零声学院 Mark 老师 QQ: 2548898954

定时器应用

- 心跳检测
- 技能冷却
- 武器冷却
- 倒计时
- 其它需要使用超时机制的功能

定时器概述

> 对于服务端来说，驱动服务端逻辑的事件主要有两个，一个是网络事件，另一个是时间事件；
>
> 在不同框架中，这两种事件有不同的实现方式；
>
> 第一种，网络事件和时间事件在一个线程当中配合使用；例如nginx、redis；
>
> 第二种，网络事件和时间事件在不同线程当中处理；例如skynet；

```
// 第一种
while (!quit) {
    int now = get_now_time();// 单位：ms
    int timeout = get_nearest_timer() - now;
    if (timeout < 0) timeout = 0;
    int nevent = epoll_wait(epfd, ev, nev, timeout);
    for (int i=0; i<nevent; i++) {
        //... 网络事件处理
    }
    update_timer(); // 时间事件处理
}
```

```
// 第二种 在其他线程添加定时任务
void* thread_timer(void * thread_param) {
    init_timer();
    while (!quit) {
        update_timer(); // 更新检测定时器，并把定时事件发送到消息队列中
        sleep(t); // 这里的 t 要小于 时间精度
    }
    clear_timer();
    return NULL;
}
pthread_create(&pid, NULL, thread_timer, &thread_param);
```

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);
// 找到最近要发生的定时任务
Node* find_nearest_timer();
// 更新检测定时器
void update_timer();
// 清除定时器
// void clear_timer();
```

数据结构选择

- 红黑树

  > 对于增删查，时间复杂度为 $O(log_2n)$；对于红黑树最小节点为最左侧节点，时间复杂度为 $O(log_2n)$；

- 最小堆

  > 对于增查，时间复杂度为 $O(log_2n)$；对于删时间复杂度为 $O(n)$，但是可以通过辅助数据结构（map或者hashtable来快速索引节点）来加快删除操作；对于最小节点为根节点，时间复杂度为 $O(1)$；

- 跳表

  > 对于增删查，时间复杂度为 $O(log_2n)$；对于跳表最小节点为最左侧节点，时间复杂度为 $O(1)$；但是空间复杂度比较高，为$O(1.5n)$；

- 时间轮

  > 对于增删查，时间复杂度为 $O(1)$；查找最小节点也为 $O(1)$；

要点

> 1. 有序的结构，且增加删除操作不影响该结构有序；
> 2. 能快速查找最小节点；
> 3. 时间轮增加操作只从单个定时任务触发，忽略定时任务之间的大小关系；而红黑树、最小堆、跳表的有序性依赖定时任务之间的大小关系；

红黑树

细节

```
void ngx_rbtree_insert_timer_value(ngx_rbtree_node_t *temp,
ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
{
    ngx_rbtree_node_t  **p;

    for ( ;; ) {
        // 这里是重点
        p = ((ngx_rbtree_key_int_t) (node->key - temp->key) < 0)
            ? &temp->left : &temp->right;
        if (*p == sentinel) {
            break;
        }
        temp = *p;
    }

