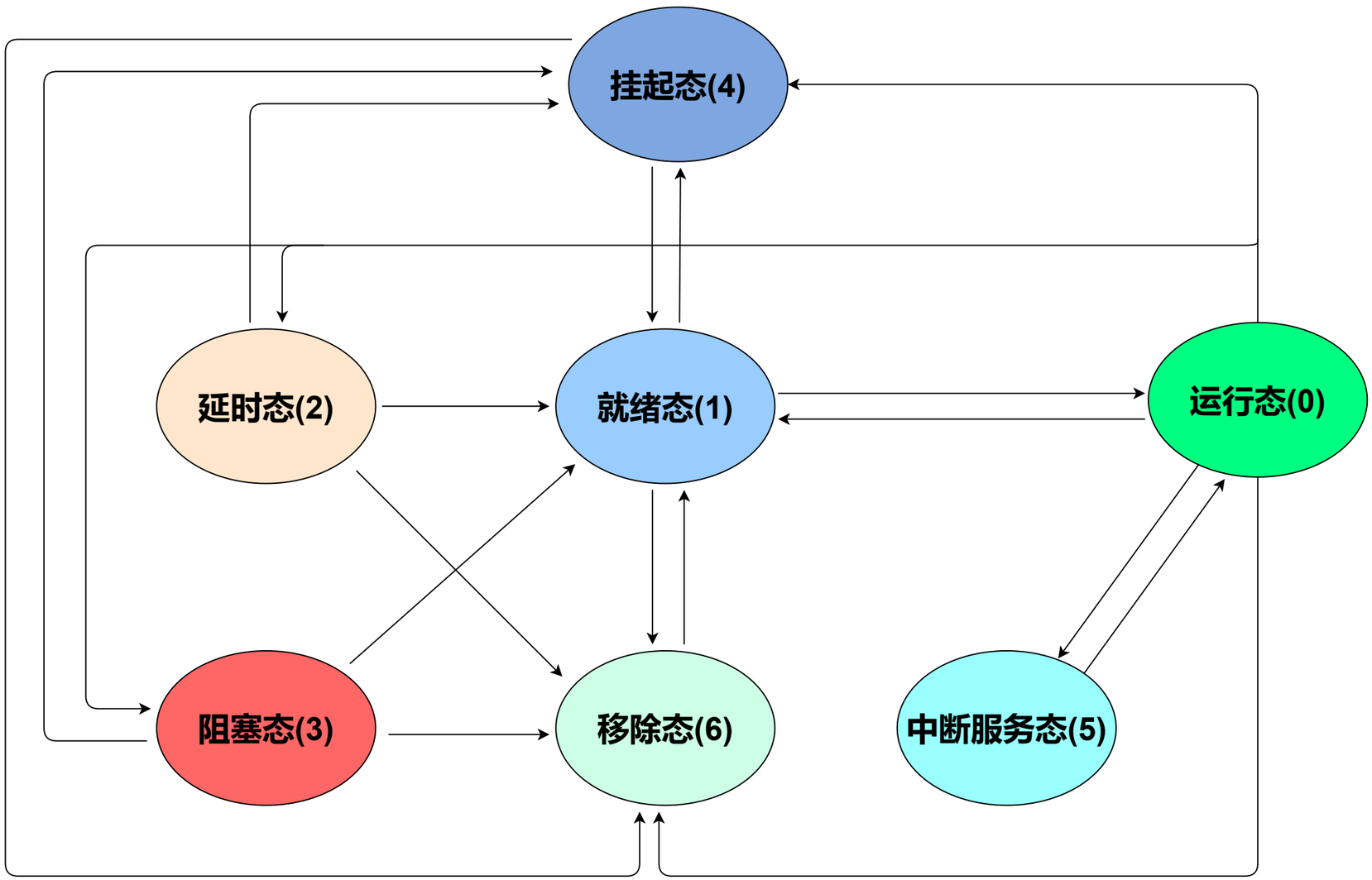
预览

RT-Plain™

一款简单的实时操作系统V2.2

该项目由ElecM、CestLaVie共同发起，旨在发展开源精神



RT-Plain任务状态图

RT-Plain状态说明

挂起态：使用挂起函数被挂起的任务状态。

延时态：使用延时函数而进入的状态

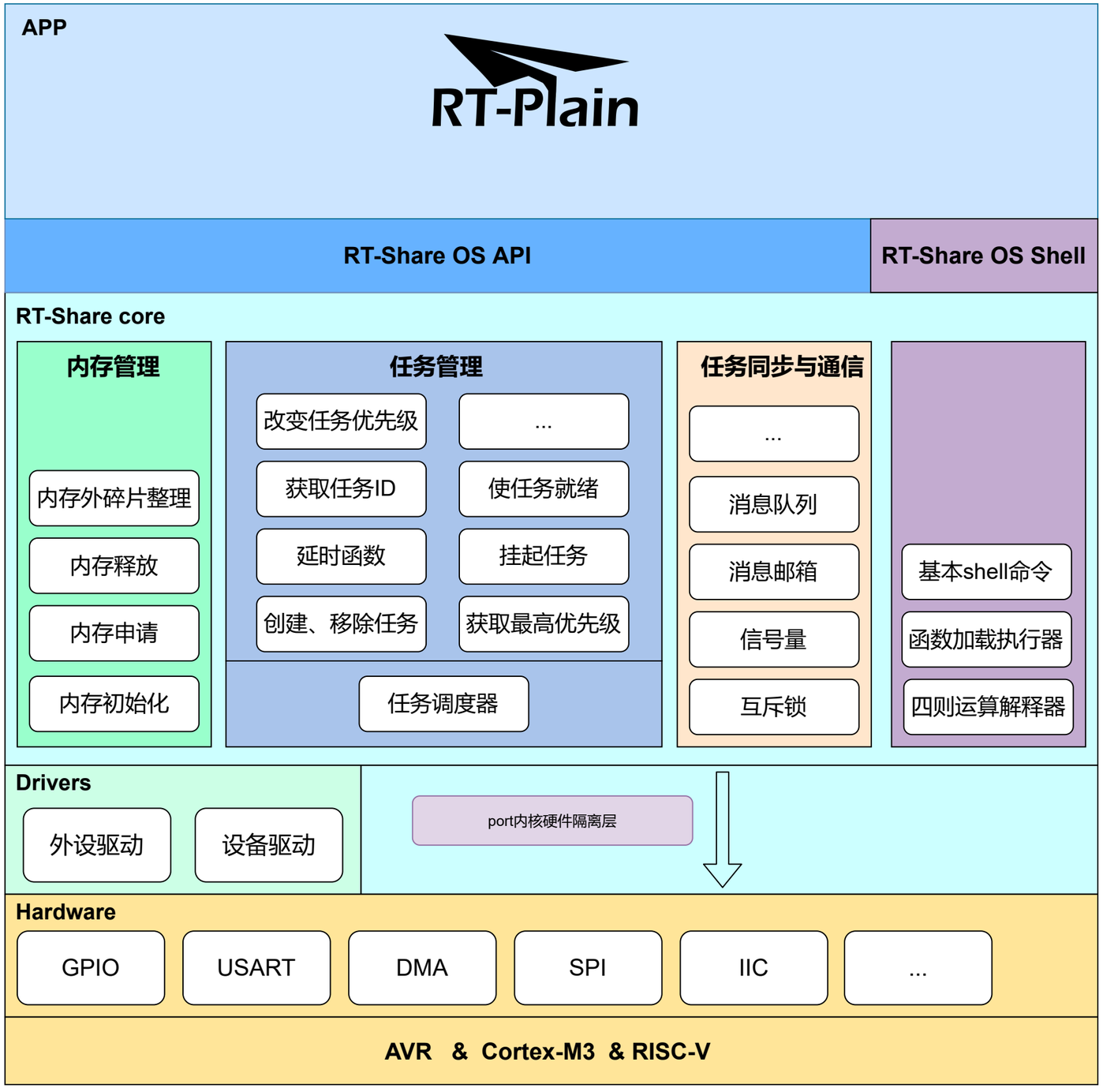
移除态：使用移除函数移除任务，而使该任务进入移除态，该任务除非使用创建任务函数，否则永远无法再运行。

