第 2 章

RT-Thread 实时线程管理及调度

一个典型的简单软件系统会被设计成串行地运行：按照准确的指令步骤一次一个指令的运行。但是这种方法对于实时应用是不可行的，因为它们通常需要在固定的时间内处理多个输入输出，实时软件应用程序必须设计成一个并行的系统。

并行设计需要开发人员把一个应用分解成一个个小的，可调度的，序列化的程序单元。当正确的这样做时，并行设计能够让系统满足实时系统的性能及时间的要求。

2.1 实时系统的需求

实时系统中主要就是指固定的时间内正确的对外部事件做出响应。这个“时间内”，系统内部会做一些处理，例如获取数据后进行分析计算，加工处理等。而在这段时间之外，系统可能会闲下来，做一些空余的事。

例如一个手机终端，当一个电话拨入的时候，系统应当在固定的时间内发出振铃或声音提示以通知主人有来电，询问是否进行接听。而在非电话拨入的时候，人们可以用它进行一些其它工作，例如听听音乐，玩玩游戏。

从上面的例子我们可以看出，实时系统是一种倾向性的系统，对于实时的事件需要在第一时间做出回应，而对非实时任务则可以在实时事件到达时为之让路 – 被抢占。所以实时系统也可以看成是一个分级的系统，不同重要性的任务具有不同的优先级。

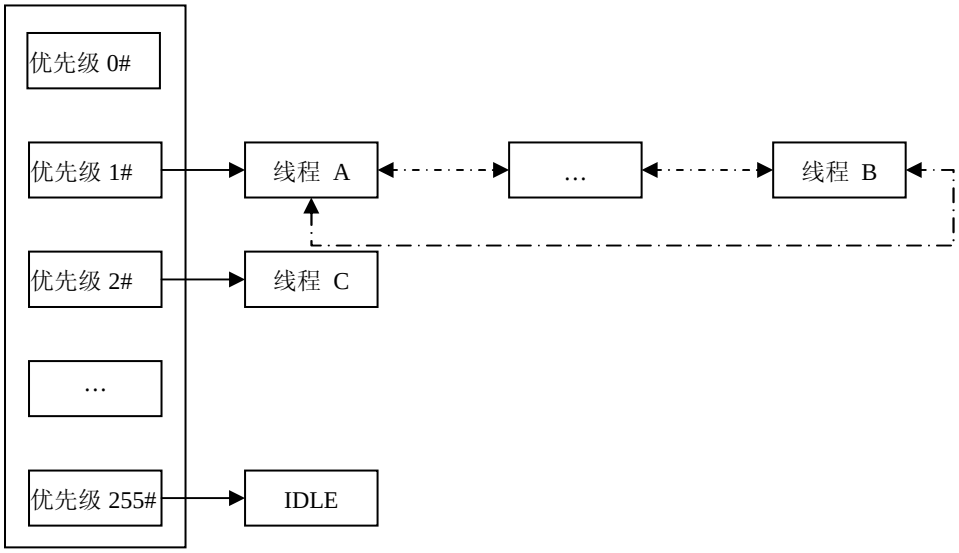
2.2 线程调度器

RT-Thread 中提供的线程调度器是基于全抢占式优先级的调度，在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。系统总共支持 256 个优先级(0 ～ 255，255 分配给空闲线程使用，一般不使用。在一些资源比较紧张的系统中，可以根据情况选择只支持 32 个优先级)。在系统中，当有比当前线程优先级还要高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理机进行运行。

如图 2 - 1 线程优先级队列

所示，RT-Thread 调度器实现中包含一组，总共 256 个优先级队列组，每个优先级队列采用双向环形链表的方式链接，255 优先级队列中一般只包含一个 idle 线程。如图例在优先级队列 1#和 2#中，分别有线程 A – 线程 C。由于线程 A、B 的优先级比线程 C 的高，所以此时线程 C 得不到运行，必须要等待线程 A – 线程 B 都让出处理机后才能得到执行。

图 2 - 1 线程优先级队列



RT-Thread 中允许创建相同优先级的线程。相同优先级的线程采用时间片轮转进行调度（也就是通常说的分时调度器）。如上图例中所示的线程 A 和线程 B，假设它们一次最大允许运行的时间片分别是 10 个时钟节拍和 7 个时钟节拍。那么线程 B 的运行需要在线程 A 运行完它的时间片（10 个时钟节拍）后才能获得运行（当然如果线程 A 被挂起了，它会马上获得运行）。

