RTT时间触发（TT）线程拓展开发手册

一、时间触发（TT）线程

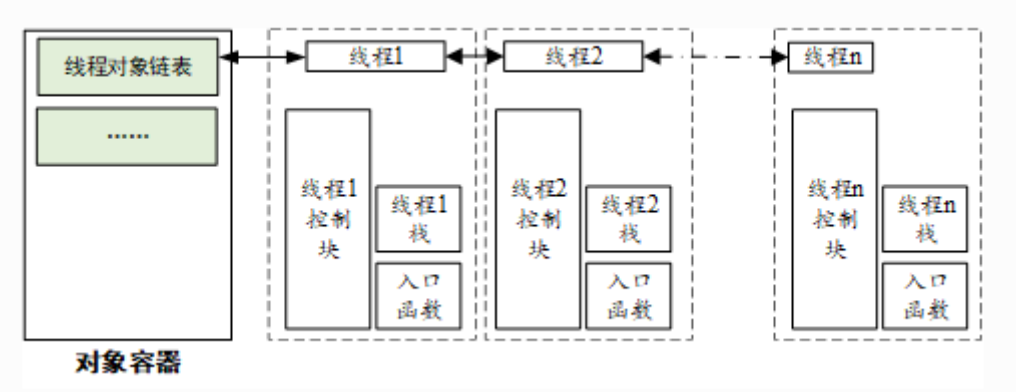
在工业应用场景下，常常需要保证某用户任务在指定时间点执行，而这一类任务通常是周期性执行的。

普通的线程或者定时器无法完成这一类工作，因为他们没有有效的实时性保证，可能会被其他线程打断。

TT（time trigger）线程通过监控系统时钟，以及对线程间切换逻辑的直接控制，保证了用户任务一定可以在指定时间运行。TT线程在创建时进行了时间冲突检测，保证不同时间触发线程之间在时间上是隔离的，进一步保证了实时性和确定性。

二、时间触发（TT）线程的工作机制

1.TT线程的管理



与事件触发（Event trigger，ET）线程类似，实时触发线程存储在线程对象链表中。但是链表中的TT线程对象是有序的，按照下一次触发时间升序排序。

与ET线程有若干优先级不同，TT线程由于在时间上是隔离的，因此只有一个优先级和一个线程链表。（实现时由于使用了跳表结构，会存在多个链表）

在向链表中插入新的TT线程对象时，使用跳表结构，以近似于$O(\log N)$的时间复杂度，找到保证链表升序排序的位置，将其插入。在删除链表中已有的对象时，直接连接该对象前后的链表节点。

2.TT线程的调度

TT线程的调度，以系统滴答（tick）为时间单位，通过监控时钟中断函数，在到达TT线程链表中第一个（下一次执行时间最早的）线程的执行时间时，调用线程切换函数。

在线程切换函数中，首先判断是否需要切换到TT线程，否则进入一般的线程切换流程，从而保证了TT线程的实时性。

在TT线程的入口函数，如果是周期性任务，一定要用循环结构实现，并且在循环体的最后调用rt_thread_yield()函数，让出CPU控制权。

Rt-thread的最小时间单位为tick（在默认情况下，1个tick为10ms），因此，TT线程调度的最小时间单位也为1个tick。如果需要增加调度的精度，可以修改系统参数，以减小tick代表的时间长度。