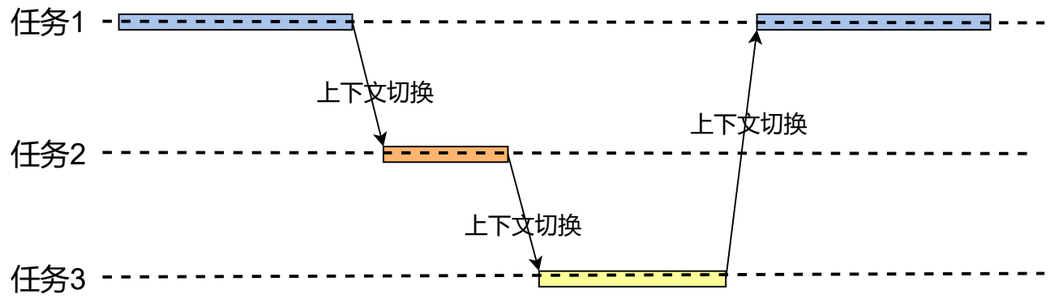
就绪态：进入调度器调度的任务状态。

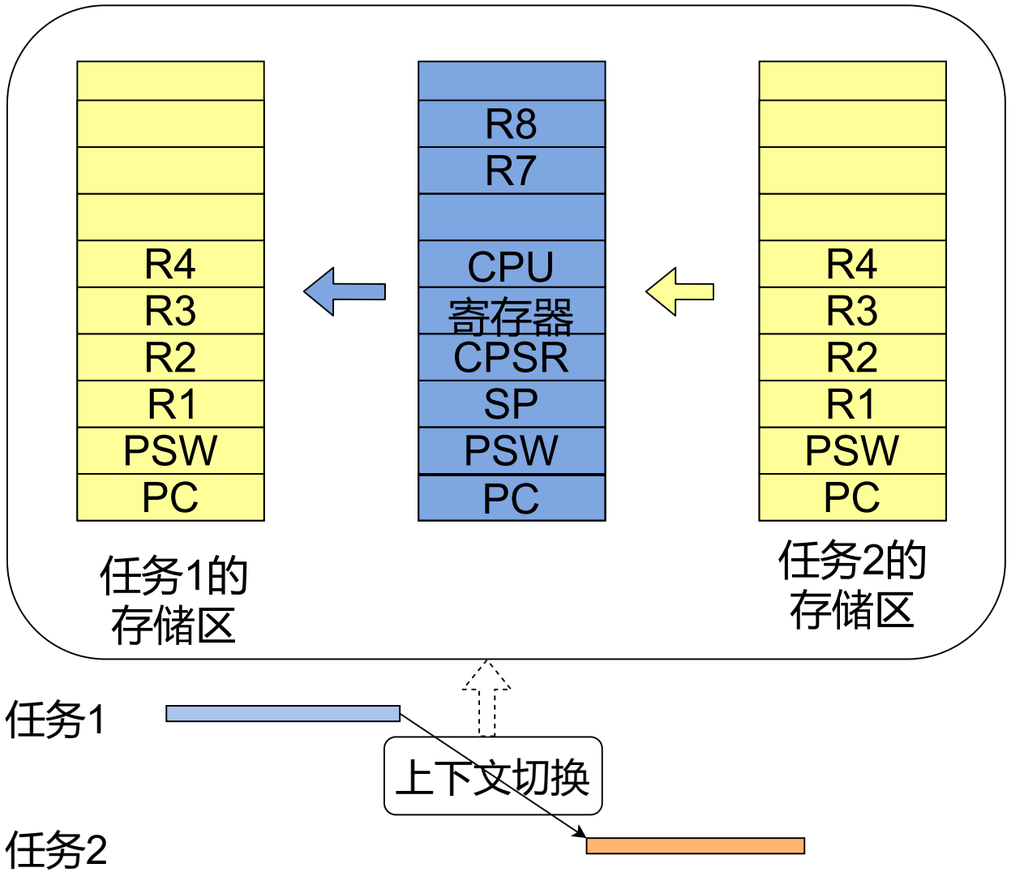
运行态：正在运行的任务状态。

中断服务态：由于任务被中断函数打断而进入的状态。

阻塞态：由于等待信号量、消息邮箱等而处于的状态，阻塞态与挂起态的区别在于阻塞态有超时机制。



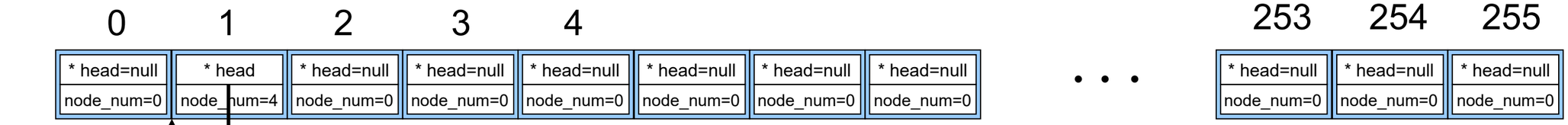


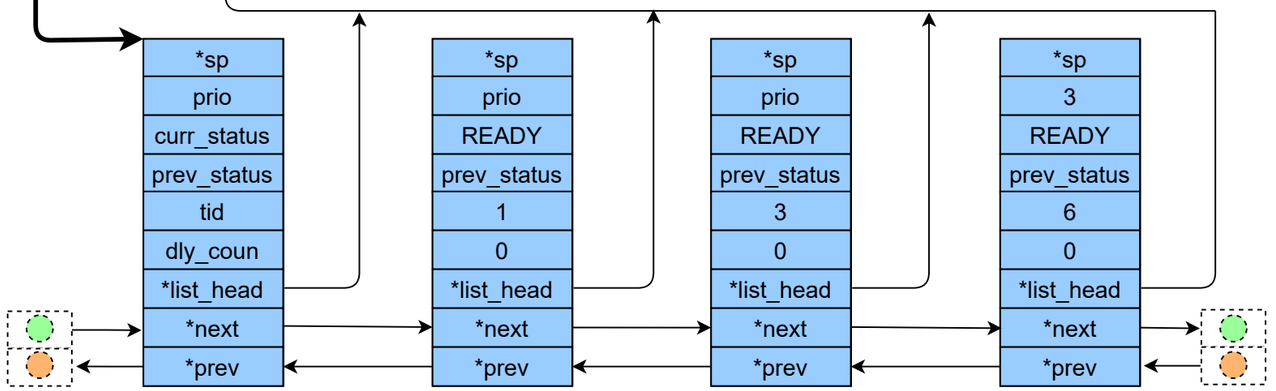


RT-Plain组件架构框图

RT-Plain任务管理相关的数据结构

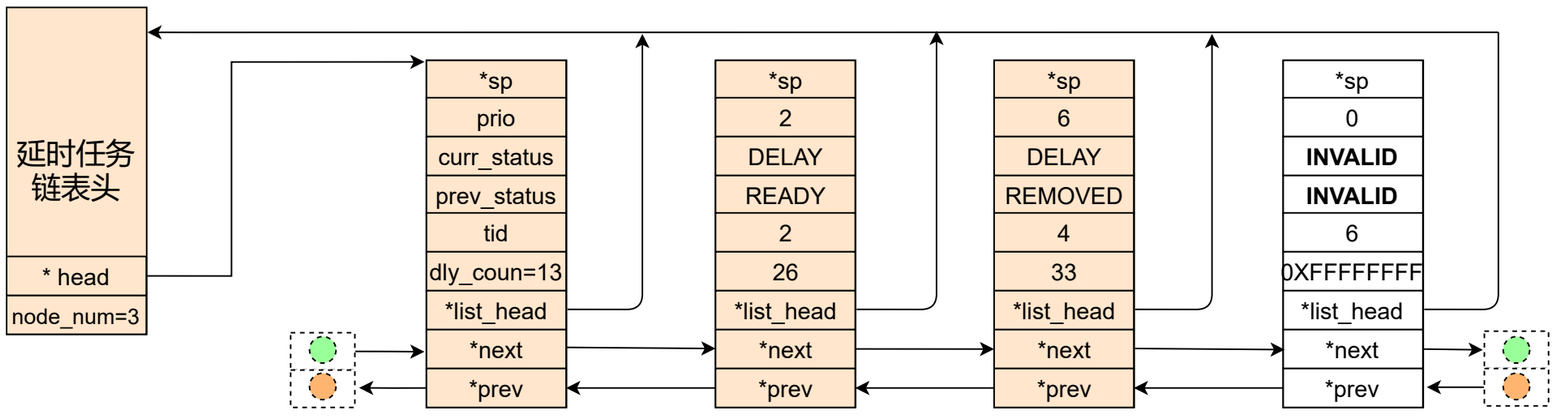
就绪表头数组





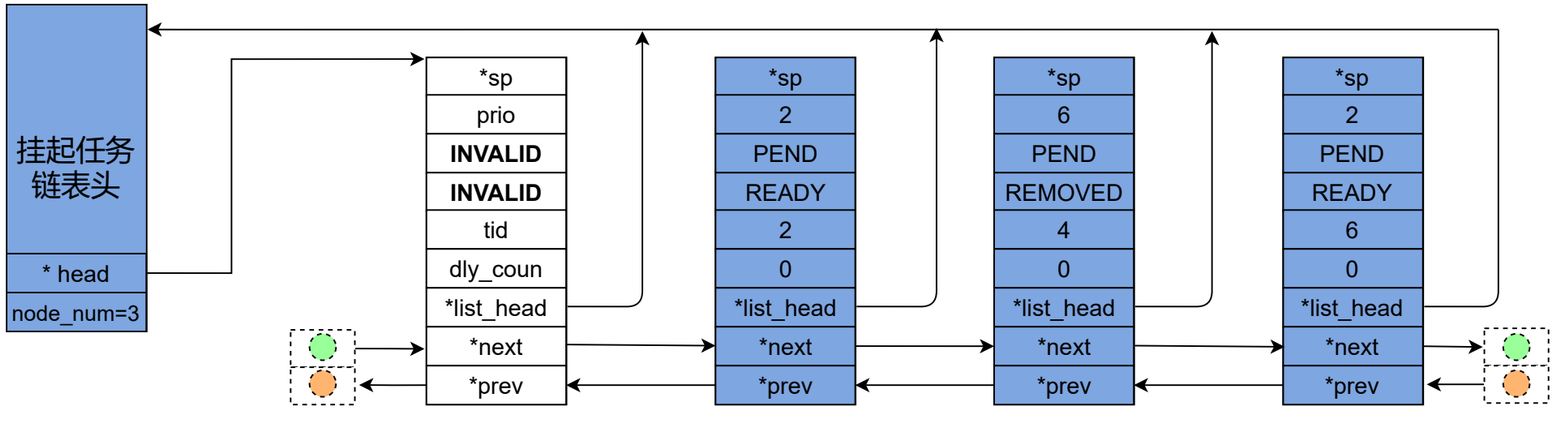
