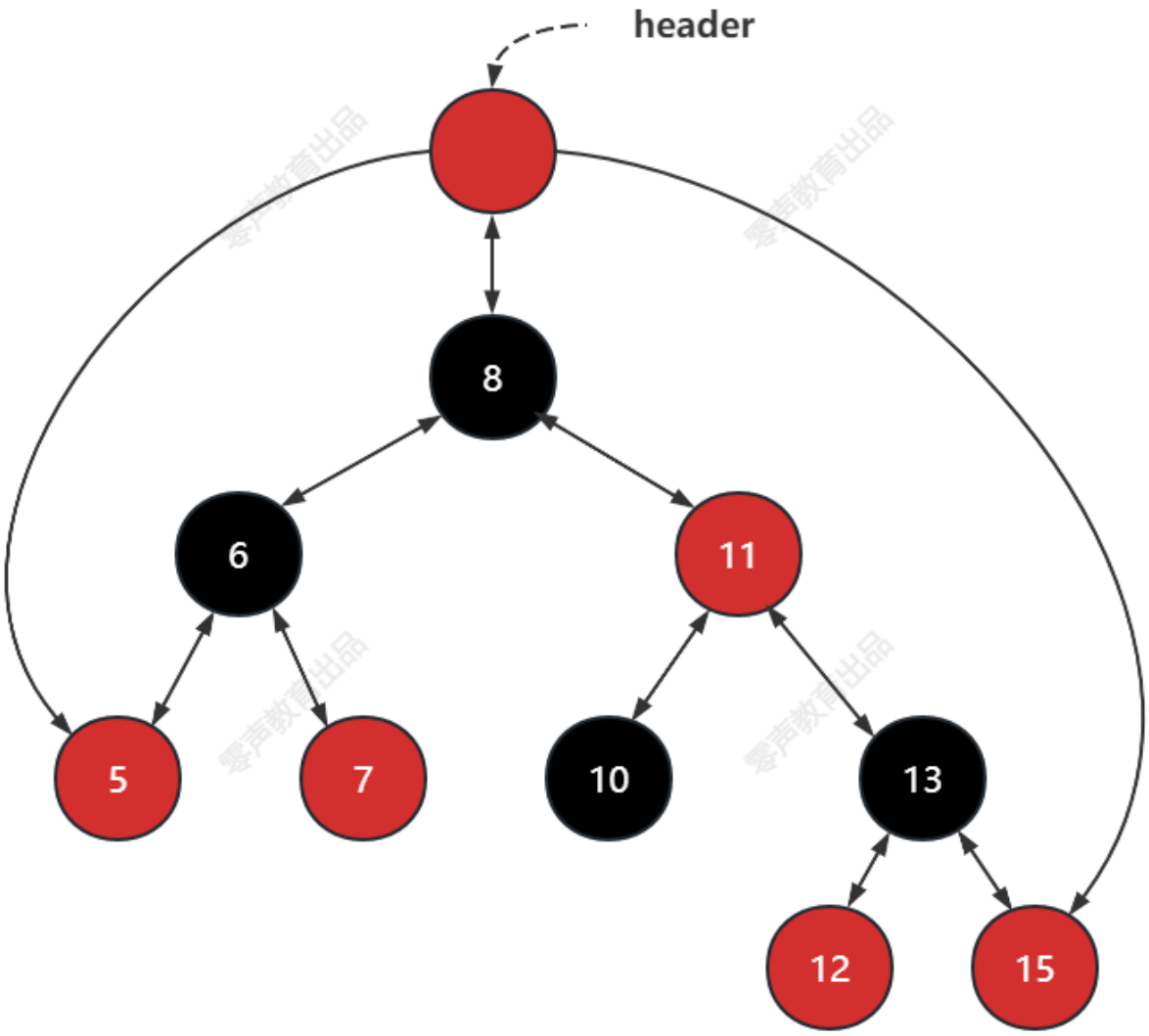
接口设计

```
// 初始化定时器
void init_timer();
// 添加定时器
Node* add_timer(int expire, callback cb);
// 删除定时器
bool del_timer(Node* node);
// 找到最近要触发的定时任务
Node* find_nearest_timer();
// 更新检测定时器
void update_timer();
// 清除定时器
// void clear_timer();
```

数据结构设计

> 对定时任务的组织本质是要对定时任务优先级的处理；由此产生两类数据结构；
>
> - 按触发时间进行顺序组织
>   - 要求数据结构有序（红黑树、跳表），或者相对有序（最小堆）；
>   - 能快速查找最近触发的定时任务；
>   - **需要考虑怎么处理相同时间触发的定时任务**；
> - 按执行顺序进行组织
>   - 时间轮