    *p = node;
    node->parent = temp;
    node->left = sentinel;
    node->right = sentinel;
    ngx_rbt_red(node);
}
```

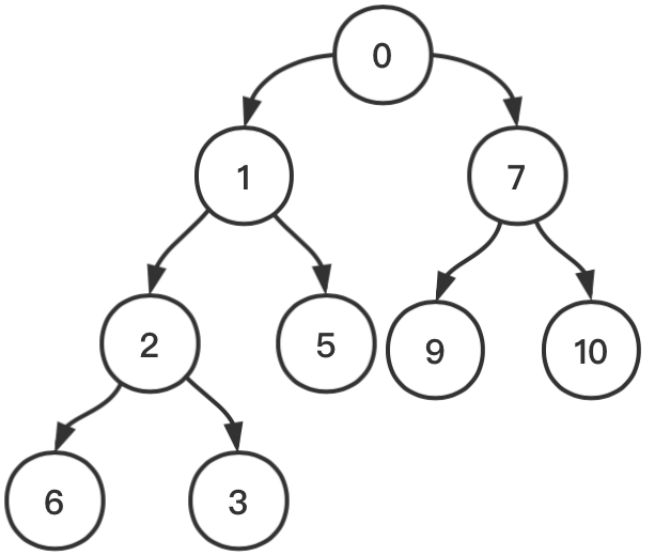
> STL中 `map` 结构采用的是红黑树来实现，但是定时器不要使用 `map` 结构来实现，因为可能多个定时任务需要同时被触发，`map` 中的key是惟一的；
>
> 红黑树的节点同时包含 `key` 和 `val`，红黑树节点的有序由 `key` 来决定的；插入节点的时候，通过**比较key**来决定节点存储位置；红黑树的实现并没有要求 `key` 唯一；如上代码示例，`for` 循环中 `(node->key - temp->key) < 0)? &temp->left : &temp->right;` 当key相同的时候取值为 `temp->right`；思考：`map` 结构中插入节点这里如何操作?

最小堆

概述

> **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$2^n;n >= 0$）；
>
> **完全二叉树**：若二叉树的深度为 `h`，除了 `h` 层外，其他层的**节点数**都是该层所能容纳节点的最大数量（满足$2^n;n >= 0$），且 `h` 层都集中在最左侧；
>
> **最小堆**：
>
> 1. 是一颗完全二叉树；
> 2. 某一个节点的值总是小于等于它的子节点的值；
> 3. 堆中每个节点的子树都是最小堆；

0	1	7	2	5	9	10	6	3



增加操作

> 为了满足完全二叉树定义，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` 需要交换位置；

删除操作

> 删除操作需要先查找是否包含这个节点，最小堆的查找效率是 $O(n)$；查找之后，交换最后一个节点，先考虑下降操作，如果操作失败则上升操作；最后删除最后一个节点；
>
> 例如：假设删除 `1` 号节点，则需要下沉操作；假设删除 `9` 号节点，则需要上升操作；

时间轮

> 从时钟表盘出发，如何用数据结构来描述秒表的运转；

> int seconds[60]; // 数组来描述表盘刻度；
>
> ++tick $\%$ 60；每秒钟 ++tick 来描述秒针移动；对 $tick\%60$ 让秒针永远在 $[0, 59]$ 间移动；
>
> 对于时钟来说，它的时间精度（最小运行单元）是1秒；

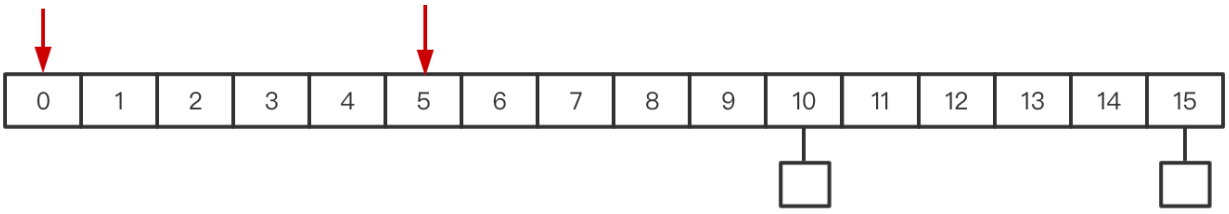
单层级时间轮