任务就绪链表（N条,N=最大优先级）

就绪链表采用普通的双向链表，使用一个表头来管理就绪链表。就绪链表要注意节点为0的特殊情况。



延时链表（1条）

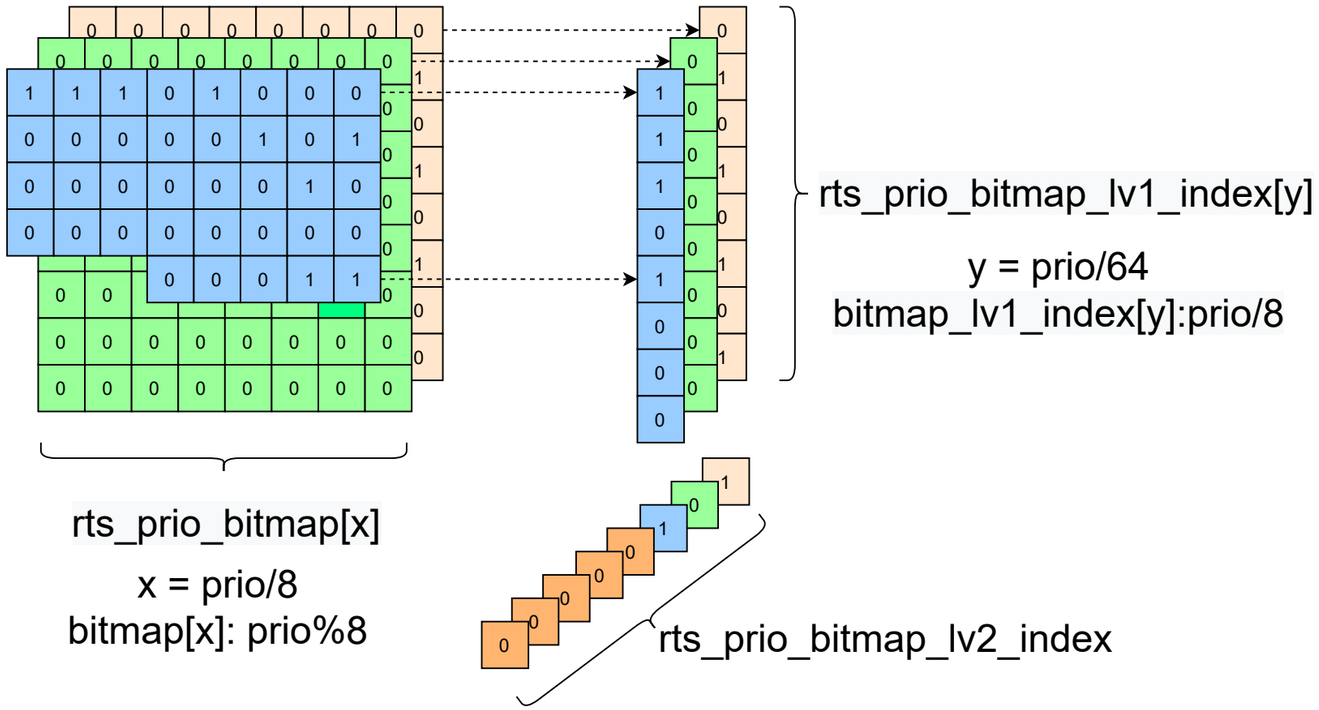
延时链表采用升序的双向链表，使用一个表头来管理就绪链表。当表头计时器与第一个节点时间相等时，无需遍历链表，只需要先将相等时间的节点移除。延时链表在OS中只有一条。



挂起链表（1条）

由挂起函数导致其进入挂起状态，挂起任务链表在OS中只有一条。

最高优先级的获取



最高优先级索引表
rts_prio_tbl[256]

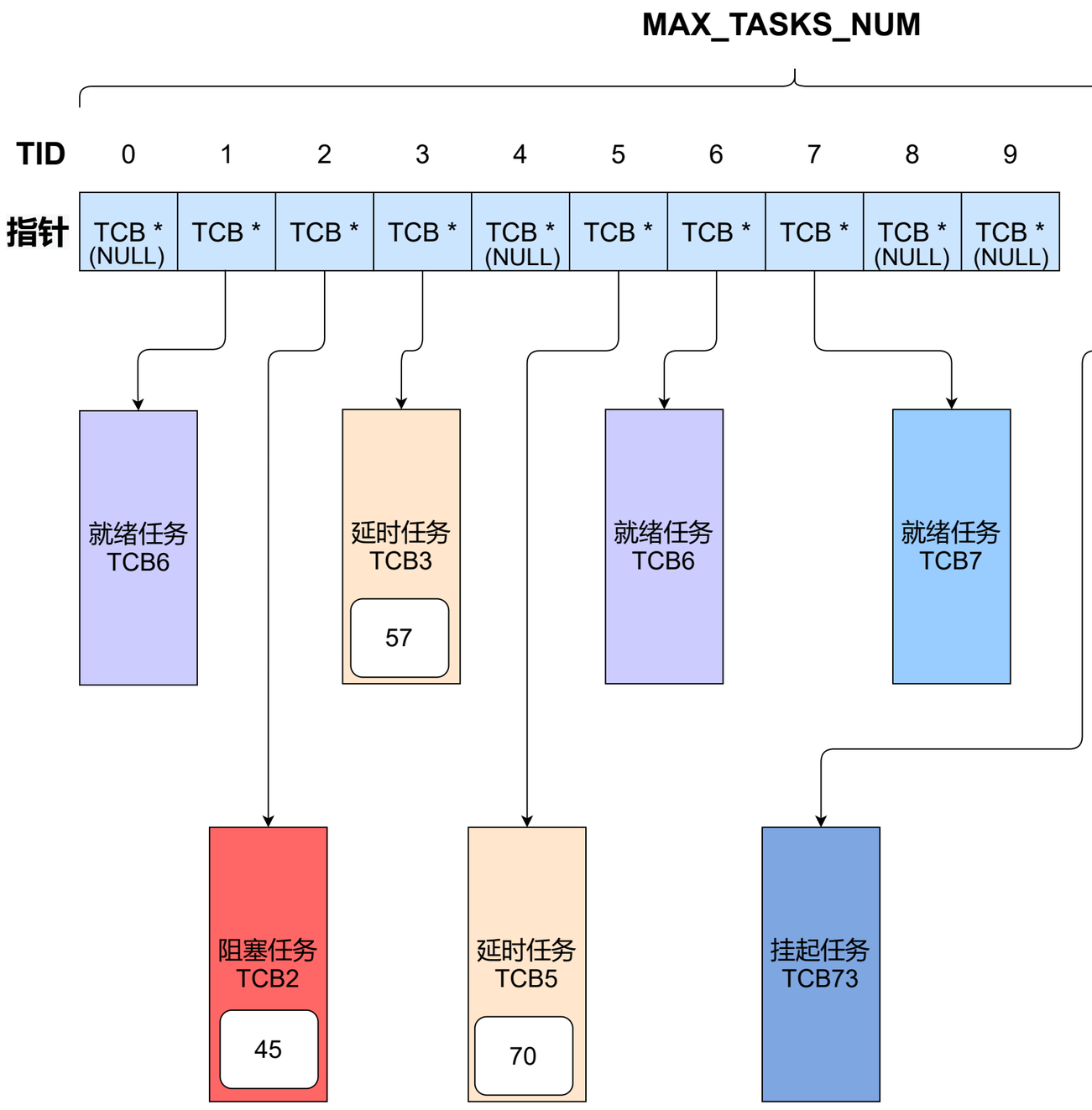
0	0	1	0	2											
													2	0	1	0