由于 RT-Thread 调度器的实现是采用优先级链表的方式，所以系统中的总线程数不受限制，只和系统所能提供的内存资源相关。

为了保证系统的实时性，系统尽最大可能地保证高优先级的线程得以运行。线程调度的原则是一旦任务状态发生了改变，并且当前运行的线程优先级小于优先级队列组中线程最高优先级时，立刻进行线程切换（除非当前系统处于中断处理程序中或禁止线程切换的状态）。RT-Thread 的调度器算法是基于优先级位图的算法，其时间复杂度是 O(1)，即和线程的多少是不相关的。

2.3 线程控制块

线程控制块是操作系统用于控制线程的一块数据结构，它会存放线程的一些信息，例如优先级，线程名称等，也包含线程于线程之间的链表结构，线程等待事件集合等。

在 RT-Thread 中，线程控制块由结构体 `struct rt_thread`（如下代码中所示）表示，另外一种写法是 `rt_thread_t`，表示的是线程的句柄，在 C 的实现上是指向线程控制块的指

针。

代码 2-1 线程控制块

```
typedef struct rt_thread* rt_thread_t;

struct rt_thread
{
    /* rt object */
    char        name[RT_NAME_MAX];          /* the name of thread */
    rt_uint8_t  type;                       /* type of object */
    rt_uint8_t  flags;                      /* thread's flags */

    rt_list_t   list;                       /* the object list */

    rt_thread_t tid;                        /* the thread id */
    rt_list_t   tlist;                      /* the thread list */

    /* stack point and entry */
    void*       sp;                         /* stack point */
    void*       entry;                      /* entry */
    void*       parameter;                  /* parameter */
    void*       stack_addr;                 /* stack address */
    rt_uint16_t stack_size;                 /* stack size */

    /* error code */
    rt_err_t    error;                      /* error code */

    /* priority */
    rt_uint8_t  current_priority;           /* current priority */
    rt_uint8_t  init_priority;              /* initialized priority */
#if RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t  number;
    rt_uint8_t  high_mask;
#endif
    rt_uint32_t number_mask;

#if defined(RT_USING_EVENT) || defined(RT_USING_FASTEVENT)
    /* thread event */
    rt_uint32_t event_set;
    rt_uint8_t  event_info;
#endif

    rt_uint8_t  stat;                       /* thread stat */

    rt_ubase_t  init_tick;                  /* thread's tick */
    rt_ubase_t  remaining_tick;             /* remaining tick */

    struct rt_timer thread_timer;           /* thread timer */

    rt_uint32_t user_data;                  /* user data */
};
```

最后的一个成员 `user_data` 可由用户挂接一些数据信息到线程控制块中，以实现类似线程私有数据。

2.4 线程状态

在线程运行的过程中，在一个时间内只允许一个线程中处理器中运行，即线程会有多种不同的线程状态，如运行态，非运行态等。在 RT-Thread 中，线程包含三种状态，操作系统会自动根据它运行的情况而动态调整它的状态。

