线程管理

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Tuesday 25th July, 2023

目录

在日常生活中，我们要完成一个大任务，一般会将它分解成多个简单、容易解决的小问题，小问题逐个被解决，大问题也就随之解决了。在多线程操作系统中，也同样需要开发人员把一个复杂的应用分解成多个小的、可调度的、序列化的程序单元，当合理地划分任务并正确地执行时，这种设计能够让系统满足实时系统的性能及时间的要求，例如让嵌入式系统执行这样的任务，系统通过传感器采集数据，并通过显示屏将数据显示出来，在多线程实时系统中，可以将这个任务分解成两个子任务，如下图所示，一个子任务不间断地读取传感器数据，并将数据写到共享内存中，另外一个子任务周期性的从共享内存中读取数据，并将传感器数据输出到显示屏上。

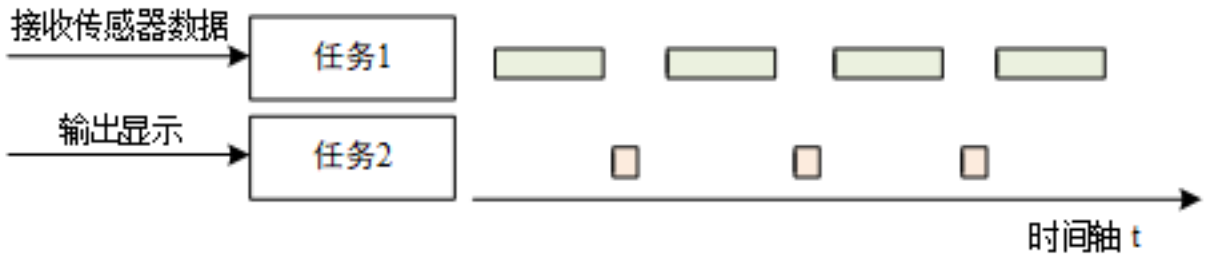

图 1: *传感器数据接收任务与显示任务的切换执行*

在 RT-Thread 中，与上述子任务对应的程序实体就是线程，线程是实现任务的载体，它是 RT-Thread 中最基本的调度单位，它描述了一个任务执行的运行环境，也描述了这个任务所处的优先等级，重要的任务可设置相对较高的优先级，非重要的任务可以设置较低的优先级，不同的任务还可以设置相同的优先级，轮流运行。

当线程运行时，它会认为自己是以独占 CPU 的方式在运行，线程执行时的运行环境称为上下文，具体来说就是各个变量和数据，包括所有的寄存器变量、堆栈、内存信息等。

本章将分成 5 节内容对 RT-Thread 线程管理进行介绍，读完本章，读者会对 RT-Thread 的线程管理机制有比较深入的了解，如：线程有哪些状态、如何创建一个线程、为什么会存在空闲线程等问题，心中也会有一个明确的答案了。

1 线程管理的功能特点

RT-Thread 线程管理的主要功能是对线程进行管理和调度，系统中总共存在两类线程，分别是系统线程和用户线程，系统线程是由 RT-Thread 内核创建的线程，用户线程是由应用程序创建的线程，这两类线程都会从内核对象容器中分配线程对象，当线程被删除时，也会被从对象容器中删除，如下图所示，每个线程都有重要的属性，如线程控制块、线程栈、入口函数等。