优先级的管理资源定义

```
#define  RTS_MAX_PRIORITIES    165

u8_t  rts_prio_bitmap[RTS_MAX_PRIORITIES/8+1] = {0}; //定义优先级位图

u8_t  rts_prio_bitmap_lv1_index[RTS_MAX_PRIORITIES/64+1]={0}; //定义优先级位图的一级索引

u8_t  rts_prio_bitmap_lv2_index = 0;//定义优先级位图的二级索引

u8_t  lv1_index       = rts_prio_tbl[rts_prio_bitmap_lv2_index];

u8_t bitmap_index = rts_prio_tbl[rts_prio_bitmap_lv1_index[lv1_index]];

u8_t bitmap_index_bit = rts_prio_bitmap[lv1_index *8 + bitmap_index];

u8_t   highest_prio = lv1_index*64 + bitmap_index*8 + bitmap_index_bit;
```

任务注册表

#define MAX_TASKS_NUM 75



任务注册表说明

当任务创建后，该任务的TCB指针将会被注册到该注册表数组中，这样做的目的是为了能在O(1)时间内找到任务ID的TCB以及其管理链表表头，因为在挂起任务函数中需要根据任务ID来找到任务的TCB以及管理该任务的链表，而TCB中包含了链表表头指针，所以找到任务的TCB就相当于找到了任务所在的链表。当使用移除函数移除任务时，则需要将该任务从管理链表中移除，同时在任务注册表中进行注销，也就是将其指针归零（NULL）。

温馨小贴士：这里不采用链表注册的原因主要是在根据任务ID号查找其TCB时需要遍历链表，其效率较低。但是其优点是创建的任务数量不受限制，而采用数组注册表方式，则在编译前需要指定最大任务数，所以其创建的任务数量有大小限制，但其效率较高，综合考虑，RT-Share选择数组来管理任务注册信息。

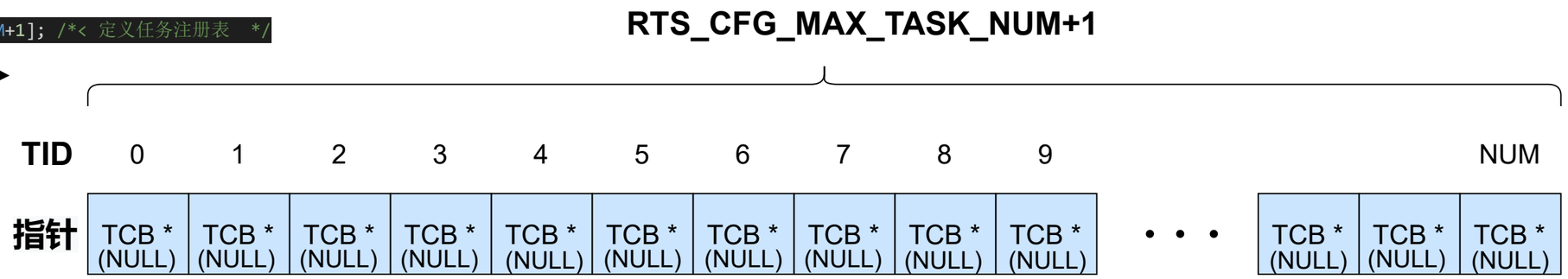
详细设计过程

RTS_OS初始化部分

注册表初始化

```
static TCB_t *rts_gb_tasks_reg_tbl[RTS_CFG_MAX_TASKS_NUM+1]; /*< 定义任务注册表  */
```

```
//RTS_OS任务注册表初始化
static void TasksRegTblInit(void)
{
    u8_t i;
    for(i=0;i<RTS_CFG_MAX_TASKS_NUM;i++)
    {
        rts_gb_tasks_reg_tbl[i]= NULL;
    }
}
```

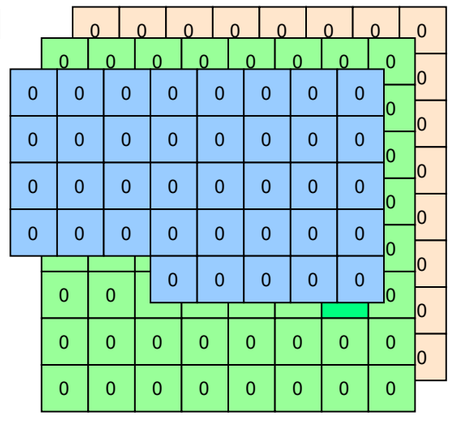


优先级位图初始化

```
#define MAX_BITMAP_ARRAY_NUM  (RTS_CFG_MAX_MAX_PRIORITIES/8+1)
```

```
static u8_t   rts_gb_prio_bitmap[MAX_BITMAP_ARRAY_NUM];
/*< 定义优先级位图 */
```

```
//RTS_OS优先级位图初始化
static void PrioBitMapInit(void)
{
    u8_t i;
    for(i=0;i<MAX_BITMAP_ARRAY_NUM;i++)
    {
        rts_gb_prio_bitmap[i] = 0;
    }
}
```



```
//+++++++获取最高优先级，通过查询方式获取最高优先级
static u8_t GetTaskHightPrio(void)
{
    u8_t i;
    for(i=0;i<MAX_BITMAP_ARRAY_NUM;i++)
    {
        if(rts_gb_prio_bitmap[i] != 0)
            return (rts_gb_prio_tbl[rts_gb_prio_bitmap[i]] + i*8);
    }
    return MAX_BITMAP_ARRAY_NUM;
}
```

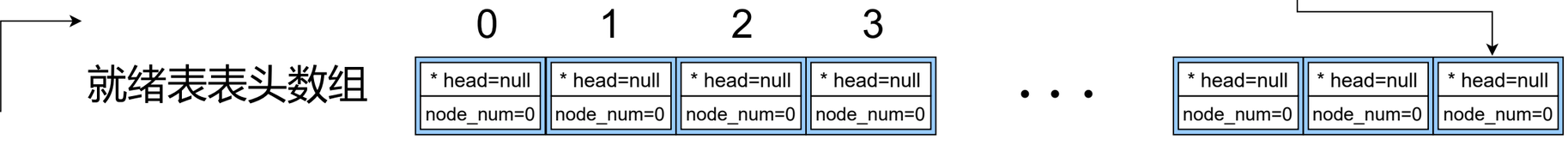
说明

V2.2版本RTS OS采用一级索引获取最高优先级，目前暂未实现两级索引。后续版本将会实现两级索引获取最高优先级。

就绪链表数组初始化

```
static taskListHead_t  rts_gb_rdy_lh_tbl[RTS_CFG_MAX_MAX_PRIORITIES+1]; /*< 定义就绪链表数组 */
```

