title: 【TencentOS tiny】深度源码分析（1）——task author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:07:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

任务的基本概念

从系统的角度看，任务是竞争系统资源的最小运行单元。TencentOS tiny是一个支持多任务的操作系统，任务可以使用或等待CPU、使用内存空间等系统资源，并独立于其它任务运行，理论上任何数量的任务都可以共享同一个优先级，这样子处于就绪态的多个相同优先级任务将会以时间片切换的方式共享处理器。

> 不过**要注意**的是：在TencentOS tiny中，不能创建与空闲任务相同优先级的任务`K_TASK_PRIO_IDLE`，相同优先级下的任务需要允许使用时间片调度，打开`TOS_CFG_ROUND_ROBIN_EN`。

简而言之： TencentOS tiny的任务可认为是一系列独立任务的集合。每个任务在自己的环境中运行。在任何时刻，只有一个任务得到运行，由TencentOS tiny调度器决定运行哪个任务。从宏观看上去所有的任务都在同时在执行。

TencentOS中的任务是抢占式调度机制，高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务阻塞或结束后才能得到调度。同时TencentOS也支持时间片轮转调度方式。

系统默认可以支持10个优先级，`0～TOS_CFG_TASK_PRIO_MAX`，这个宏定义是可以修改的，优先级数值越大的任务优先级越低，`(TOS_CFG_TASK_PRIO_MAX - (k_prio_t)1u)`为最低优先级，分配给空闲任务使用。

```
#define K_TASK_PRIO_IDLE         (k_prio_t)(TOS_CFG_TASK_PRIO_MAX - (k_prio_t)1u)
#define K_TASK_PRIO_INVALID      (k_prio_t)(TOS_CFG_TASK_PRIO_MAX)
```

任务状态

TencentOS tiny任务状态有以下几种。

- 就绪态（K_TASK_STATE_READY）：该任务在就绪列表中，就绪的任务已经具备执行的能力，只等待调度器进行调度，新创建的任务会初始化为就绪态。
- 运行态（K_TASK_STATE_READY）：该状态表明任务正在执行，此时它占用处理器，**其实此时的任务还是处于就绪列表中的**，TencentOS调度器选择运行的永远是处于最高优先级的就绪态任务，当任务被运行的一刻，它的任务状态就变成了运行态。
- 睡眠态（K_TASK_STATE_SLEEP）：如果任务当前正在休眠让出CPU使用权，那么就可以说这个任务处于休眠状态，该任务不在就绪列表中，此时任务处于睡眠列表中（或者叫延时列表）。
- 等待态（K_TASK_STATE_PEND）：任务正在等待信号量、队列或者等待事件等状态。
- 挂起态（K_TASK_STATE_SUSPENDED）：任务被挂起，此时任务对调度器而言是不可见的。
- 退出态（K_TASK_STATE_DELETED）：该任务运行结束，并且被删除。
- 等待超时状态（K_TASK_STATE_PENDTIMEOUT）：任务正在等待信号量、队列或者等待事件发生超时的状态。
- 睡眠挂起态（K_TASK_STATE_SLEEP_SUSPENDED）：任务在睡眠中被挂起时的状态。

- 等待挂起态（K_TASK_STATE_PEND_SUSPENDED）：任务正在等待信号量、队列或者等待事件时被挂起的状态。
- 等待超时挂起态（K_TASK_STATE_PENDTIMEOUT_SUSPENDED）：任务正在等待信号量、队列或者等待事件发生超时，但此时任务已经被挂起的状态。

```
// ready to schedule
// a task's pend_list is in readyqueue
#define K_TASK_STATE_READY                (k_task_state_t)0x0000

// delayed, or pend for a timeout
// a task's tick_list is in k_tick_list
#define K_TASK_STATE_SLEEP                (k_task_state_t)0x0001

// pend for something
// a task's pend_list is in some pend object's list
#define K_TASK_STATE_PEND                 (k_task_state_t)0x0002

// suspended
#define K_TASK_STATE_SUSPENDED            (k_task_state_t)0x0004

// deleted
#define K_TASK_STATE_DELETED              (k_task_state_t)0x0008

// actually we don't really need those TASK_STATE below, if you understand the
task state deeply, the code can be much more elegant.

// we are pending, also we are waitting for a timeout(eg. tos_sem_pend with a
valid timeout, not TOS_TIME_FOREVER)
// both a task's tick_list and pend_list is not empty
#define K_TASK_STATE_PENDTIMEOUT                      (k_task_state_t)
(K_TASK_STATE_PEND | K_TASK_STATE_SLEEP)

// suspended when sleeping
#define K_TASK_STATE_SLEEP_SUSPENDED                  (k_task_state_t)
(K_TASK_STATE_SLEEP | K_TASK_STATE_SUSPENDED)

// suspened when pending
#define K_TASK_STATE_PEND_SUSPENDED                   (k_task_state_t)
(K_TASK_STATE_PEND | K_TASK_STATE_SUSPENDED)

// suspended when pendtimeout
#define K_TASK_STATE_PENDTIMEOUT_SUSPENDED            (k_task_state_t)
(K_TASK_STATE_PENDTIMEOUT | K_TASK_STATE_SUSPENDED)
```