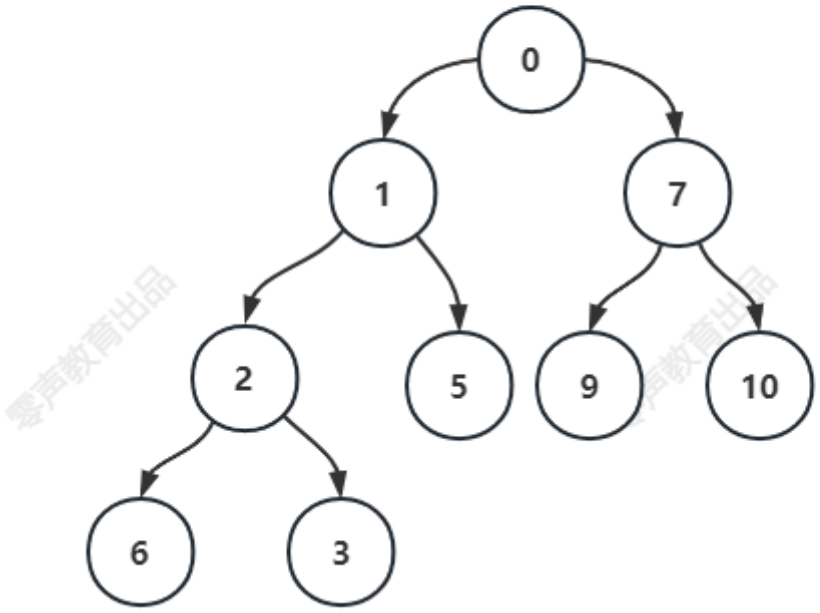
红黑树



最小堆

> 满二叉树：所有的层节点数都是该层所能容纳节点的最大数量（满足 2^n）
>
> 完全二叉树：若二叉树的深度为 h，去除了 h 层的节点，就是一个满二叉树；且 h 层都集中在最左侧排列；
>
> 最小堆：约束了父子之间的大小关系
>
> - 是一颗完全二叉树；（可以数组来存储）
> - 某一个节点的值总是小于等于它的子节点的值；
> - 堆中任意一个节点的子树都是最小堆；

0	1	7	2	5	9	10	6	3



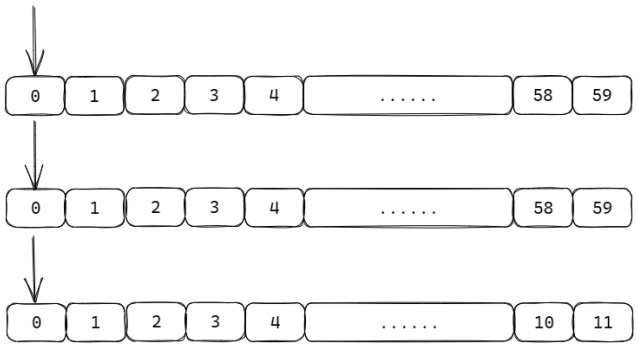
最小堆添加节点

> 为了满足完全二叉树的定义，往二叉树最高层沿着最左侧添加一个节点；然后考虑是否能**上升**操作；如果此时添加值为 4 的节点，4 节点是 5 节点的左子树；4 比 5 小，4 和 5 需要交换值；

最小堆删除节点

> 删除操作需要先查找是否包含这个节点；确定存在后，交换最后一个节点，先考虑能否执行**下降**操作，否则执行**上升**操作；最后删除最后一个节点；
>
> 例如：删除 1 号节点，则需要**下沉**操作；删除 9 号节点，则需要**上升**操作；

时间轮




```
struct node {
    struct node *next;
    uint32_t expire;  //. tick + dis
    handler_pt callback;
    uint8_t cancel;
};
```