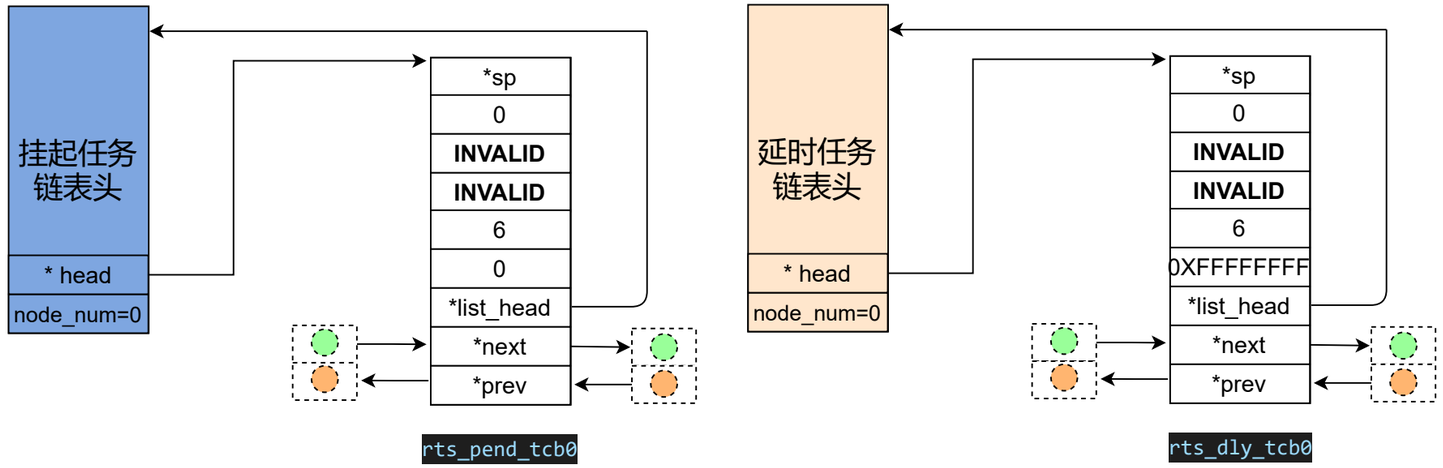
```
//RTS_OS就绪任务链表数组初始化
static void RdyListHeadTblInit(void)
{
    u8_t i;
    for(i=0;i<RTS_CFG_MAX_MAX_PRIORITIES;i++)
    {
        rts_gb_rdy_lh_tbl[i].head = NULL;
        rts_gb_rdy_lh_tbl[i].node_num = 0;
    }
}
```



挂起和延时链表初始化

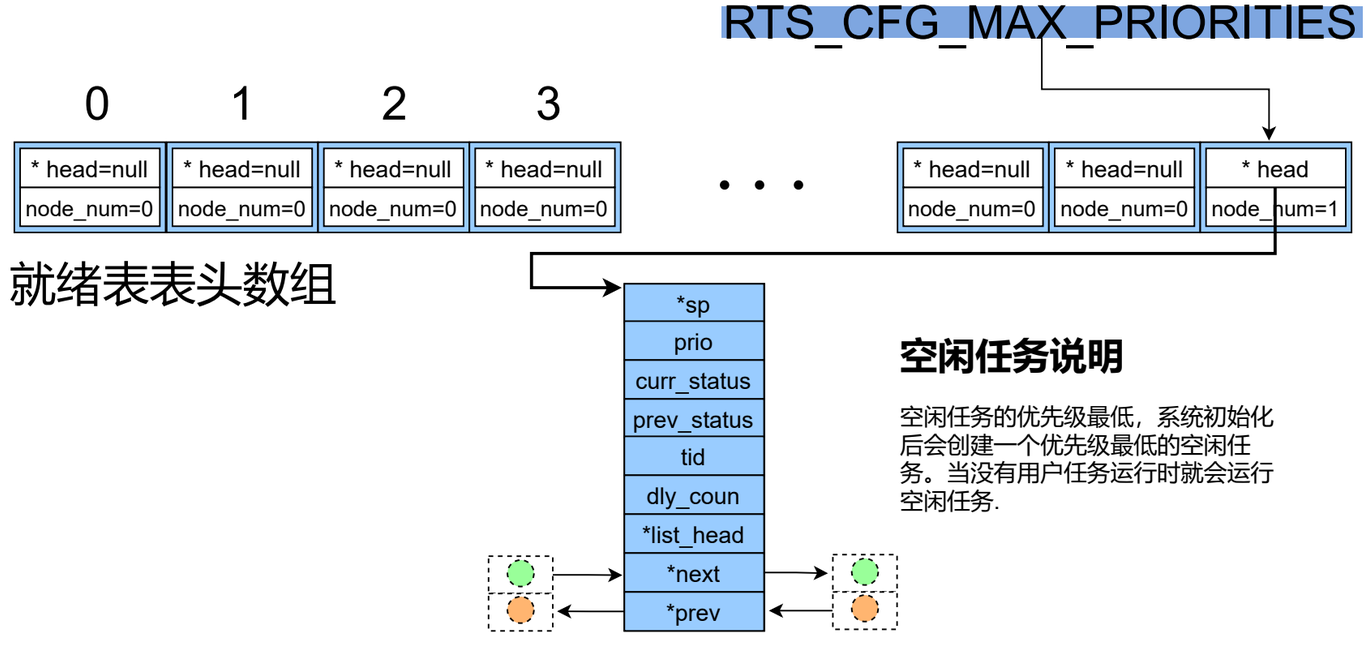
```
static taskListHead_t  rts_gb_pend_lh;    /*< 定义任务挂起链表头 */
static TCB_t           rts_pend_tcb0;     /*< 定义任务挂起TCB0  */

#if(RTS_CFG_DELAY_ENB > 0u)
static taskListHead_t  rts_gb_dly_lh;     /*< 定义任务延时链表头 */
static TCB_t           rts_dly_tcb0;      /*< 定义任务延时TCB0  */
#endif
```

```
//RTS_OS任务挂起、延时链表头初始化
static void Pend_DlyListHeadInit(void)
{
    rts_pend_tcb0.curr_status = RTS_TASK_STATUS_INVALID;
    rts_pend_tcb0.prev_status = RTS_TASK_STATUS_INVALID;
    rts_pend_tcb0.prev = &rts_pend_tcb0;
    rts_pend_tcb0.next = &rts_pend_tcb0;
    rts_pend_tcb0.dly_coun = 0;
    rts_pend_tcb0.list_head = &rts_gb_pend_lh;
    rts_gb_pend_lh.head = &rts_pend_tcb0;
    rts_gb_pend_lh.node_num = 0;

    #if(RTS_CFG_DELAY_ENB > 0u)
    rts_dly_tcb0.curr_status = RTS_TASK_STATUS_INVALID;
    rts_dly_tcb0.prev_status = RTS_TASK_STATUS_INVALID;
    rts_dly_tcb0.prev = &rts_dly_tcb0;
    rts_dly_tcb0.next = &rts_dly_tcb0;
    rts_dly_tcb0.dly_coun = (u32_t)0xffffffffU;
    rts_dly_tcb0.list_head = &rts_gb_dly_lh;
    rts_gb_dly_lh.head = &rts_dly_tcb0;
    rts_gb_dly_lh.node_num = 0;
    #endif
}
```



空闲任务初始化

```
//定义空闲任务
static void IdleTask(void *data)
{
    data = data;
    while(1)
    {
        #if(RTS_CFG_IDLE_HOOK_ENB > 0u)
        if(rts_gb_idle_task_pf != NULL)  //空闲任务的钩子函数指针
            rts_gb_idle_task_pf(data);
        #endif
    }
}

//空闲任务初始化
static u8_t IdleTaskInit(void)
{
    static u8_t idle_stack[128];
    return
RTS_CreateTask(IdleTask,idle_stack,128,RTS_CFG_MAX_MAX_PRIORITIES);
}
```



空闲任务说明

空闲任务的优先级最低，系统初始化后会创建一个优先级最低的空闲任务。当没有用户任务运行时就会运行空闲任务.