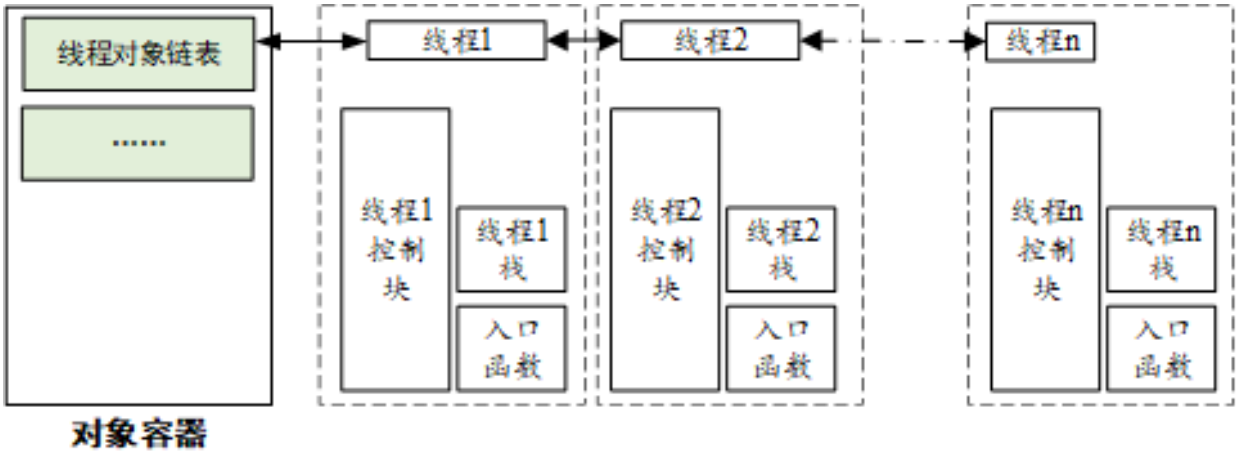

图 2: *对象容器与线程对象*

RT-Thread 的线程调度器是抢占式的，主要的工作就是从就绪线程列表中查找最高优先级线程，保证最高优先级的线程能够被运行，最高优先级的任务一旦就绪，总能得到 CPU 的使用权。

当一个运行着的线程使一个比它优先级高的线程满足运行条件，当前线程的 CPU 使用权就被剥夺了，或者说被让出了，高优先级的线程立刻得到了 CPU 的使用权。

如果是中断服务程序使一个高优先级的线程满足运行条件，中断完成时，被中断的线程挂起，优先级高的线程开始运行。

当调度器调度线程切换时，先将当前线程上下文保存起来，当再切回到这个线程时，线程调度器将该线程的上下文信息恢复。

2 线程的工作机制

2.1 线程控制块

在 RT-Thread 中，线程控制块由结构体 struct rt_thread 表示，线程控制块是操作系统用于管理线程的一个数据结构，它会存放线程的一些信息，例如优先级、线程名称、线程状态等，也包含线程与线程之间连接用的链表结构，线程等待事件集合等，详细定义如下：

```
/* 线程控制块 */
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];      /* 线程名称 */
    rt_uint8_t  type;                    /* 对象类型 */
    rt_uint8_t  flags;                   /* 标志位 */

    rt_list_t   list;                    /* 对象列表 */
    rt_list_t   tlist;                   /* 线程列表 */

    /* 栈指针与入口指针 */
    void       *sp;                      /* 栈指针 */
    void       *entry;                   /* 入口函数指针 */
```

```
    void       *parameter;               /* 参数 */
    void       *stack_addr;              /* 栈地址指针 */
    rt_uint32_t stack_size;             /* 栈大小 */

    /* 错误代码 */
    rt_err_t    error;                  /* 线程错误代码 */
    rt_uint8_t  stat;                   /* 线程状态 */

    /* 优先级 */
    rt_uint8_t  current_priority;    /* 当前优先级 */
    rt_uint8_t  init_priority;        /* 初始优先级 */
    rt_uint32_t number_mask;

    ......

    rt_ubase_t  init_tick;              /* 线程初始化计数值 */
    rt_ubase_t  remaining_tick;         /* 线程剩余计数值 */

    struct rt_timer thread_timer;      /* 内置线程定时器 */

    void (*cleanup)(struct rt_thread *tid);  /* 线程退出清除函数 */
    rt_uint32_t user_data;                   /* 用户数据 */
};
```

其中 init_priority 是线程创建时指定的线程优先级，在线程运行过程当中是不会被改变的（除非用户执行线程控制函数进行手动调整线程优先级）。cleanup 会在线程退出时，被空闲线程回调一次以执行用户设置的清理现场等工作。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，以提供类似线程私有数据的实现。

2.2 线程重要属性

2.2.1. 线程栈

RT-Thread 线程具有独立的栈，当进行线程切换时，会将当前线程的上下文存在栈中，当线程要恢复运行时，再从栈中读取上下文信息，进行恢复。

线程栈还用来存放函数中的局部变量：函数中的局部变量从线程栈空间中申请；函数中局部变量初始时从寄存器中分配（ARM 架构），当这个函数再调用另一个函数时，这些局部变量将放入栈中。

对于线程第一次运行，可以以手工的方式构造这个上下文来设置一些初始的环境：入口函数（PC 寄存器）、入口参数（R0 寄存器）、返回位置（LR 寄存器）、当前机器运行状态（CPSR 寄存器）。