TencentOS中维护任务的数据结构

就绪列表

TencentOS tiny维护一条就绪列表，用于挂载系统中的所有处于就绪态的任务，他是readyqueue_t 类型的列表，其成员变量如下：

```
readyqueue_t        k_rdyq;

typedef struct readyqueue_st {
    k_list_t    task_list_head[TOS_CFG_TASK_PRIO_MAX];
    uint32_t    prio_mask[K_PRIO_TBL_SIZE];
    k_prio_t    highest_prio;
} readyqueue_t;
```

task_list_head是列表类型k_list_t的数组，TencentOS tiny为每个优先级的任务都分配一个列表，系统支持最大优先级为TOS_CFG_TASK_PRIO_MAX prio_mask则是优先级掩码数组，它是一个类型为32位变量的数组，数组成员个数由TOS_CFG_TASK_PRIO_MAX决定：

```
#define K_PRIO_TBL_SIZE         ((TOS_CFG_TASK_PRIO_MAX + 31) / 32)
```

当TOS_CFG_TASK_PRIO_MAX不超过32时数组成员变量只有一个，就是32位的变量数值，那么该变量的每一位代表一个优先级。比如当TOS_CFG_TASK_PRIO_MAX为64时，prio_mask[0]变量的每一位（bit）代表0-31优先级，而prio_mask[1]变量的每一位代表32-63优先级。

highest_prio则是记录当前优先级列表的最高优先级，方便索引task_list_head。

延时列表

与系统时间相关的任务都会被挂载到这个列表中，可能是睡眠、有期限地等待信号量、事件、消息队列等情况~

```
k_list_t            k_tick_list;
```

任务控制块

在多任务系统中，任务的执行是由系统调度的。系统为了顺利的调度任务，为每个任务都额外定义了一个任务控制块，这个任务控制块就相当于任务的身份证，里面存有任务的所有信息，比如任务的栈指针，任务名称，任务的形参等。有了这个任务控制块之后，以后系统对任务的全部操作都可以通过这个任务控制块来实现。TencentOS 任务控制块如下：

```
typedef struct k_task_st {
    k_stack_t          *sp;                 /**< 任务栈指针,用于切换上下文*/

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    knl_obj_t           knl_obj;            /**< 只是为了验证，测试当前对象是否真的是一项任务。*/
#endif
```

```
    char               *name;               /**< 任务名称 */
    k_task_entry_t      entry;              /**< 任务主体 */
    void               *arg;                /**< 任务主体形参 */
    k_task_state_t      state;              /**< 任务状态 */
    k_prio_t            prio;               /**< 任务优先级 */

    k_stack_t          *stk_base;           /**< 任务栈基地址 */
    size_t              stk_size;           /**< 任务栈大小 */

    k_tick_t            tick_expires;       /**< 任务阻塞的时间 */

    k_list_t            tick_list;          /**< 延时列表 */
    k_list_t            pend_list;          /**< 就绪、等待列表 */

#if TOS_CFG_MUTEX_EN > 0u
    k_list_t            mutex_own_list;     /**< 任务拥有的互斥量 */
    k_prio_t            prio_pending;       /*< 用于记录持有互斥量的任务初始优先级，
在优先级继承中使用 */
#endif

    pend_obj_t         *pending_obj;       /**< 记录任务此时挂载到的列表 */
    pend_state_t        pend_state;         /**< 等待被唤醒的原因（状态） */

#if TOS_CFG_ROUND_ROBIN_EN > 0u
    k_timeslice_t       timeslice_reload;   /**< 时间片初始值（重装载值） */
    k_timeslice_t       timeslice;          /**< 剩余时间片 */
#endif

#if TOS_CFG_MSG_EN > 0u
    void               *msg_addr;           /**< 保存接收到的消息 */
    size_t              msg_size;                   /**< 保存接收到的消息大小
*/
#endif

#if TOS_CFG_EVENT_EN > 0u
    k_opt_t             opt_event_pend;     /**< 等待事件的的操作类型：
TOS_OPT_EVENT_PEND_ANY 、 TOS_OPT_EVENT_PEND_ALL */
    k_event_flag_t      flag_expect;        /**< 期待发生的事件 */
    k_event_flag_t     *flag_match;         /**< 等待到的事件 */
#endif
} k_task_t;
```