三、时间触发线程的实现

1.TT线程的创建

```
/**
 * This function will create a thread object and allocate thread object memory
 * and stack.
 *
 * @param name the name of thread, which shall ET unique
 * @param entry the entry function of thread
 * @param parameter the parameter of thread enter function
 * @param stack_size the size of thread stack
 * @param priority the priority of thread
 * @param tick the time slice if there are same priority thread
 * @param cycle TT线程执行周期
 * @param offset TT线程周期内偏移
 * @param maxi_exec_time TT线程声明的最长执行时间
 *
 * @return the created thread object
 */
rt_thread_t rt_TT_thread_create(const char *name,
                                void (*entry)(void *parameter),
                                void       *parameter,
                                rt_uint32_t stack_size,
                                rt_uint8_t  priority,
                                rt_uint32_t tick,
                                rt_uint32_t cycle,
                                rt_uint32_t offset,
                                rt_uint32_t maxi_exec_time)
{
    /* 执行冲突检测 */
    /* 如果返回true表示有冲突 */
    if(rt_TT_thread_time_collision_check(cycle, offset, maxi_exec_time))
    {
        rt_kprintf("create coallision : %s\n", name);
        return RT_NULL;
    }

    /* 直接把时间片信息设置为线程的最长运行时间
     * 在clock.c的rt_tick_increase里面，如果运行时间超过时间片
     * 就会被yield，这个TT线程会被挂起，并且在空闲时间再次被运行
     */
    rt_thread_t TT_thread = RT_NULL;
    TT_thread = rt_thread_create(name,
                                 entry, RT_NULL,
                                 stack_size,
                                 RT_THREAD_PRIORITY_MAX, maxi_exec_time);

    if(TT_thread == RT_NULL)
        return RT_NULL;

    TT_thread->rt_is_TT_Thread = 1;
    TT_thread->thread_exec_cycle = cycle;
    TT_thread->thread_exec_offset = offset;
    TT_thread->thread_maxi_exec_time = maxi_exec_time;
    /* 第一次执行的时间就是offset，以后每执行一次，开始时间+thread_exec_cycle */
```

```
    TT_thread->thread_start_time = get_TT_thread_start_time() + offset;

    rt_list_insert_ETfore(&rt_created_TT_thread_list,
                          &(TT_thread->time_collision_list));

    return TT_thread;
}
RTM_EXPORT(rt_TT_thread_create);
```

参数	名称
name	线程的名称；线程名称的最大长度由 rtconfig.h 中的宏 RT_NAME_MAX 指定，多余部分会被自动截掉
entry	线程入口函数
parameter	线程入口函数参数
stack_size	线程栈大小，单位是字节
priority	线程的优先级。在TT线程中无效。
tick	线程的时间片大小。在TT线程中无效。
cycle	TT线程执行周期
offset	TT线程周期内执行偏移
maxi_exec_time	TT线程声明的最长执行时间
返回	
thread	线程创建成功，返回线程句柄
RT_NULL	线程创建失败

2.系统滴答时钟监控

```
/**
 * This function will notify kernel there is one tick passed. Normally,
 * this function is invoked by clock ISR.
 */
void rt_tick_increase(void)
{
    .
    .
    .
    if ((get_first_TT_Thread_start_time() >= rt_get_global_time() &&
get_running_TT_Thread_count())
      || (thread->remaining_tick == 0 && thread->rt_is_TT_Thread))
    {
        //如果rt_tick到达指定时间，或者当前TT线程到达最长运行时间，则挂起当前线程并触发线程调
度
        rt_thread_yield();
    }
    .
    .
    .
```

```
}
```

如果当前系统时间等于（或者小于）最早的TT线程指定运行时间，则进行线程切换

3.线程切换逻辑控制

```
/**
 * This function will perform one schedule. It will select one thread
 * with the highest priority level, then switch to it.
 */
void rt_schedule(void)
{
    .
    .
    .
        /* 如果当前线程是TT线程，且时间片没有用完
         * 说明当前TT线程还没有执行完，不进行线程切换
         * 如果TT线程主动挂起或者终止运行，时间片一定会被置为0 */
        if(rt_current_thread->rt_is_TT_Thread)
        {
            if(rt_current_thread->remaining_tick)
            {
                rt_hw_interrupt_enable(level);
                return;
            }
            else
            {
                rt_current_thread->remaining_tick = rt_current_thread-
>init_tick;
            }
        }
        if(rt_running_TT_Thread_count && rt_first_TT_Thread_start_time <=
rt_get_global_time())
        {
          /* 如果有TT线程，且首个TT线程已经到了执行时间
           * 如果确定要执行TT线程，那么一定是取得
rt_TT_thread_list[RT_TT_THREAD_SKIP_LIST_LEVEL - 1]的第一个元素 */
            to_thread =
rt_list_entry(rt_TT_thread_list[RT_TT_THREAD_SKIP_LIST_LEVEL - 1].next,
                                    struct rt_thread,
                                    tlist);
        }
        else
        {
            //ET线程调度
        }
    .
    .
    .
        //线程切换
        rt_hw_context_switch_interrupt((rt_uint32_t)&from_thread->sp,
                                               (rt_uint32_t)&to_thread->sp);
    .
    .
    .
```