线程栈的增长方向是芯片构架密切相关的，RT-Thread 3.1.0 以前的版本，均只支持栈由高地址向低地址增长的方式，对于 ARM Cortex-M 架构，线程栈可构造如下图所示。

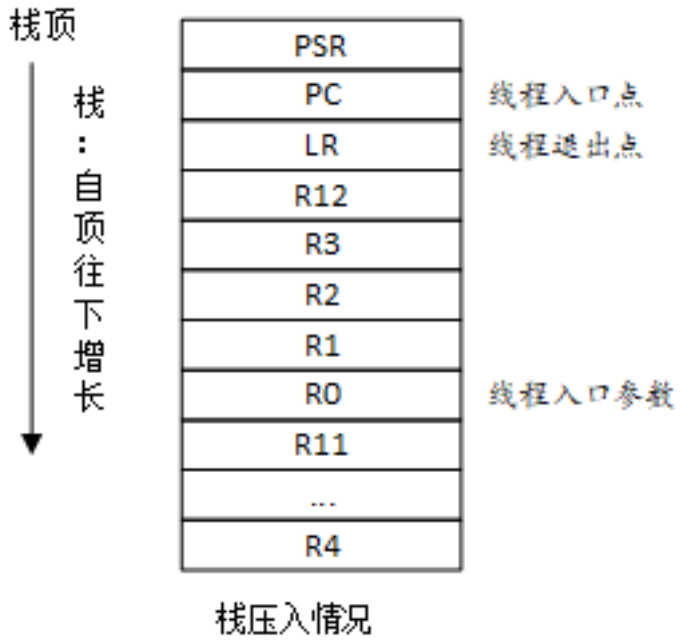

图 3: *线程栈 (ARM)*

线程栈大小可以这样设定，对于资源相对较大的 MCU，可以适当设计较大的线程栈；也可以在初始时设置较大的栈，例如指定大小为 1K 或 2K 字节，然后在 FinSH 中用 list_thread 命令查看线程运行的过程中线程所使用的栈的大小，通过此命令，能够看到从线程启动运行时，到当前时刻点，线程使用的最大栈深度，而后加上适当的余量形成最终的线程栈大小，最后对栈空间大小加以修改。

2.2.2. 线程状态

线程运行的过程中，同一时间内只允许一个线程在处理器中运行，从运行的过程上划分，线程有多种不同的运行状态，如初始状态、挂起状态、就绪状态等。在 RT-Thread 中，线程包含五种状态，操作系统会自动根据它运行的情况来动态调整它的状态。RT-Thread 中线程的五种状态，如下表所示：

状态	描述
初始状态	当线程刚开始创建还没开始运行时就处于初始状态；在初始状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_INIT
就绪状态	在就绪状态下，线程按照优先级排队，等待被执行；一旦当前线程运行完毕让出处理器，操作系统会马上寻找最高优先级的就绪态线程运行。此状态在 RT-Thread 中的宏定义为 RT_THREAD_READY
运行状态	线程当前正在运行。在单核系统中，只有 rt_thread_self() 函数返回的线程处于运行状态；在多核系统中，可能就不止这一个线程处于运行状态。此状态在 RT-Thread 中的宏定义为 RT_THREAD_RUNNING
挂起状态	也称阻塞态。它可能因为资源不可用而挂起等待，或线程主动延时一段时间而挂起。在挂起状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_SUSPEND
关闭状态	当线程运行结束时将处于关闭状态。关闭状态的线程不参与线程的调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_CLOSE

2.2.3. 线程优先级

RT-Thread 线程的优先级是表示线程被调度的优先程度。每个线程都具有优先级，线程越重要，赋予的优先级就应越高，线程被调度的可能才会越大。

RT-Thread 最大支持 256 个线程优先级 (0~255)，数值越小的优先级越高，0 为最高优先级。在一些资源比较紧张的系统中，可以根据实际情况选择只支持 8 个或 32 个优先级的系统配置；对于 ARM Cortex-M 系列，普遍采用 32 个优先级。最低优先级默认分配给空闲线程使用，用户一般不使用。在系统中，当有比当前线程优先级更高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理器运行。