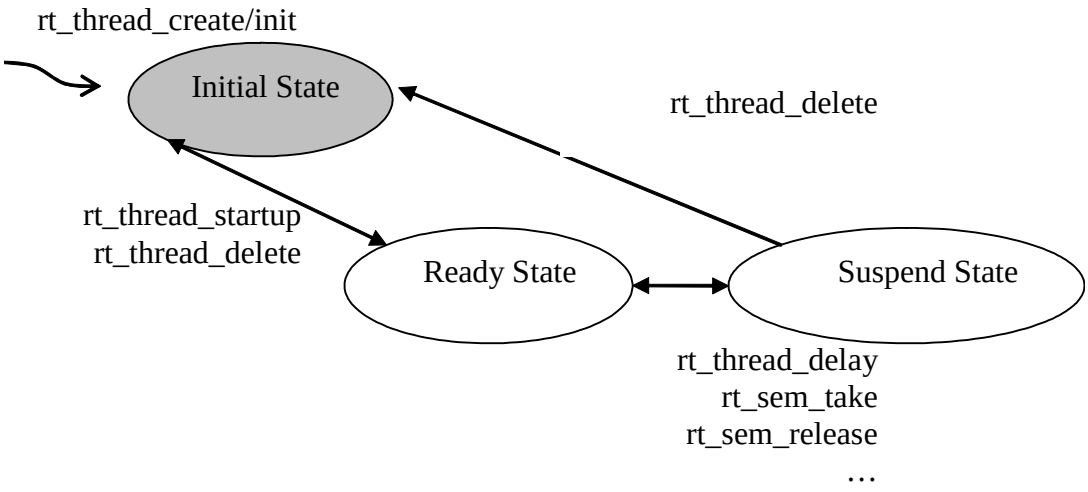
RT-Thread 中的三种线程状态：

RT_THREAD_INIT/CLOSE	线程初始状态。当线程刚开始创建还没开始运行时就处于这个状态；当线程运行结束时也处于这个状态。在这个状态下，线程会参与调度
RT_THREAD_SUSPEND	挂起态。线程此时被挂起：它可能因为资源不可用而等待挂起；或主动延时一段时间而被挂起。在这个状态下，线程不参与调度
RT_THREAD_READY	就绪态。线程正在运行；或当前线程运行完让出处理机后，操作系统寻找最高优先级的就绪态线程运行

RT-Thread RTOS 提供一系列的操作系统调用接口，使得线程的状态在这三个状态之间来回的变换。例如一个就绪态的任务由于申请一个资源(例如使用 `rt_sem_take`)，而有可能进入阻塞态。又如，一个外部中断发生，转入中断处理函数，中断处理函数释放了某个资源，导致了当前运行任务的切换，唤醒了另一阻塞态的任务，改变其状态为就绪态等等。

三种状态间的转换关系如下图所示：

图 2 - 2 线程状态转换图



线程通过调用函数 `rt_thread_create/init` 调用进入到初始状态（`Initial State, RT_THREAD_INIT`），通过函数 `rt_thread_startup` 调用后进入到就绪状态（`Ready State，RT_THREAD_READY`）。当这个线程调用 `rt_thread_delay，rt_sem_take，rt_mb_recv` 等函数时，将主动挂起或由于获取不到资源进入到挂起状态（`Suspend`

State，RT_THREAD_SUSPEND）。在挂起状态的线程，如果它等待超时依然未获得资源或由于其他线程释放了资源，它将返回到就绪状态。

2.5 空闲线程

空闲线程是系统线程中一个比较特殊的线程，它具备最低的优先级，当系统中无其他线程可运行时，调度器将自动调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起。在 RT-Thread 中空闲线程提供了钩子函数，以让系统在空闲的时候执行一定任务，例如系统运行指示灯闪烁等。

2.6 调度器相关接口

2.6.1 调度器初始化

在系统启动时需要执行调度器的初始化，以初始化调度器用到的一些全局变量。调度器初始化可以调用以下接口。

```
void rt_system_scheduler_init(void)
```

2.6.2 启动调度器

在系统完成初始化后切换到第一个线程，可以调用如下接口。

```
void rt_system_scheduler_start(void)
```

在调用这个函数时，它会查找系统中最高的就绪态的线程，然后切换过去运行。
注：在调用这个函数前，必须先做 idle 线程的初始化。此函数是永远不会返回的。

2.6.3 执行调度

让调度器执行一次线程的调度可通过如下接口。

```
void rt_schedule(void)
```

调用这个函数后，系统会计算一次系统中就绪态的线程，如果存在比当前线程更高优先级的线程时，系统将会切换到高优先级的线程去。通常情况下，用户不需要直接调用这个函数。

注：在中断服务例程中也可以调用这个函数，如果满足任务切换的条件，它会记录下中断前的线程及切换到更高优先级的线程，在中断服务例程处理完毕后执行正真的线程上下文切换。

2.7 线程相关接口

2.7.1 线程创建

一个线程要成为可执行的对象就必须由操作系统内核来为它创建/初始化一个线程句柄。可以通过如下的接口来创建一个线程。

```
rt_thread_t rt_thread_create (const char* name,
    void (*entry)(void* parameter), void* parameter,
    rt_uint32_t stack_size,
    rt_uint8_t priority, rt_uint32_t tick)
```

在调用这个函数时，将为线程指定名称，线程入口位置，入口参数，线程栈大小，优先级及时间片大小。线程名称的最大长度由宏 RT_NAME_MAX 指定，多余部分会被自动截掉。栈大小的单位是字节，在大多数系统中需要做对齐（例如 ARM 体系结构中需要向 4 字节对齐）。线程的优先级范围从 0 ～ 255，数值越小优先级越高。时间片的单位是操作系统的时钟节拍。