RTS_OS内核(操作)函数部分

获取最高优先级

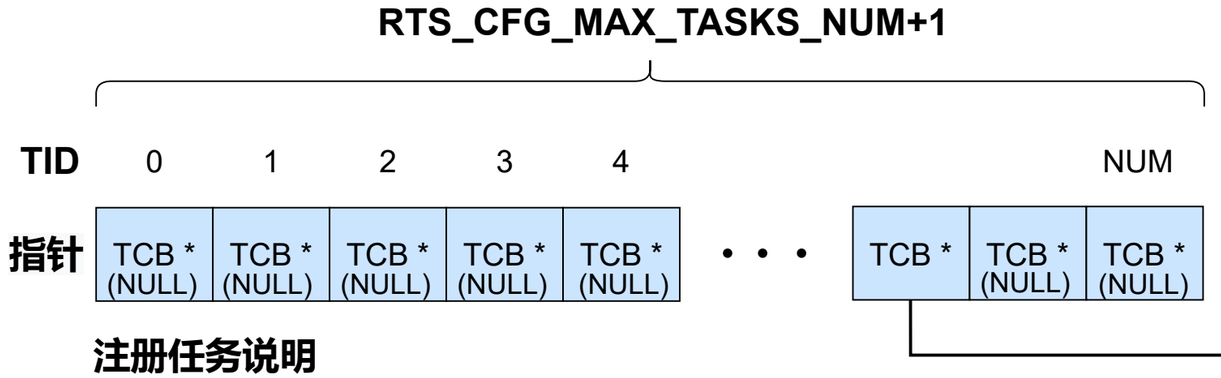
```
//++++++获取最高优先级，通过查询方式获取最高优先级
static u8_t GetTaskHightPrio(void)
{
    u8_t i;
    for(i=0;i<MAX_BITMAP_ARRAY_NUM;i++)
    {
        if(rts_gb_prio_bitmap[i] != 0)
            return (rts_gb_prio_tbl[rts_gb_prio_bitmap[i]] + i*8);
    }
    return MAX_BITMAP_ARRAY_NUM;
}
```

说明

V2.2版本RTS OS采用一级索引获取最高优先级，目前暂未实现两级索引。后续版本将会实现两级索引获取最高优先级。

注册任务TCB到任务注册表中

```
//++++++返回0表示注册失败，规定0号tid用户不能使用
static u8_t RegisterTask(TCB_t *tcb)
{
    u8_t tid;
    for(tid = 1;tid < RTS_CFG_MAX_TASKS_NUM+1;tid++)
    {
        //如果该位置没有被注册，则将TCB注册到该位置上
        if(rts_gb_tasks_reg_tbl[tid] == NULL)
        {
            rts_gb_tasks_reg_tbl[tid] = tcb;
            tcb->tid = tid;
            break;
        }
    }
    return tid;
}
```

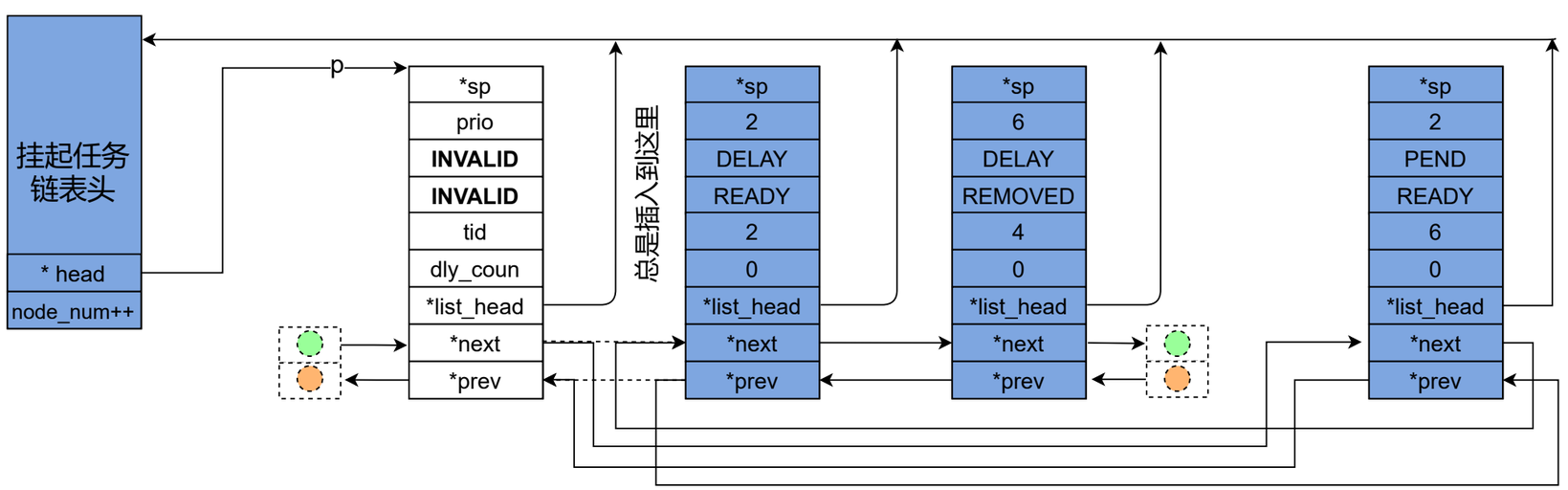


注册任务说明

正常的用户任务的的tid应该在[1,RTS_MAX_TASKS_NUM].
任务注册成功后返回正数，返回0表示注册失败。

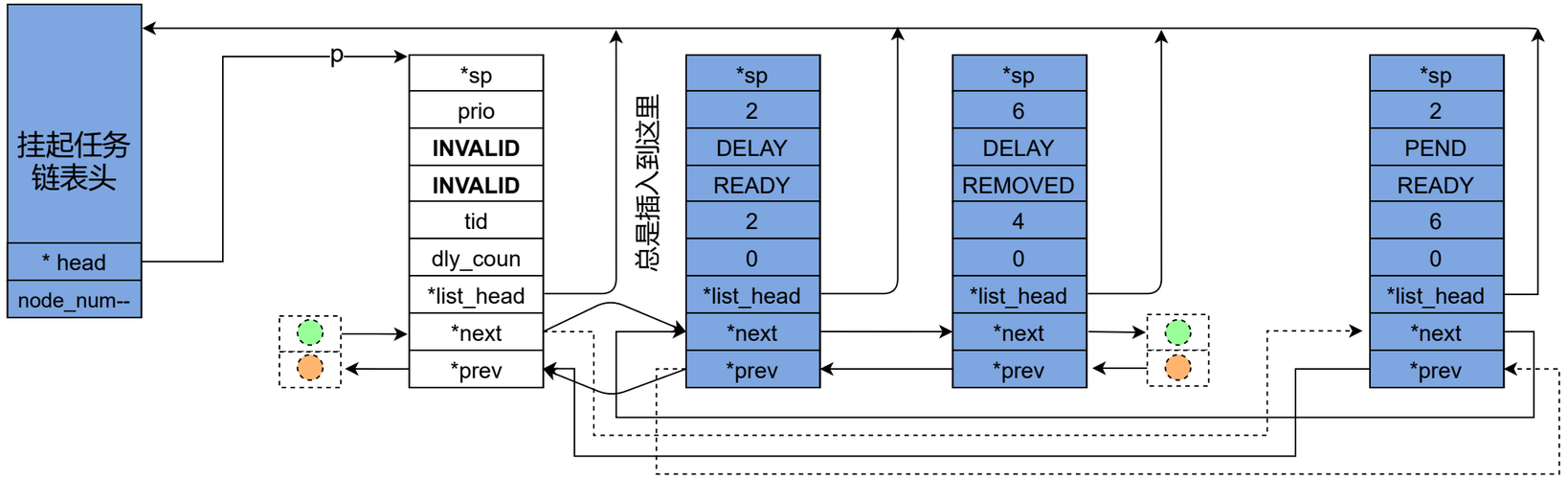
将任务TCB插入到挂起链表中

```
//++++++将任务TCB插入到挂起链表下
static void InsertTcbToPendList(taskListHead_t *list, TCB_t *tcb)
{
    TCB_t *p;
    p = list->head;

    tcb->prev = p;          //总是将TCB插入到挂起链表中无效填充链表的下一个位置
    tcb->list_head = list;
    tcb->next = p->next;
    p->next->prev = tcb;
    p->next = tcb;

    ++list->node_num;
}
```