2.2.4. 时间片

每个线程都有时间片这个参数，但时间片仅对优先级相同的就绪态线程有效。系统对优先级相同的就绪态线程采用时间片轮转的调度方式进行调度时，时间片起到约束线程单次运行时长的作用，其单位是一个系统节拍（OS Tick），详见《时钟管理》章节。假设有 2 个优先级相同的就绪态线程 A 与 B，A 线程的时间片设置为 10，B 线程的时间片设置为 5，那么当系统中不存在比 A 优先级高的就绪态线程时，系统会在 A、B 线程间来回切换执行，并且每次对 A 线程执行 10 个节拍的时长，对 B 线程执行 5 个节拍的时长，如下图。

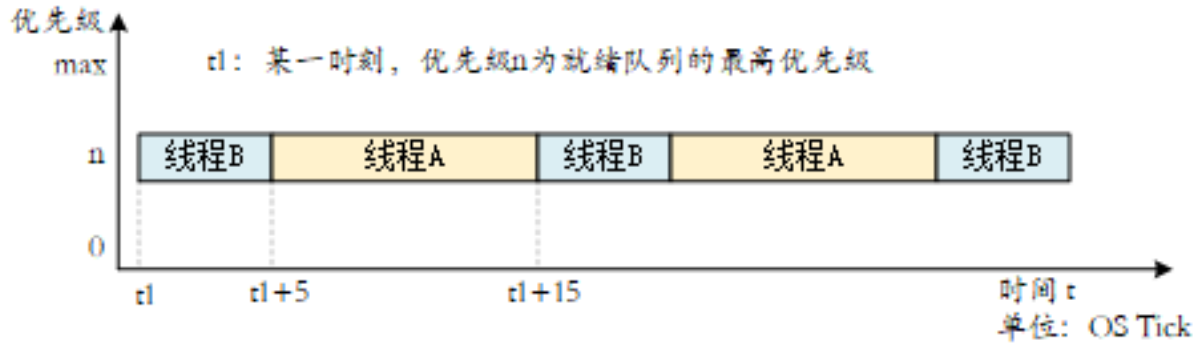

图 4: *相同优先级时间片轮转*

2.2.5. 线程的入口函数

线程控制块中的 entry 是线程的入口函数，它是线程实现预期功能的函数。线程的入口函数由用户设计实现，一般有以下两种代码形式：

-无限循环模式：

在实时系统中，线程通常是被动式的：这个是由实时系统的特性所决定的，实时系统通常总是等待外界事件的发生，而后进行相应的服务：

```
void thread_entry(void* paramenter)
{
    while (1)
    {
    /* 等待事件的发生 */

    /* 对事件进行服务、进行处理 */
    }
```

```
}
```

线程看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是作为一个实时系统，一个优先级明确的实时系统，如果一个线程中的程序陷入了死循环操作，那么比它优先级低的线程都将不能够得到执行。所以在实时操作系统中必须注意的一点就是：线程中不能陷入死循环操作，必须要有让出 CPU 使用权的动作，如循环中调用延时函数或者主动挂起。用户设计这种无限循环的线程的目的，就是为了让这个线程一直被系统循环调度运行，永不删除。

-顺序执行或有限次循环模式：

如简单的顺序语句、do while() 或 for() 循环等，此类线程不会循环或不会永久循环，可谓是 “一次性” 线程，一定会被执行完毕。在执行完毕后，线程将被系统自动删除。

```
static void thread_entry(void* parameter)
{
    /* 处理事务 #1 */
    …
    /* 处理事务 #2 */
    …
    /* 处理事务 #3 */
}
```

2.2.6. 线程错误码

一个线程就是一个执行场景，错误码是与执行环境密切相关的，所以每个线程配备了一个变量用于保存错误码，线程的错误码有以下几种：

```
#define RT_EOK            0 /* 无错误      */
#define RT_ERROR          1 /* 普通错误      */
#define RT_ETIMEOUT       2 /* 超时错误      */
#define RT_EFULL          3 /* 资源已满      */
#define RT_EEMPTY         4 /* 无资源      */
#define RT_ENOMEM         5 /* 无内存      */
#define RT_ENOSYS         6 /* 系统不支持      */
#define RT_EBUSY          7 /* 系统忙      */
#define RT_EIO            8 /* IO 错误       */
#define RT_EINTR          9 /* 中断系统调用    */
#define RT_EINVAL        10 /* 非法参数       */
```