背景

> 心跳检测：
>
> 客户端每 5 秒钟发送心跳包；服务端若 10 秒内没收到心跳数据，则清除连接；
>
> 实际在开发过程中，若收到除了心跳包的其他数据，心跳检测也算通过，在这里为了简化流程，只判断心跳包；
>
> 作为对比：我们假设使用 map<int, conn*> 来存储所有连接数；每秒检测 map 结构，那么每秒需要遍历所有的连接，如果这个map结构包含几万条连接，那么我们做了很多无效检测；考虑极端情况，刚添加进来的连接，下一秒就需要去检测，实际上只需要10秒后检测就行了；那么我们考虑使用时间轮来检测；
>
> **注意**：这个例子只是用来帮助理解时间轮，不代表实际解决方案；

设计

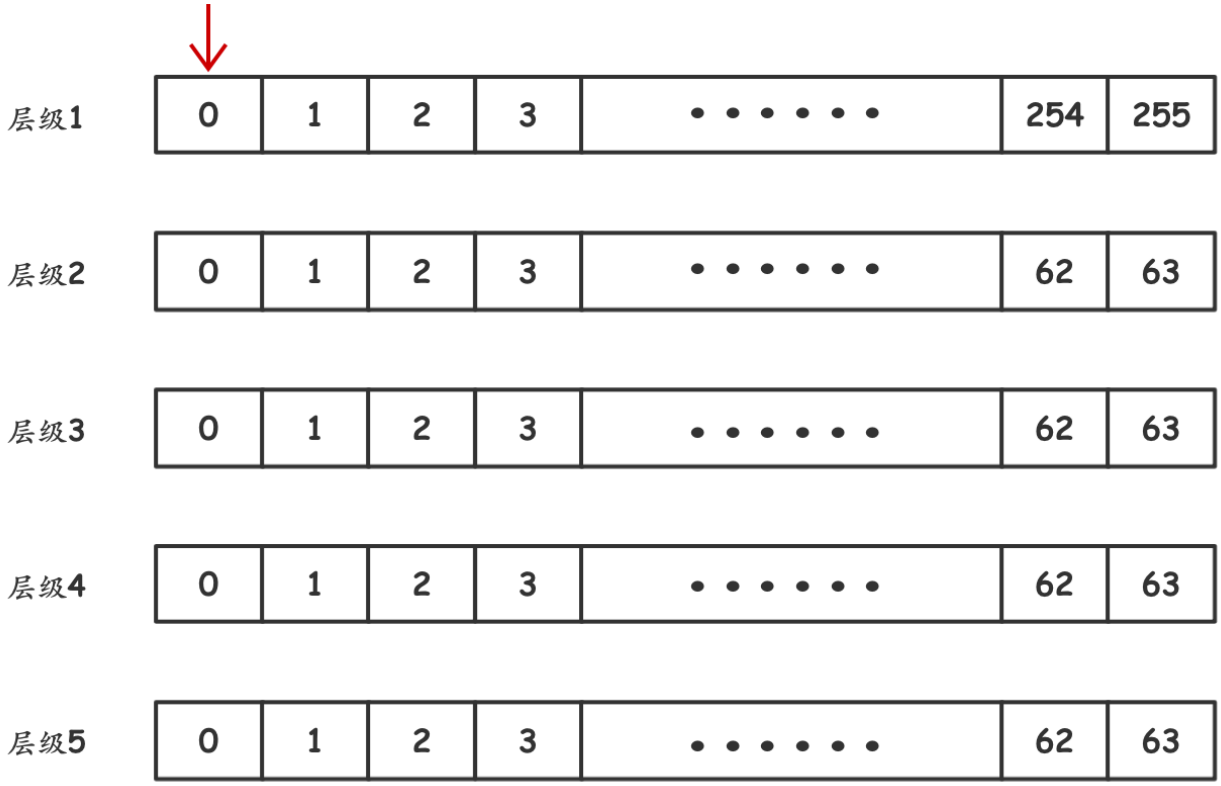
> 1. 准备一个数组存储连接数据；那么数组长度设置为多少?
> 2. 考虑一秒内添加了多条连接，那么可以参考 `hash` 结构处理冲突的方式，用链表链接起来；
> 3. 回到 `1` 中的问题，如果想 `2` 中链表稀疏，将数组长度设置大一些；如果想紧凑些，则将数组长度设置小些（但是必须大于10）；
> 4. 假设我们设置数组长度为 11；那么检测指针的移动可描述为 ++point $\%$ 11；
>
> $m\%n = m - n \times floor(\frac{m}{n})$
>
>   优化：将 n 替换为 2^k，这里 2^k 恰好大于 n；这样以来 $m\%2^k$ 可以转化为 $m\&(2^k - 1)$；
>
>   所以我们可以选择 16（2^4），那么检测指针移动可优化为 ++point$\&(16 - 1)$；
> 5. 考虑到正常情况下 5 秒钟发送一次心跳包，10 秒才检测一次，如下图到索引为 10 的时候并不能踢掉连接；所以需要每收到一次心跳包则 `used++`，每检测一次 `used--`；当检测到 `used == 0` 则踢掉连接；



多层级时间轮

背景

> 参照时钟表盘的运转规律，可以将定时任务根据触发的紧急程度，分布到不同层级的时间轮中；
>
> 假设时间精度为 `10ms` ；在第 `1` 层级每 `10ms` 移动一格；每移动一格执行该格子当中所有的定时任务；
>
> 当第 `1` 层指针从 255 格开始移动，此时层级 `2` 移动一格；层级 `2` 移动一格的行为定义为，将该格当中的定时任务重新映射到层级 `1` 当中；同理，层级 `2` 当中从 `63` 格开始移动，层级 `3` 格子中的定时任务重新映射到层级 `2`；以此类推层级 `4` 往层级 `3` 映射，层级 `5` 往层级 `4` 映射；
>
> 如何重新映射？定时任务的过期时间对上一层级的长度取余分布在上一层级不同格子当中；



添加节点