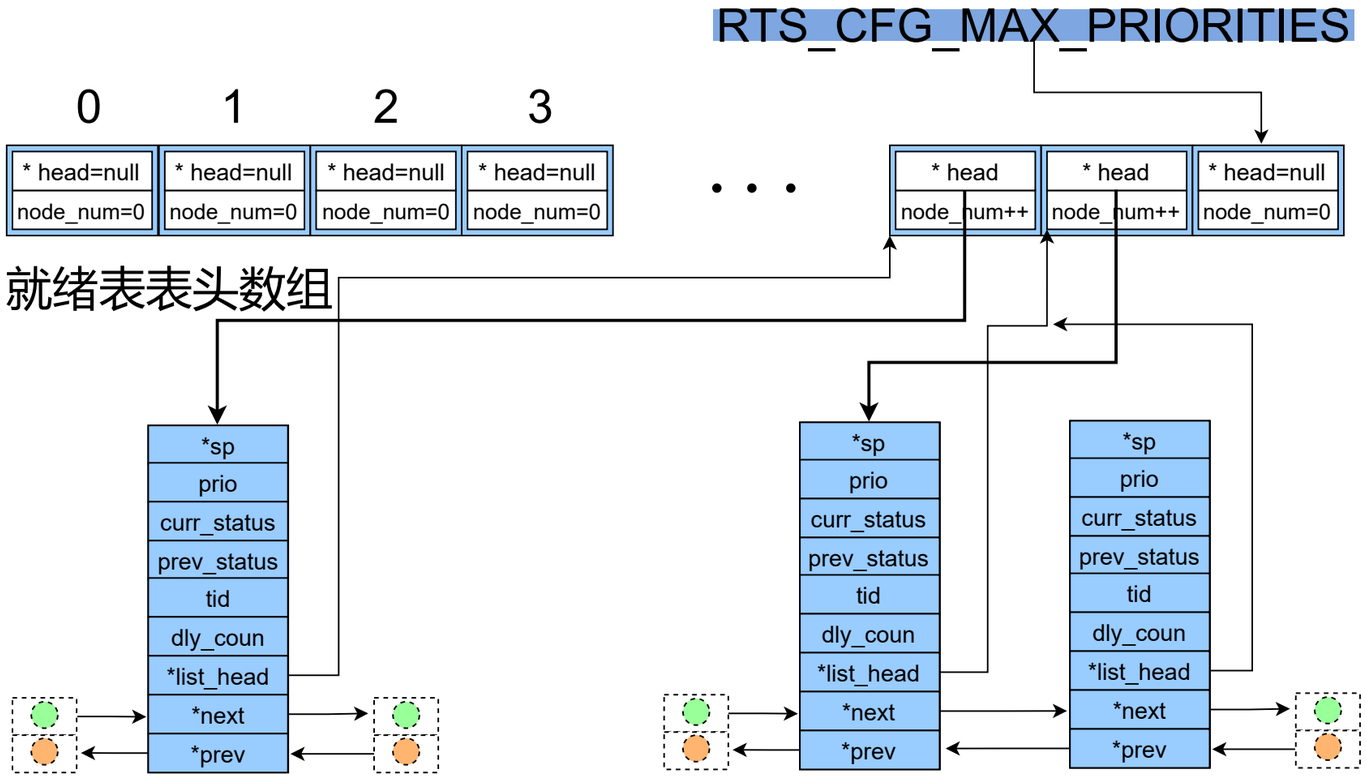
将任务TCB从挂起链表中移除

```
//+++++将任务TCB从挂起链表下移除
static void RemoveTcbFromPendList(taskListHead_t *list, TCB_t *tcb)
{
    tcb->prev->next = tcb->next;
    tcb->next->prev = tcb->prev;
    if(list->node_num > 0)
        --list->node_num;
}
```



将任务TCB插入到就绪链表中

```
//++++++将任务TCB插入到任务就绪链表中
static void InsertTcbToRdyList(TCB_t *tcb)
{
    u8_t prio,num;
    TCB_t *head = NULL;
    prio = tcb->prio;  //获取任务的优先级
    head = rts_gb_rdy_lh_tbl[prio].head;  //取得即将要插入就绪链表数组中的位置
    if(head == NULL)   //当就绪链表下没有就绪TCB时
    {
        rts_gb_rdy_lh_tbl[prio].head = tcb;
        tcb->next = tcb;
        tcb->prev = tcb;
    }
    else               //当就绪链表下存在就绪TCB时
    {
        tcb->next = head;
        tcb->prev = head->prev;
        head->prev->next = tcb;
        head->prev = tcb;
    }
    ++rts_gb_rdy_lh_tbl[prio].node_num;
    tcb->list_head = &rts_gb_rdy_lh_tbl[prio];
    rts_gb_prio_bitmap[prio/8] |= (u8_t)(1<<(prio%8)); //更新优先级位图
}
```