2.3 线程状态切换

RT-Thread 提供一系列的操作系统调用接口，使得线程的状态在这五个状态之间来回切换。几种状态间的转换关系如下图所示：

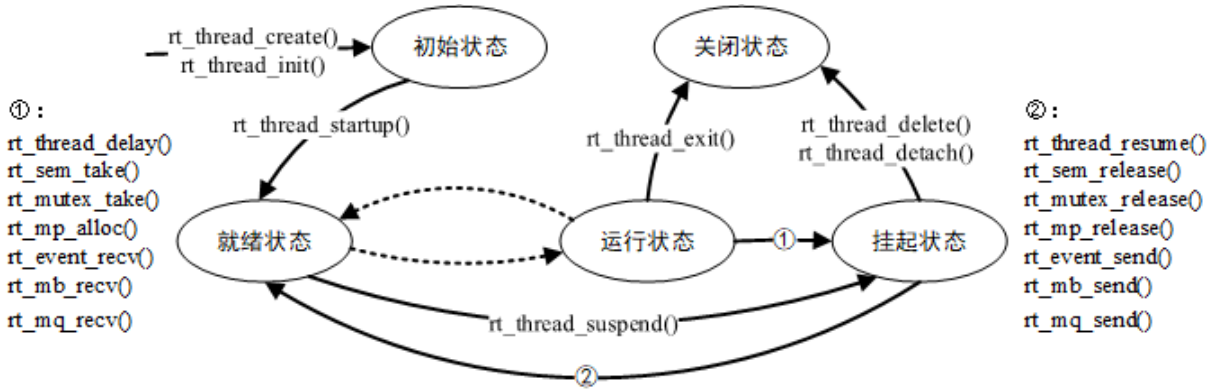

图 5: *线程状态转换图*

线程通过调用函数 rt_thread_create/init() 进入到初始状态（RT_THREAD_INIT）；初始状态的线程通过调用函数 rt_thread_startup() 进入到就绪状态（RT_THREAD_READY）；就绪状态的线程被调度器调度后进入运行状态（RT_THREAD_RUNNING）；当处于运行状态的线程调用 rt_thread_delay()，rt_sem_take()，rt_mutex_take()，rt_mb_recv() 等函数或者获取不到资源时，将进入到挂起状态（RT_THREAD_SUSPEND）；处于挂起状态的线程，如果等待超时依然未能获得资源或由于其他线程释放了资源，那么它将返回到就绪状态。挂起状态的线程，如果调用 rt_thread_delete/detach() 函数，将更改为关闭状态（RT_THREAD_CLOSE）；而运行状态的线程，如果运行结束，就会在线程的最后部分执行 rt_thread_exit() 函数，将状态更改为关闭状态。

> 注意：RT-Thread V5.0.0 以下的版本中，实际上线程并不存在运行状态，就绪状态和运行状态是等同的。

2.4 系统线程

前文中已提到，系统线程是指由系统创建的线程，用户线程是由用户程序调用线程管理接口创建的线程，在 RT-Thread 内核中的系统线程有空闲线程和主线程。

2.4.1. 空闲线程

空闲线程（idle）是系统创建的最低优先级的线程，线程状态永远为就绪态。当系统中无其他就绪线程存在时，调度器将调度到空闲线程，它通常是一个死循环，且永远不能被挂起。另外，空闲线程在 RT-Thread 也有着它的特殊用途：

若某线程运行完毕，系统将自动删除线程：自动执行 rt_thread_exit() 函数，先将该线程从系统就绪队列中删除，再将该线程的状态更改为关闭状态，不再参与系统调度，然后挂入 rt_thread_defunct 僵尸队列（资源未回收、处于关闭状态的线程队列）中，最后空闲线程会回收被删除线程的资源。

空闲线程也提供了接口来运行用户设置的钩子函数，在空闲线程运行时会调用该钩子函数，适合处理功耗管理、看门狗喂狗等工作。空闲线程必须有得到执行的机会，即其他线程不允许一直 while(1) 死卡，必须调用具有阻塞性质的函数；否则例如线程删除、回收等操作将无法得到正确执行。

2.4.2. 主线程

在系统启动时，系统会创建 main 线程，它的入口函数为 main_thread_entry()，用户的应用入口函数 main() 就是从这里真正开始的，系统调度器启动后，main 线程就开始运行，过程如下图，用户可以在 main() 函数里添加自己的应用程序初始化代码。