调用这个函数后，系统会从动态堆内存中分配一个线程句柄（或者叫做 TCB）以及按照参数中指定的栈大小从动态堆内存中分配相应的空间。

创建一个线程的例子如下代码所示。

代码 2-2 创建线程

```
#include <rtthread.h>

/* 线程入口 */
static void entry(void* parameter)
{
    rt_uint32_t count = 0;
    while (1)
    {
        rt_kprintf("count:%d\n", ++count);
        rt_thread_delay(50);
    }
}

/* 用户应用程序入口 */
int rt_application_init()
{
    /* 创建一个线程，入口时entry函数 */
    rt_thread_t thread = rt_thread_create("t1",
        entry, RT_NULL,
        1024, 200, 10);
    if (thread != RT_NULL)
        rt_thread_startup(thread);

    return 0;
}
```

2.7.2 线程删除

当需要删除用 rt_thread_create 创建出的线程时（例如线程出错无法恢复时），可以使用以下接口：
rt_err_t rt_thread_delete (rt_thread_t thread)

调用该接口后，线程对象将会被移出线程队列并且从内核对象管理器中删除，线程占用的堆栈空间也会被释放。

注：线程运行完成，自动结束时，系统会自动删除线程。用 rt_thread_init 初始化的静态线程请不要使用此接口删除。

线程删除例子如下。

代码 2-3 删除线程

```
#include <rtthread.h>

void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;
    while (1)
    {
        rt_kprintf("count:%d\n", ++count);
        rt_thread_delay(50);
    }
}

rt_uint32_t to_delete_thread1 = 0;
rt_thread_t thread1, thread2;
void thread2_entry(void* parameter)
{
    while (1)
    {
        if (to_delete_thread1 == 1)
        {
            /* to_delete_thread1置位，删除thread1 */
            rt_thread_delete(thread1);

            /* 删除完成后退出 */
            return ;
        }

        /* to_delete_thread1未置位，等待100个时钟节拍后再查询 */
        rt_thread_delay(100);
    }
}

int rt_application_init()
{
    /* 创建thread1并运行 */
    thread1 = rt_thread_create("t1",
```

```
        thread1_entry, RT_NULL,
        1024, 200, 10);
    if (thread1 != RT_NULL) rt_thread_startup(thread1);

    /* 创建thread2并运行 */
    thread2 = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        1024, 120, 10);
    if (thread2 != RT_NULL) rt_thread_startup(thread2);

    return 0;
}
```

2.7.3 线程初始化

线程的初始化可以使用以下接口完成：

```
rt_err_t rt_thread_init(struct rt_thread* thread,
    const char* name,
    void (*entry)(void* parameter), void* parameter,
    void* stack_start, rt_uint32_t stack_size,
    rt_uint8_t priority, rt_uint32_t tick);
```

通常线程初始化函数用来初始化静态线程对象，线程句柄，线程栈由用户提供，一般都设置为全局变量在编译时被分配，内核不负责动态分配空间。

线程初始化例子如下所示。

代码 2-4 线程初始化

```
#include <rtthread.h>

static rt_uint8_t thread_stack[512];
static struct rt_thread thread;

/* 线程入口 */
static void entry(void* parameter)
{
    int i;
    rt_thread_t self;

    /* 获得当前线程句柄 */
    self = rt_thread_self();

    while (1)
    {
        rt_kprintf("thread[%s] count %d\n", self->name, ++i);
        rt_thread_delay(100);
    }
}

int rt_application_init()
{
    /* 初始化线程 */
```

```
        rt_thread_init(&thread,
            "thread1",
            entry, RT_NULL,
            &thread_stack[0], sizeof(thread_stack),
            200, 10);
        /* 启动线程 */
        rt_thread_startup(&thread);

        return 0;
    }
```

2.7.4 线程脱离

脱离线程将使线程对象被从线程队列和内核对象管理器中删除。脱离线程使用以下接口。

```
rt_err_t rt_thread_detach (rt_thread_t thread)
```

注：这个函数接口是和 rt_thread_delete 相对应的，rt_thread_delete 操作的对象是 rt_thread_create 创建的句柄，而 rt_thread_detach 操作的对象是使用 rt_thread_init 初始化的句柄。

2.7.5 线程启动

创建/初始化的线程对象的状态是 RT_THREAD_INIT 状态，并未进入调度队列，可以调用如下接口启动一个创建/初始化的线程对象：

```
rt_err_t rt_thread_startup (rt_thread_t thread)
```