创建任务

在TencentOS tiny中，凡是使用__API__ 修饰的函数都是提供给用户使用的，而使用__KERNEL__修饰的代码则是给内核使用的。 TencentOS的创建任务函数有好几个参数：

参数	含义
task	任务控制块

参数	含义
name	任务名字
entry	任务主体
arg	任务形参
prio	优先级
stk_base	任务栈基地址
stk_size	任务栈大小
timeslice	时间片

参数详解（来源TencentOS tiny开发指南）：

- task

  这是一个k_task_t类型的指针，k_task_t是内核的任务结构体类型。注意：task指针，应该指向生命周期大于待创建任务体生命周期的k_task_t类型变量，如果该指针指向的变量生命周期比待创建的任务体生命周期短，譬如可能是一个生命周期极端的函数栈上变量，可能会出现任务体还在运行而k_task_t变量已被销毁，会导致系统调度出现不可预知问题。

- name

  指向任务名字符串的指针。注意：同task，该指针指向的字符串生命周期应该大于待创建的任务体生命周期，一般来说，传入字符串常量指针即可。

- entry

  任务体运行的函数入口。当任务创建完毕进入运行状态后，entry是任务执行的入口，用户可以在此函数中编写业务逻辑。

- arg

  传递给任务入口函数的参数。

- prio

  任务优先级。prio的数值越小，优先级越高。用户可以在tos_config.h中，通过TOS_CFG_TASK_PRIO_MAX来配置任务优先级的最大数值，在内核的实现中，idle任务的优先级会被分配为TOS_CFG_TASK_PRIO_MAX - 1,此优先级只能被idle任务使用。因此对于一个用户创建的任务来说，合理的优先级范围应该为[0, TOS_CFG_TASK_PRIO_MAX - 2]。另外TOS_CFG_TASK_PRIO_MAX的配置值必需大于等于8。

- stk_base

  任务在运行时使用的栈空间的起始地址。注意：同task，该指针指向的内存空间的生命周期应该大于待创建的任务体生命周期。stk_base是k_stack_t类型的数组起始地址。

- stk_size

任务的栈空间大小。注意：因为stk_base是k_stack_t类型的数组指针，因此实际栈空间所占内存大小为stk_size * sizeof(k_stack_t)。

- timeslice

  时间片轮转机制下当前任务的时间片大小。当timeslice为0时，任务调度时间片会被设置为默认大小（TOS_CFG_CPU_TICK_PER_SECOND / 10），系统时钟滴答（systick）数 / 10。

创建任务的实现如下：首先对参数进行检查，还要再提一下：**在TencentOS中，不能创建与空闲任务相同优先级的任务**K_TASK_PRIO_IDLE。然后调用cpu_task_stk_init函数将任务栈进行初始化，并且将传入的参数记录到任务控制块中。如果打开了TOS_CFG_ROUND_ROBIN_EN 宏定义，则表示支持时间片调度，则需要配置时间片相关的信息timeslice 到任务控制块中。然后调用task_state_set_ready函数将新创建的任务设置为就绪态K_TASK_STATE_READY，再调用readyqueue_add_tail函数将任务插入就绪列表k_rdyq中。如果调度器运行起来了，则进行一次任务调度。