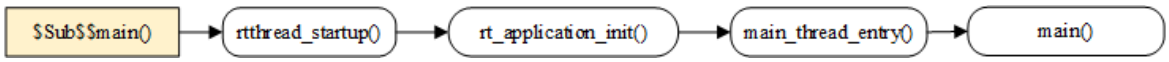

图 6: *主线程调用过程*

3 线程的管理方式

本章前面 2 节对线程的功能与工作机制进行了概念上的讲解，相信大家对线程已经不再陌生。本节将深入到 RT-Thread 线程的各个接口，并给出部分源码，帮助读者在代码层次上理解线程。

下图描述了线程的相关操作，包含：创建 / 初始化线程、启动线程、运行线程、删除 / 脱离线程。可以使用 rt_thread_create() 创建一个动态线程，使用 rt_thread_init() 初始化一个静态线程，动态线程与静态线程的区别是：动态线程是系统自动从动态内存堆上分配栈空间与线程句柄（初始化 heap 之后才能使用 create 创建动态线程），静态线程是由用户分配栈空间与线程句柄。

图 7: *线程相关操作*

3.1 创建和删除线程

一个线程要成为可执行的对象，就必须由操作系统的内核来为它创建一个线程。可以通过如下的接口创建一个动态线程：

```
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);
```

调用这个函数时，系统会从动态堆内存中分配一个线程句柄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。分配出来的栈空间是按照 rtconfig.h 中配置的 RT_ALIGN_SIZE 方式对齐。线程创建 rt_thread_create() 的参数和返回值见下表：

参数	描述
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。优先级范围根据系统配置情况（rtconfig.h 中的 RT_THREAD_PRIORITY_MAX 宏定义），如果支持的是 256 级优先级，那么范围是从 0~255，数值越小优先级越高，0 代表最高优先级
tick	线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行
返回	——
thread	线程创建成功，返回线程句柄
RT_NULL	线程创建失败

对于一些使用 rt_thread_create() 创建出来的线程，当不需要使用，或者运行出错时，我们可以使用下面的函数接口来从系统中把线程完全删除掉：

```
rt_err_t rt_thread_delete(rt_thread_t thread);
```

调用该函数后，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，收回的空间将重新用于其他的内存分配。实际上，用 rt_thread_delete() 函数删除线程接口，仅仅是把相应的线程状态更改为 RT_THREAD_CLOSE 状态，然后放入到 rt_thread_defunct 队列中；而真正的删除动作（释放线程控制块和释放线程栈）需要到下一次执行空闲线程时，由空闲线程完成最后的线程删除动作。线程删除 rt_thread_delete() 接口的参数和返回值见下表：

参数	描述
thread	要删除的线程句柄
返回	——
RT_EOK	删除线程成功
-RT_ERROR	删除线程失败

注：rt_thread_create() 和 rt_thread_delete() 函数仅在使能了系统动态堆时才有效（即 RT_USING_HEAP 宏定义已经定义了）。

3.2 初始化和脱离线程

线程的初始化可以使用下面的函数接口完成，来初始化静态线程对象：

```
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);
```

静态线程的线程句柄（或者说线程控制块指针）、线程栈由用户提供。静态线程是指线程控制块、线程运行栈一般都设置为全局变量，在编译时就被确定、被分配处理，内核不负责动态分配内存空间。需要注意的是，用户提供的栈首地址需做系统对齐（例如 ARM 上需要做 4 字节对齐）。线程初始化接口 rt_thread_init() 的参数和返回值见下表：

参数	描述
thread	线程句柄。线程句柄由用户提供出来，并指向对应的线程控制块内存地址
name	线程的名称；线程名称的最大长度由 rtconfig.h 中定义的 RT_NAME_MAX 宏指定，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_start	线程栈起始地址
stack_size	线程栈大小，单位是字节。在大多数系统中需要做栈空间地址对齐（例如 ARM 体系结构中需要向 4 字节地址对齐）
priority	线程的优先级。优先级范围根据系统配置情况（rtconfig.h 中的 RT_THREAD_PRIORITY_MAX 宏定义），如果支持的是 256 级优先级，那么范围是从 0 ～ 255，数值越小优先级越高，0 代表最高优先级
tick	线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行
返回	——
RT_EOK	线程创建成功
-RT_ERROR	线程创建失败

对于用 rt_thread_init() 初始化的线程，使用 rt_thread_detach() 将使线程对象在线程队列和内核对象管理器中被脱离。线程脱离函数如下：