2.7.6 当前线程

在程序的运行过程中，相同的一段代码可能会被多个线程执行，在执行的时候可以通过下面的接口获得当前执行的线程句柄。

```
rt_thread_t rt_thread_self (void)
```

注：请不要在中断服务程序中调用此函数，因为它并不能准确获得当前的执行线程。

2.7.7 线程让出处理机

当前线程的时间片用完或者该线程自动要求让出处理器资源时，它不再占有处理机，调度器会选择下一个最高优先级的线程执行。这时，放弃处理器资源的线程仍然在就绪队列中。线程让出处理器使用以下接口：

```
rt_err_t rt_thread_yield ()
```

调用该接口后，线程首先把自己从它所在的队列中删除，然后把自己挂到与该线程优先级对

应的就绪线程链表的尾部，然后激活调度器切换到优先级最高的线程。

线程让出处理机代码例子如下所示。

代码 2-5 让出处理机

```
void funcion()
{
    ...
    rt_thread_yield();
    ...
}
```

注：rt_thread_yield 函数和 rt_schedule 函数比较相像，但在相同优先级线程存在时，系统的行为是完全不一样的。当有相同优先级就绪线程存在，并且系统中不存在更高优先级的就绪线程时，执行 rt_thread_yield 函数后，当前线程被换出。而执行 rt_schedule 函数后，当前线程并不被换成。

2.7.8 线程睡眠

在实际应用中，经常需要线程延迟一段时间，指定的时间到达后，线程重新运行，线程睡眠使用以下接口：

```
rt_err_t rt_thread_sleep(rt_tick_t tick)
rt_err_t rt_thread_delay(rt_tick_t tick)
```

以上两个接口作用是相同的，调用该线程可以使线程暂时挂起指定时间，它接受一个参数，该参数指定了线程的休眠时间（时钟节拍数）。

2.7.9 线程挂起

当线程调用 rt_thread_delay，rt_sem_take，rt_mb_recv 等函数时，将主动挂起。或者由于线程获取不到资源，它也会进入到挂起状态。在挂起状态的线程，如果等待的资源超时或由于其他线程释放资源，它将返回到就绪状态。挂起线程使用以下接口：

```
rt_err_t rt_thread_suspend (rt_thread_t thread)
```

注：如果挂起当前任务，需要在调用这个函数后，紧接着调用 rt_schedule 函数进行手动的线程上下文切换。

挂起线程的代码例子如下所示。

代码 2-6 挂起线程代码

```
#include <rtthread.h>

/* 线程句柄 */
rt_thread_t thread = RT_NULL;

/* 线程入口 */
```

```
static void entry(void* parameter)
{
    while (1)
    {
        /* 挂起自己 */
        rt_thread_suspend(thread);

        /* 调用rt_thread_suspend虽然把thread从就绪队列中删除，
         * 但代码依然在运行，需要手动让调度器调度一次
         */
        rt_schedule();

        /* 运行到此处，相当于thread已经被唤醒 */
        rt_kprintf("thread is resumed\n");
    }
}

int rt_application_init()
{
    /* 创建线程 */
    thread = rt_thread_create("tid", entry, RT_NULL, 1024, 250, 20);

    /* 启动线程 */
    rt_thread_startup(thread);

    return 0;
}
```

2.7.10 线程恢复

线程恢复使得挂起的线程重新进入就绪状态。线程恢复使用以下接口：

```
rt_err_t rt_thread_resume (rt_thread_t thread)
```

恢复挂起线程的代码例子如下所示。

代码 2-7 恢复挂起线程

```
#include <rtthread.h>

rt_thread_t thread = RT_NULL;
void function ()
{
   ...
   rt_thread_resume(thread);
   ...
}
```

线程控制

```
rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg)
```

2.7.11 初始化空闲线程

在系统调度器运行前，必须通过调用如下的函数初始化空闲线程。

```
void rt_thread_idle_init(void)
```

2.7.12 设置空闲线程钩子

可以调用如下的函数，设置空闲线程运行时执行的钩子函数。

```
void rt_thread_idle_set_hook(void (*hook)())
```

当空闲线程运行时会自动执行设置的钩子函数，由于空闲线程具有系统的最低优先级，所以只有在空闲时刻才会执行此钩子函数。空闲线程是一个线程状态永远为就绪状态的线程，因此挂入的钩子函数必须保证空闲线程永远不会处于挂起状态，例如 `rt_thread_delay`，`rt_sem_take` 等可能会导致线程挂起的函数不能使用。