如果系统中有正在运行的TT线程，且当前系统时间到达了最早的TT线程的执行时间，则切换为TT线程

4.TT线程的挂起

```
rt_err_t rt_thread_yield(void)
{
    /* 如果是TT线程，那么应当按照开始时间升序排列，这里已经实现 */
    /* 由TT线程转普通线程 */
    if(thread->rt_is_TT_Thread)
    {
        /* TT线程的优先级一定是RT_THREAD_PRIORITY_MAX */
        RT_ASSERT(thread->current_priority == RT_THREAD_PRIORITY_MAX);

        if(thread->remaining_tick)
        {
            /* remove thread from thread list */
            rt_base_t i;
            for(i = 0; i < RT_TT_THREAD_SKIP_LIST_LEVEL; i++)
            {
                rt_list_remove(thread->thread_skip_list_nodes[i]);
            }
            thread->thread_start_time += thread->thread_exec_cycle;
            /* 将时间片信息置为0，用以避免ET线程创建时的切换影响TT线程
             * 执行时间会在rt_schedule()中被重置*/
            thread->remaining_tick = 0;

            rt_list_TT_thread_insert(thread);

            rt_hw_interrupt_enable(level);
            rt_schedule();

            return RT_EOK;
        }
        /* 如果剩余时间片为0，那么说明这个TT线程不是自己结束的
         * 而是因为超过最长执行时间，被强制结束的
         * 这个时候就不能算作一次成功的执行，直接退出
          */
        else
        {
            for (rt_base_t i = 0; i < RT_TT_THREAD_TIMEOUT_HOOK_LIST_SIZE; i++)
            {
              if (TT_thread_timeout_hook_list[i] != RT_NULL)
              {
                TT_thread_timeout_hook_list[i]();
              }
            }
            /* rt_thread_exit()里面包含了线程调度 */
            rt_hw_interrupt_enable(level);
            rt_thread_exit();
            return RT_EOK;
        }

    }
    /* 由普通线程转为TT线程，写这个条件是为了避免极端情况下普通线程独占最高优先级，不进行线程替换 */
```

```
    else if(get_first_TT_Thread_start_time() && get_first_TT_Thread_start_time()
<= rt_get_global_time())
    {
        /* remove thread from thread list */
        rt_list_remove(&(thread->tlist));

        /* put thread to end of ready queue */
        /* 如果是普通线程，由于避让TT线程导致挂起，那么加入链表最前面 */
        rt_list_insert_after(&(rt_thread_priority_table[thread-
>current_priority]),
                             &(thread->tlist));
        thread->stat = RT_THREAD_READY | (thread->stat & ~RT_THREAD_STAT_MASK);
        /* enable interrupt */
        rt_hw_interrupt_enable(level);
        rt_schedule();

        return RT_EOK;
    }
    else
    {
        //ET线程挂起流程
        .
        .
        .
    }
}
```