> 个人感觉吧，没有从堆中动态分配还是有点小小的遗憾，我更喜欢简单的函数接口~！

代码如下：

```
__API__ k_err_t tos_task_create(k_task_t *task,
                                char *name,
                                k_task_entry_t entry,
                                void *arg,
                                k_prio_t prio,
                                k_stack_t *stk_base,
                                size_t stk_size,
                                k_timeslice_t timeslice)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_IN_IRQ_CHECK();

    TOS_PTR_SANITY_CHECK(task);
    TOS_PTR_SANITY_CHECK(entry);
    TOS_PTR_SANITY_CHECK(stk_base);

    if (unlikely(stk_size < sizeof(cpu_context_t))) {
        return K_ERR_TASK_STK_SIZE_INVALID;
    }

    if (unlikely(prio == K_TASK_PRIO_IDLE && !knl_is_idle(task))) {
        return K_ERR_TASK_PRIO_INVALID;
    }

    if (unlikely(prio > K_TASK_PRIO_IDLE)) {
        return K_ERR_TASK_PRIO_INVALID;
    }

    task_reset(task);
#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    knl_object_init(&task->knl_obj, KNL_OBJ_TYPE_TASK);
#endif
```

```
    task->sp        = cpu_task_stk_init((void *)entry, arg, (void *)task_exit,
stk_base, stk_size);
    task->entry     = entry;
    task->arg       = arg;
    task->name      = name;
    task->prio      = prio;
    task->stk_base  = stk_base;
    task->stk_size  = stk_size;

#if TOS_CFG_ROUND_ROBIN_EN > 0u
    task->timeslice_reload = timeslice;

    if (timeslice == (k_timeslice_t)0u) {
        task->timeslice = k_robin_default_timeslice;
    } else {
        task->timeslice = timeslice;
    }
#endif

    TOS_CPU_INT_DISABLE();
    task_state_set_ready(task);
    readyqueue_add_tail(task);
    TOS_CPU_INT_ENABLE();

    if (tos_knl_is_running()) {
        knl_sched();
    }

    return K_ERR_NONE;
}
```

任务销毁

这个函数十分简单，根据传递进来的任务控制块销毁任务，也可以传递进NULL表示销毁当前运行的任务。但是不允许销毁空闲任务k_idle_task，当调度器被锁住时不能销毁自身，会返回K_ERR_SCHED_LOCKED错误代码。如果使用了互斥量，当任务被销毁时会释放掉互斥量，并且根据任务所处的状态进行销毁，比如任务处于就绪态、延时态、等待态，则会从对应的状态列表中移除。 代码实现如下：

```
__API__ k_err_t tos_task_destroy(k_task_t *task)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_IN_IRQ_CHECK();

    if (unlikely(!task)) {
        task = k_curr_task;
    }

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
```

```
    if (!knl_object_verify(&task->knl_obj, KNL_OBJ_TYPE_TASK)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    if (knl_is_idle(task)) {
        return K_ERR_TASK_DESTROY_IDLE;
    }

    if (knl_is_self(task) && knl_is_sched_locked()) {
        return K_ERR_SCHED_LOCKED;
    }

    TOS_CPU_INT_DISABLE();

#if TOS_CFG_MUTEX_EN > 0u
    // when we die, wakeup all the people in this land.
    if (!tos_list_empty(&task->mutex_own_list)) {
        task_mutex_release(task);
    }
#endif

    if (task_state_is_ready(task)) { // that's simple, good kid
        readyqueue_remove(task);
    }
    if (task_state_is_sleeping(task)) {
        tick_list_remove(task);
    }
    if (task_state_is_pending(task)) {
        pend_list_remove(task);
    }

    task_reset(task);
    task_state_set_deleted(task);

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

任务睡眠

任务睡眠非常简单，主要的思路就是将任务从就绪列表移除，然后添加到延时列表中k_tick_list，如果调度器被锁，直接返回错误代码K_ERR_SCHED_LOCKED，如果睡眠时间为0，则调用tos_task_yield函数发起一次任务调度；调用tick_list_add函数将任务插入延时列表中，睡眠的时间delay是由用户指定的。不过**需要注意的是如果任务睡眠的时间是永久睡眠TOS_TIME_FOREVER**，将返回错误代码K_ERR_DELAY_FOREVER，这是因为任务睡眠是主动行为，如果永久睡眠了，将没法主动唤醒，而任务等待事件、信号量、消息队列等行为是被动行为，可以是永久等待，一旦事件发生了、信号量呗释放、消息队列不为空时任务就会被唤醒，这是被动

行为，这两点需要区分开来。最后调用readyqueue_remove函数将任务从就绪列表中移除，然后调用knl_sched函数发起一次任务调度，就能切换另一个任务。 任务睡眠的代码如下：

```
__API__ k_err_t tos_task_delay(k_tick_t delay)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_IN_IRQ_CHECK();

    if (knl_is_sched_locked()) {
        return K_ERR_SCHED_LOCKED;
    }

    if (unlikely(delay == (k_tick_t)0u)) {
        tos_task_yield();
        return K_ERR_NONE;
    }

    TOS_CPU_INT_DISABLE();

    if (tick_list_add(k_curr_task, delay) != K_ERR_NONE) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_DELAY_FOREVER;
    }

    readyqueue_remove(k_curr_task);

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

喜欢就关注我吧！

相关代码可以在公众号后台获取。