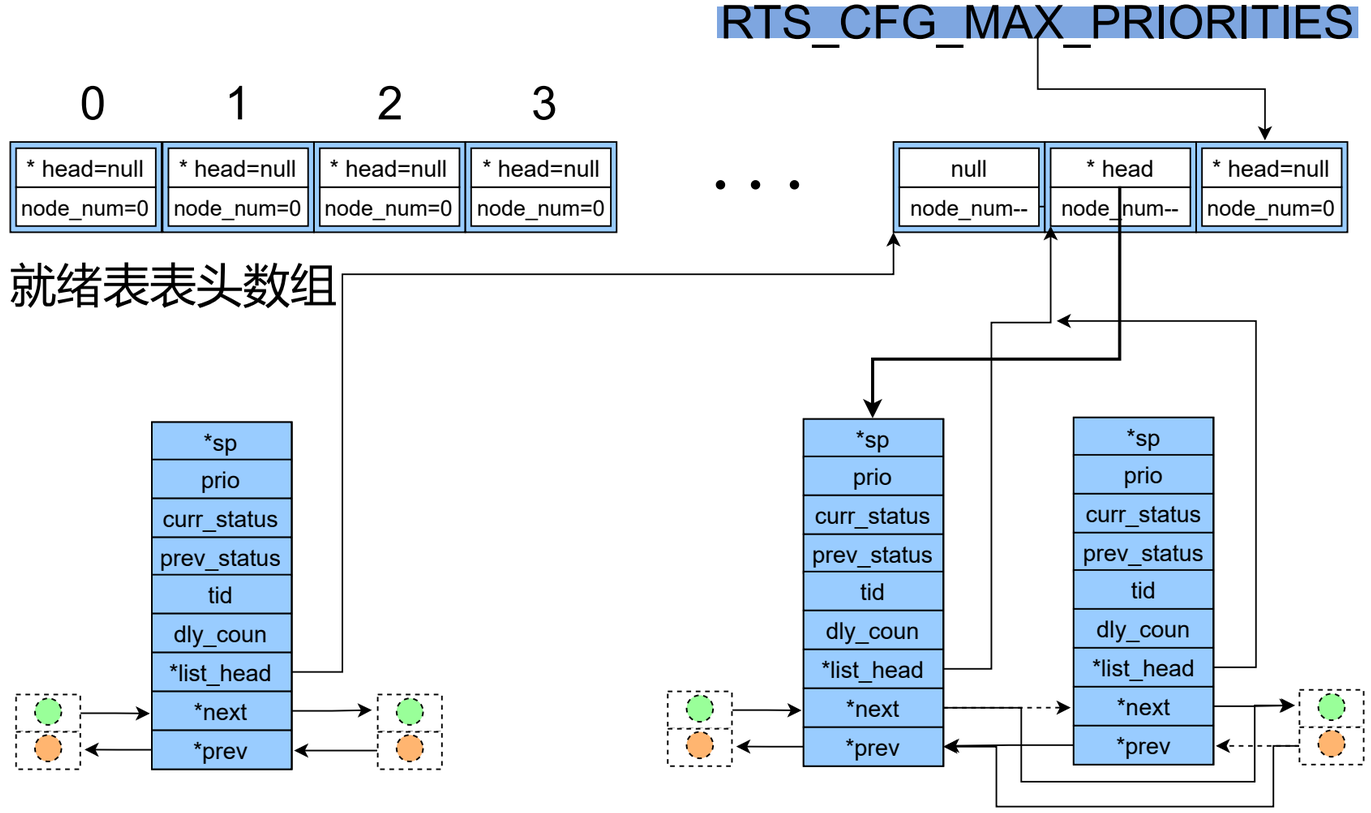


插入前就绪链表节点为0时

插入前就绪链表节点大于0时

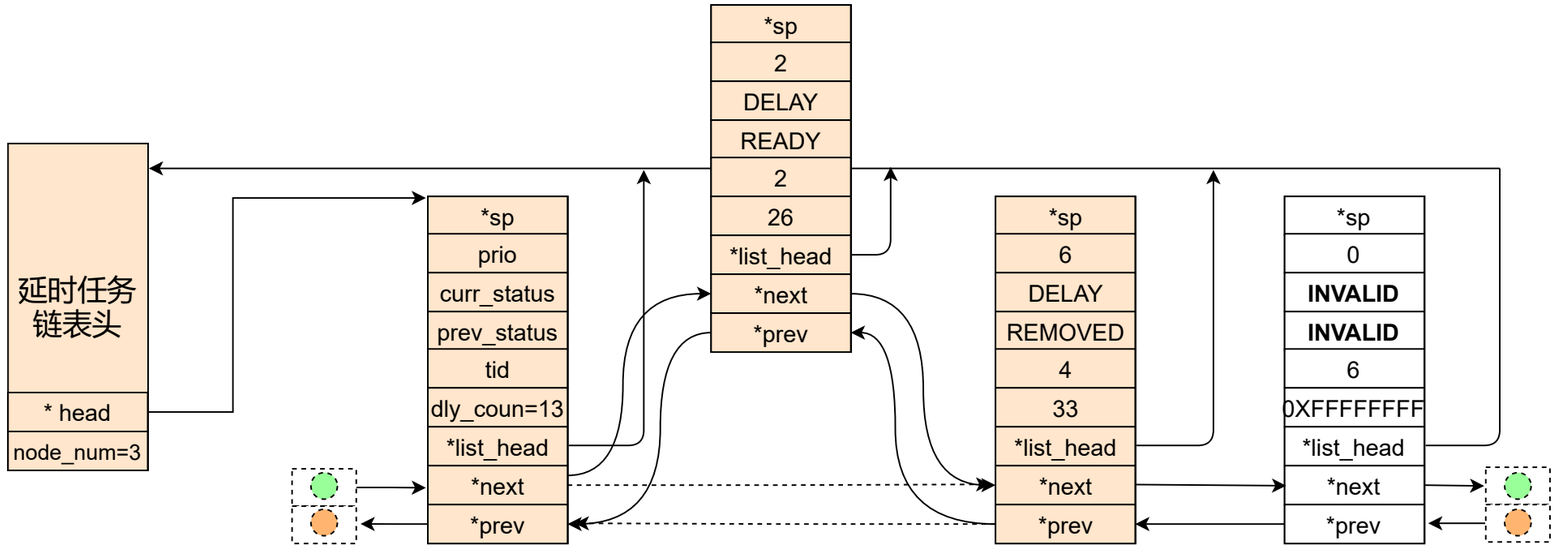
将任务TCB从就绪链表中移除

```
//++++++将任务TCB从就绪链表中删除
static void RemoveTcbFromRdyList(TCB_t *tcb)
{
    u8_t prio,num;
    prio = tcb->prio;
    num = rts_gb_rdy_lh_tbl[prio].node_num;
    if(num == 1)  //如果移除后就绪链表中的节点为0时
    {
        rts_gb_rdy_lh_tbl[prio].head = NULL;
        rts_gb_prio_bitmap[prio/8] &= (u8_t)(~(1<<(prio%8))); //更新优先级位图
        rts_gb_rdy_lh_tbl[prio].node_num = 0;
    }
    else if(num > 1)
    {
        rts_gb_rdy_lh_tbl[prio].head = tcb->next;
        tcb->prev->next = tcb->next;
        tcb->next->prev = tcb->prev;
        --rts_gb_rdy_lh_tbl[prio].node_num;
    }
}
```

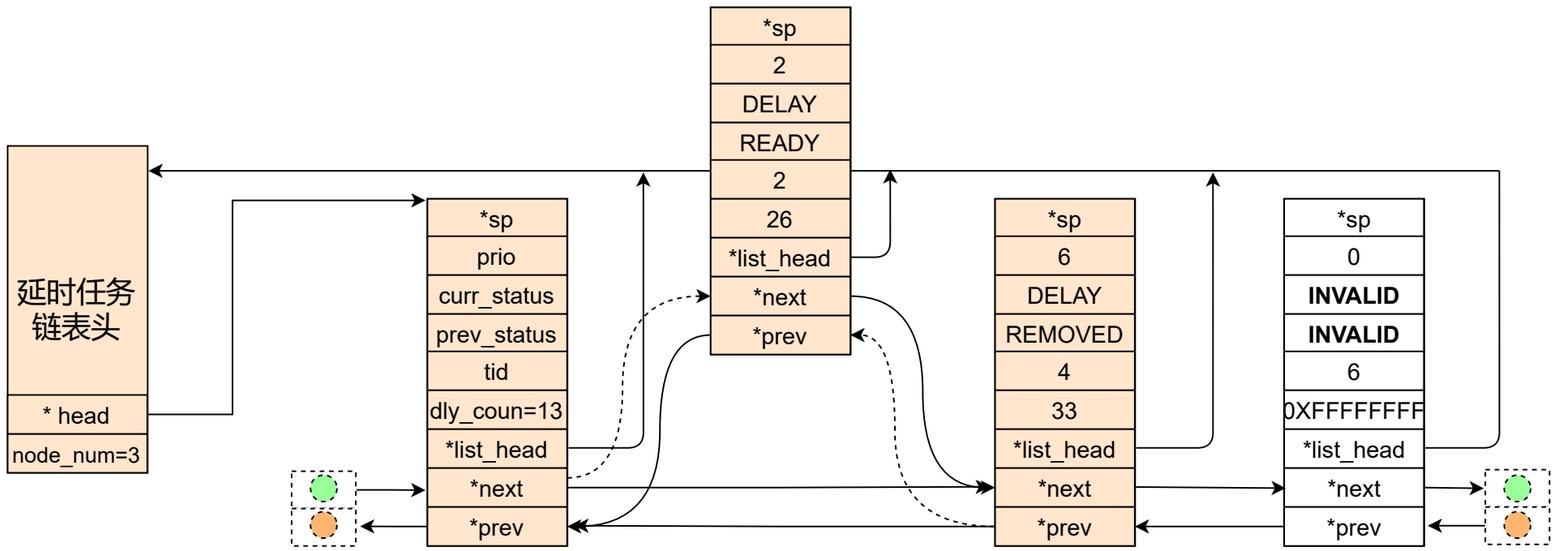


延时链表的插入和移除

```c
//+++++将任务TCB插入到延时链表中 (在后续版本中会加入错误处理)
#if(RTS_CFG_DELAY_ENB > 0u)
static void InsertTcbToDlyList(taskListHead_t *list, TCB_t *tcb)
{
    //这里需要按照从小到大的顺序插入到延时链表中
    u8_t i;
    TCB_t *p;
    u32_t end_ticks = tcb->dly_coun + rts_gb_systicks; //计算到达的时间点
    p = list->head;
    //此时滴答定时器溢出,此种情况的概率很小，大约1年才会发生一次
    if(end_ticks < rts_gb_systicks)
    {
        //遍历到链表末尾，堆链表元素进行处理
        while(p->curr_status != RTS_TASK_STATUS_INVALID)
        {
            p->dly_coun -= rts_gb_systicks;
            p = p->next;
        }
        end_ticks = tcb->dly_coun;
        rts_gb_systicks = 0;
    }
    p = list->head;  //开始重新插入
    //这里千万注意，如果后插进来的TCB块延时数等于当前数，则插入到该TCB后面
    while(end_ticks >= p->dly_coun)//找出要插入的位置
    {
       p = p->next;
    }
    tcb->next = p;  //开始将TCB插入到延时链表中
    tcb->prev = p->prev;
    p->prev->next = tcb;
    p->prev = tcb;
    //如果当前的计数最小，则让延时链表头指向该TCB
    if(end_ticks < list->head->dly_coun)
        list->head = tcb;
    //将延时时间到达的时刻写回TCB中
    tcb->dly_coun = end_ticks;
    tcb->list_head = &rts_gb_dly_lh;
}

//+++++将任务TCB从延时链表下移除
static void RemoveTcbFromDlyList(taskListHead_t *list, TCB_t *tcb)
{
    //这里为了加快速度，不对延时链表头进行处理，延时链表头的处理需要在调用该函数时进行
    tcb->prev->next = tcb->next;
    tcb->next->prev = tcb->prev;
}
#endif
```



延时链表元素的插入



延时链表元素的删除