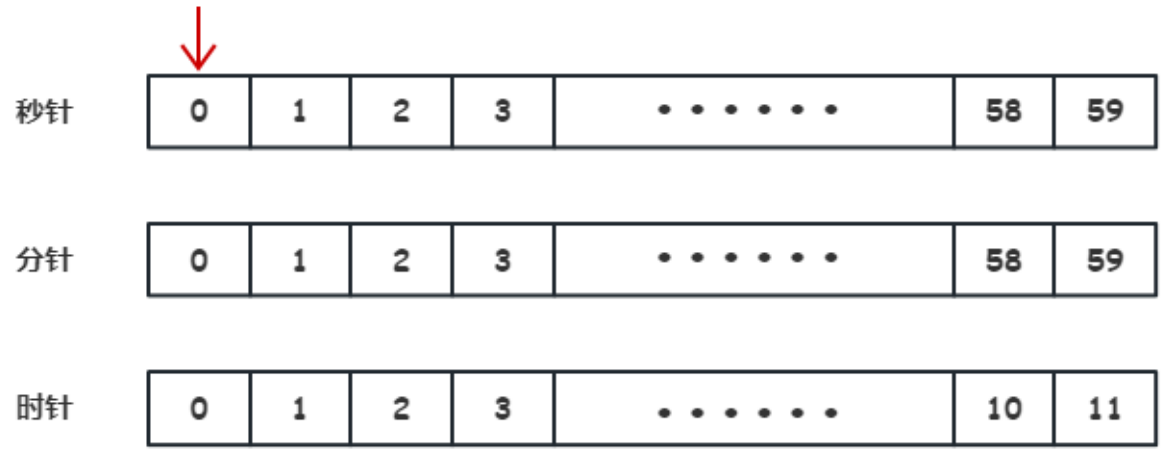
* 添加任务
根据 dis 来决定存放具体位置
先根据 dis 的范围找到所在层级
再根据 tick（当前时间指针）做偏移
存放在具体某个槽位中，槽位中多个节点通过链表链接
* 时间指针移动
只有第一层时间指针移动到哪，从该槽位取出所有过期任务，去分发或执行
前一层指针移动一圈，当层指针移动一格
除了第一层，其他层级指针移动一格，则取出任务往上一层级重新映射

* 时间指针如何移动?
第一层时间轮的时间指针代表整个时间轮的时间精度（误差）
假设精度为 1s，可以采用 sleep(0.25s) 的方式来驱动
时间指针移动。为什么采用 0.25s，这样可以修正时间指针。

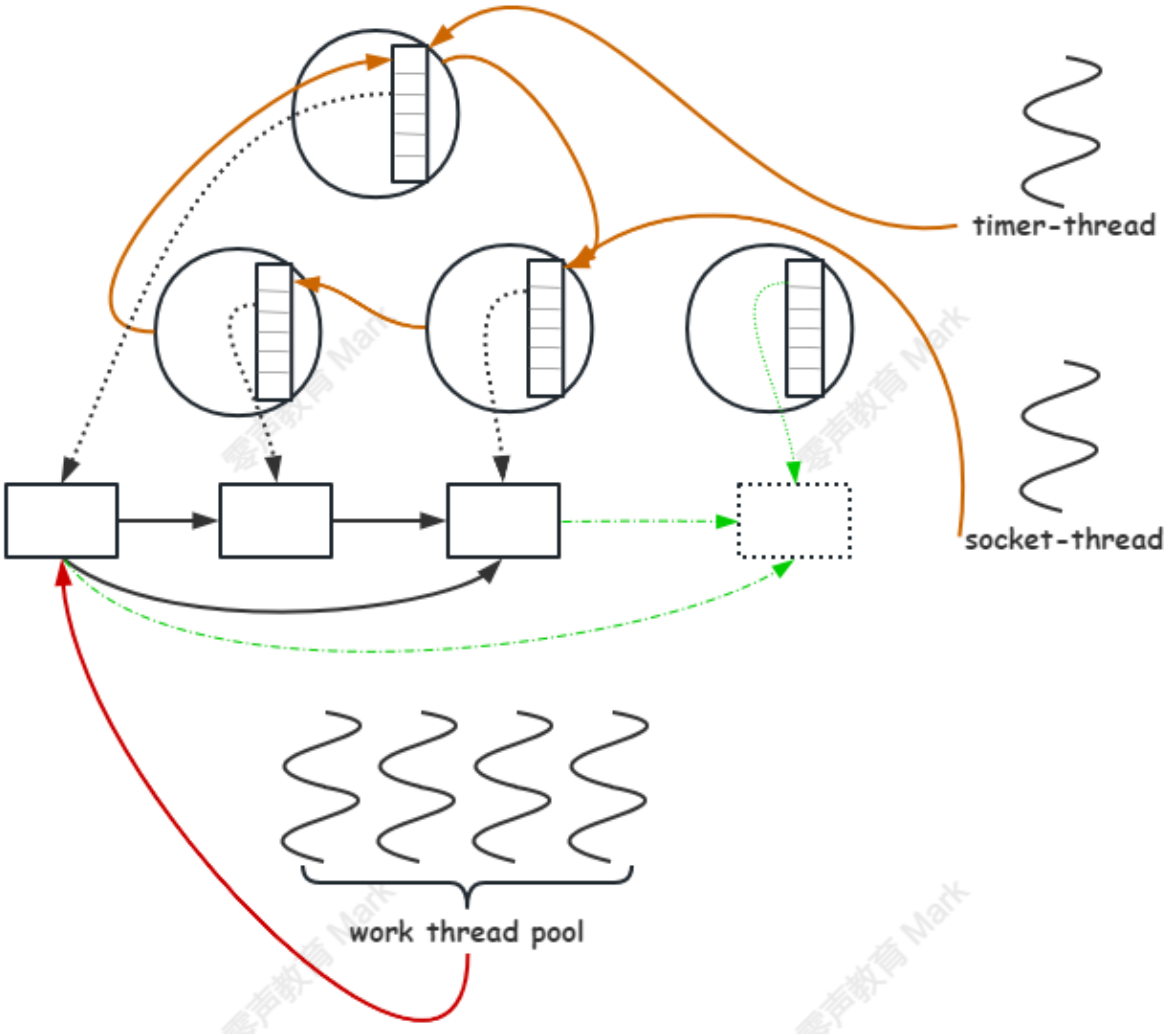
* 如何重新映射?
dis` = node.expire – tick`(当前时间指针) --[[tick` 和 添加任务中 tick 不是一个时间]]
根据 dis 添加任务到时间轮（必然会存放在上一层级）