```
void add_node(timer_t *T, timer_node_t *node) {
    uint32_t time=node->expire;
    uint32_t current_time=T->time;
    uint32_t msec = time - current_time;
    if (msec < TIME_NEAR) { //[0, 0x100)
        // time % 256
        link(&T->near[time&TIME_NEAR_MASK],node);
    } else if (msec < (1 << (TIME_NEAR_SHIFT+TIME_LEVEL_SHIFT)))
{//[0x100, 0x4000)
        // floor(time/2^8) % 64
        link(&T->t[0][((time>>TIME_NEAR_SHIFT) & TIME_LEVEL_MASK)],node);
```

```
    } else if (msec < (1 << (TIME_NEAR_SHIFT+2*TIME_LEVEL_SHIFT))) {//[0x4000, 0x100000)
        // floor(time/2^14) % 64
        link(&T->t[1][((time>>(TIME_NEAR_SHIFT + TIME_LEVEL_SHIFT)) & TIME_LEVEL_MASK)],node);
    } else if (msec < (1 << (TIME_NEAR_SHIFT+3*TIME_LEVEL_SHIFT))) {//[0x100000, 0x4000000)
        // floor(time/2^20) % 64
        link(&T->t[2][((time>>(TIME_NEAR_SHIFT + 2*TIME_LEVEL_SHIFT)) & TIME_LEVEL_MASK)],node);
    } else {//[0x4000000, 0xffffffff]
        // floor(time/2^26) % 64
        link(&T->t[3][((time>>(TIME_NEAR_SHIFT + 3*TIME_LEVEL_SHIFT)) & TIME_LEVEL_MASK)],node);
    }
}
```

重新映射

```
void
timer_shift(timer_t *T) {
    int mask = TIME_NEAR;
    uint32_t ct = ++T->time; // 第一层级指针移动 ++ 一次代表10ms
    if (ct == 0) {
        move_list(T, 3, 0);
    } else {
        // floor(ct / 256)
        uint32_t time = ct >> TIME_NEAR_SHIFT;
        int i=0;
        // ct % 256 == 0 说明是否移动到了 不同层级的 最后一格
        while ((ct & (mask-1))==0) {
            int idx=time & TIME_LEVEL_MASK;
            if (idx!=0) {
                move_list(T, i, idx); // 这里发生重新映射，将i+1层级idx格子中的定时任务重新映射到i层级中
            }
            mask <<= TIME_LEVEL_SHIFT;
            time >>= TIME_LEVEL_SHIFT;
            ++i;
        }
    }
}
```