如果TT线程挂起时时间片为0，说明线程超时，这在实时任务中是致命的，因此直接退出。

如果时间片不为0，则在维护必要的信息后，将CPU交给ET线程

5.TT线程对象有序插入TT线程链表

```
/*
这个方法用来添加TT线程到链表
封装在rt_schedule_insert_TT_thread()里面使用时是首次添加TT线程
单独使用时是在一个周期的执行结束之后，再次添加到链表当中
*/
void rt_list_TT_thread_insert(struct rt_thread *TT_thread){
    /* 把TT线程加入rt_thread_priority_table的指定链表中，使用跳表结构 */
    rt_list_t *TT_thread_list = rt_TT_thread_list;
    rt_list_t *row_head[RT_TT_THREAD_SKIP_LIST_LEVEL];
    static rt_uint32_t random_nr;
    rt_uint32_t tst_nr;
    row_head[0]  = TT_thread_list[0].next;
    rt_uint32_t row_lvl;
    for (row_lvl = 0; row_lvl < RT_TT_THREAD_SKIP_LIST_LEVEL; row_lvl++)
    {
        //找到当前层中，目标节点的插入位置
        for (; row_head[row_lvl] != &TT_thread_list[row_lvl];
             row_head[row_lvl]  = row_head[row_lvl]->next)
        {
            struct rt_thread *cur_TT_thread;
#if RT_TT_THREAD_SKIP_LIST_LEVEL > 1
            if(row_lvl < RT_TT_THREAD_SKIP_LIST_LEVEL - 1)
            {
```

```
                cur_TT_thread = rt_list_entry(row_head[row_lvl], struct
rt_thread, row[row_lvl]);
            }
            else
                cur_TT_thread = rt_list_entry(row_head[row_lvl], struct
rt_thread, tlist);
#else
            cur_TT_thread = rt_list_entry(row_head[row_lvl], struct rt_thread,
tlist);
#endif

            if (cur_TT_thread->thread_start_time > TT_thread->thread_start_time)
            {
                break;
            }
        }
        //移动到下一层的相应位置
        row_head[row_lvl] = row_head[row_lvl]->prev;
#if RT_TT_THREAD_SKIP_LIST_LEVEL > 1
        if (row_lvl < RT_TT_THREAD_SKIP_LIST_LEVEL - 1)
        {
            /* 这里可以+1的原因不是链表的特性，而是因为每一个链表节点原本都是数组的其中一个元
素
             * +1就可以到数组中的下一个元素，而下一个元素已经维护好了，一定在链表下一层的正确
位置
             */
            if(row_head[row_lvl] == &TT_thread_list[row_lvl])
            {
                row_head[row_lvl + 1] = TT_thread_list[row_lvl + 1].next;
            }
            else
            {
                struct rt_thread *cur_TT_thread;

                if(row_lvl < RT_TT_THREAD_SKIP_LIST_LEVEL - 2)
                {
                    row_head[row_lvl + 1] = row_head[row_lvl] + 1;
                }
                else
                {
                    cur_TT_thread = rt_list_entry(row_head[row_lvl], struct
rt_thread, row[row_lvl]);
                    row_head[row_lvl + 1] = cur_TT_thread-
>thread_skip_list_nodes[row_lvl + 1];
                }
            }
        }
#endif
    }
    random_nr++;
    tst_nr = random_nr;
    //将目标节点加入最底层的指定位置
    rt_list_insert_after(row_head[RT_TT_THREAD_SKIP_LIST_LEVEL - 1],
                         TT_thread-
>thread_skip_list_nodes[RT_TT_THREAD_SKIP_LIST_LEVEL - 1]);
    for (row_lvl = 2; row_lvl <= RT_TT_THREAD_SKIP_LIST_LEVEL; row_lvl++)
    {
```

```
        /* 加到链表里面的，是一个链表节点数组的其中一个，在这个节点上+1，就可以到数组下一个元素，也就是下一层的链表节点
         * 从倒数第二层开始往上，如果当前节点需要作为框架节点加入非底层的层，那么就将节点的对应层加到链表的指定层上
         * 这里的是否需要加入是随机的，找一个随机数，如果指定的二进制位为1，那么就加入，否则直接退出
         * 退出的时候，这个节点一定加入了最底层，有可能加入了上层，不可能跳过一层
         * 也就是说，一定是某一层往下的全部层都加入了该节点
        */
        if (!(tst_nr & RT_TIMER_SKIP_LIST_MASK))
            rt_list_insert_after(row_head[RT_TT_THREAD_SKIP_LIST_LEVEL - row_lvl],
                                 TT_thread->thread_skip_list_nodes[RT_TT_THREAD_SKIP_LIST_LEVEL - row_lvl]);
        else
            break;
        /* Shift over the bits we have tested. Works well with 1 bit and 2
         * bits. */
        tst_nr >>= (RT_TIMER_SKIP_LIST_MASK + 1) >> 1;
    }

    /* 而且每一次插入和删除的时候都要维护rt_first_TTThread_start_time变量，也就是接下来最早的TT线程的开始时间 */
    struct rt_thread *first_TT_thread;
    first_TT_thread = rt_list_entry(rt_TT_thread_list[RT_TT_THREAD_SKIP_LIST_LEVEL - 1].next,
                                    struct rt_thread,
                                    tlist);
    rt_first_TT_Thread_start_time = first_TT_thread->thread_start_time;
}
```