注意：执行任务是耗时的，会影响时间指针移动，从而造成定时任务延误。所以，通常
时间轮组件只做延时任务检测，具体执行丢给其他线程处理。
一个时间轮配一个线程，在该线程中只做任务超时检测。
其他线程往时间轮结构中添加任务。

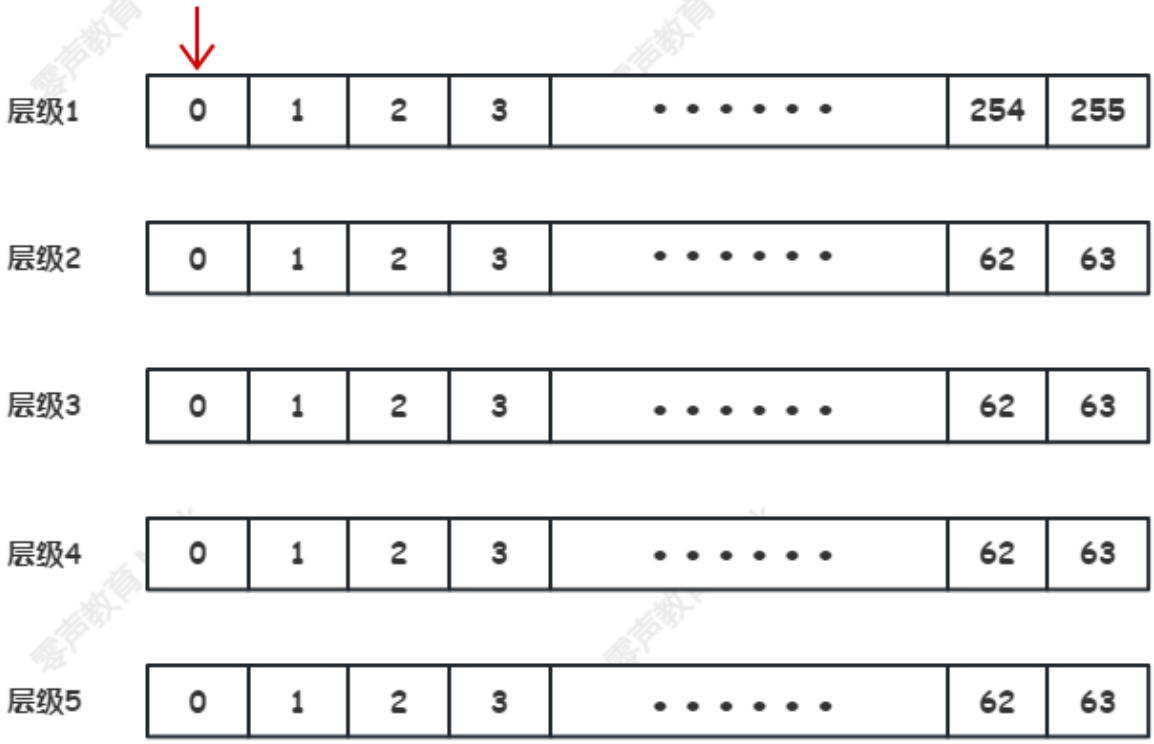
* 时间轮线程安全问题?
因为时间轮的操作都是 O(1) 的时间复杂度，可以考虑使用细粒度的锁（自旋锁）。


秒针
0
1
2
3
58
59
分针
0
1
2
3
58
59
时针
0
1
2
3
10
11

运行图


timer-thread
socket-thread
work thread pool

原理图


层级1
0
1
2
3
254
255
层级2
0
1
2
3
62
63
层级3
0
1
2
3
62
63
层级4
0
1
2
3
62
63
层级5
0
1
2
3
62
63