使用跳表结构实现了$O(\log N)$时间复杂度找到插入位置。

这里跳表的层数RT_TT_THREAD_SKIP_LIST_LEVEL是可以在rtconfig.h中配置的，默认层数为5

6.超时异常钩子函数

```
/**
 * @ingroup Hook
 * This function sets a hook function to TT thread timeout exception. When the TT thread time out,
 * this hook function should ET invoked.
 *
 * @param hook the specified hook function
 *
 * @return RT_EOK: set OK
 *         -RT_EFULL: hook list is full
 *
 * @note the hook function must ET simple and never ET blocked or suspend.
 */
rt_err_t rt_TT_thread_timeout_sethook(void (*hook)(void))
{
    rt_size_t i;
    rt_base_t level;
    rt_err_t ret = -RT_EFULL;

    /* disable interrupt */
```

```
    level = rt_hw_interrupt_disable();

    for (i = 0; i < RT_TT_THREAD_TIMEOUT_HOOK_LIST_SIZE; i++)
    {
        if (TT_thread_timeout_hook_list[i] == RT_NULL)
        {
            TT_thread_timeout_hook_list[i] = hook;
            ret = RT_EOK;
            break;
        }
    }
    /* enable interrupt */
    rt_hw_interrupt_enable(level);

    return ret;
}

/**
 * delete the TT thread timeout hook on hook list
 *
 * @param hook the specified hook function
 *
 * @return RT_EOK: delete OK
 *         -RT_ENOSYS: hook was not found
 */
rt_err_t rt_TT_thread_timeout_delhook(void (*hook)(void))
{
    rt_size_t i;
    rt_base_t level;
    rt_err_t ret = -RT_ENOSYS;

    /* disable interrupt */
    level = rt_hw_interrupt_disable();

    for (i = 0; i < RT_TT_THREAD_TIMEOUT_HOOK_LIST_SIZE; i++)
    {
        if (TT_thread_timeout_hook_list[i] == hook)
        {
            TT_thread_timeout_hook_list[i] = RT_NULL;
            ret = RT_EOK;
            break;
        }
    }
    /* enable interrupt */
    rt_hw_interrupt_enable(level);

    return ret;
}
```

如果用户需要在TT线程超时时，执行指定的行为，可以通过钩子函数实现。默认可以添加一个钩子函数，可以通过修改RT_TT_THREAD_TIMEOUT_HOOK_LIST_SIZE调整钩子函数上限。

四、时间触发线程的应用示例

```
#include <rtthread.h>
#include <board.h>
```

```c
#define THREAD_PRIORITY         25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_thread_t tid1 = RT_NULL;

rt_uint32_t TT_start_time;

static void thread_entry(void *parameter)
{
    while(1){
        rt_tick_t delta = rt_tick_get();
        rt_kprintf("TT_thread1 exec at: %d\n", delta - TT_start_time);
        //rt_kprintf("thread next exec at: %d\n",
get_first_TTThread_start_time());
        rt_thread_yield();
    }
}

int TT_thread_sample(void)
{
    set_TT_thread_start_time(rt_tick_get());
    TT_start_time = get_TT_thread_start_time();

    /* 创建线程，入口函数是thread_entry */
    tid1 = rt_TT_thread_create("TT_thread1",
                            thread_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE,
                            50, 37, 2);

    /* 检查并执行 */
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    return 0;
}

/*将函数投影到命令*/
MSH_CMD_EXPORT(TT_thread_sample, Time Trigger thread sample);
```

在命令行输入TT_thread_sample命令即可使用