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

线程脱离接口 rt_thread_detach() 的参数和返回值见下表：

参数	描述
thread	线程句柄，它应该是由 rt_thread_init 进行初始化的线程句柄。
返回	——
RT_EOK	线程脱离成功
-RT_ERROR	线程脱离失败

这个函数接口是和 rt_thread_delete() 函数相对应的，rt_thread_delete() 函数操作的对象是 rt_thread_create() 创建的句柄，而 rt_thread_detach() 函数操作的对象是使用 rt_thread_init() 函数初始化的线程控制块。同样，线程本身不应调用这个接口脱离线程本身。

3.3 启动线程

创建（初始化）的线程状态处于初始状态，并未进入就绪线程的调度队列，我们可以在线程初始化 / 创建成功后调用下面的函数接口让该线程进入就绪态：

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

当调用这个函数时，将把线程的状态更改为就绪状态，并放到相应优先级队列中等待调度。如果新启动的线程优先级比当前线程优先级高，将立刻切换到这个线程。线程启动接口 rt_thread_startup() 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	——
RT_EOK	线程启动成功
-RT_ERROR	线程启动失败

3.4 获得当前线程

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函数接口获得当前执行的线程句柄：

```
rt_thread_t rt_thread_self(void);
```

该接口的返回值见下表：

返回	描述
thread	当前运行的线程句柄
RT_NULL	失败，调度器还未启动

3.5 使线程让出处理器资源

当前线程的时间片用完或者该线程主动要求让出处理器资源时，它将不再占有处理器，调度器会选择相同优先级的下一个线程执行。线程调用这个接口后，这个线程仍然在就绪队列中。线程让出处理器使用下面的函数接口：

```
rt_err_t rt_thread_yield(void);
```

调用该函数后，当前线程首先把自己从它所在的就绪优先级线程队列中删除，然后把自己挂到这个优先级队列链表的尾部，然后激活调度器进行线程上下文切换（如果当前优先级只有这一个线程，则这个线程继续执行，不进行上下文切换动作）。

rt_thread_yield() 函数和 rt_schedule() 函数比较相像，但在有相同优先级的其他就绪态线程存在时，系统的行为却完全不一样。执行 rt_thread_yield() 函数后，当前线程被换出，相同优先级的下一个就绪线程将被执行。而执行 rt_schedule() 函数后，当前线程并不一定被换出，即使被换出，也不会被放到就绪线程链表的尾部，而是在系统中选取就绪的优先级最高的线程执行（如果系统中没有比当前线程优先级更高的线程存在，那么执行完 rt_schedule() 函数后，系统将继续执行当前线程）。

3.6 使线程睡眠

在实际应用中，我们有时需要让运行的当前线程延迟一段时间，在指定的时间到达后重新运行，这就叫做 “线程睡眠”。线程睡眠可使用以下三个函数接口：

```
rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);
rt_err_t rt_thread_mdelay(rt_int32_t ms);
```

这三个函数接口的作用相同，调用它们可以使当前线程挂起一段指定的时间，当这个时间过后，线程会被唤醒并再次进入就绪状态。这个函数接受一个参数，该参数指定了线程的休眠时间。线程睡眠接口 rt_thread_sleep/delay/mdelay() 的参数和返回值见下表：

参数	描述
tick/ms	线程睡眠的时间：sleep/delay 的传入参数 tick 以 1 个 OS Tick 为单位；mdelay 的传入参数 ms 以 1ms 为单位；
返回	——
RT_EOK	操作成功

3.7 挂起和恢复线程

当线程调用 rt_thread_delay() 时，线程将主动挂起；当调用 rt_sem_take()，rt_mb_recv() 等函数时，资源不可使用也将导致线程挂起。处于挂起状态的线程，如果其等待的资源超时（超过其设定的等待时间），那么该线程将不再等待这些资源，并返回到就绪状态；或者，当其他线程释放掉该线程所等待的资源时，该线程也会返回到就绪状态。

线程挂起使用下面的函数接口：

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

线程挂起接口 rt_thread_suspend() 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	——
RT_EOK	线程挂起成功
-RT_ERROR	线程挂起失败，因为该线程的状态并不是就绪状态

> 注意：一个线程尝试挂起另一个线程是一个非常危险的行为，因此 RT-Thread 对此函数有严格的使用限制：该函数只能使用来挂起当前线程（即自己挂起自己），不可以在线程 A 中尝试挂起线程 B。而且在挂起线程自己后，需要立刻调用 rt_schedule() 函数进行手动的线程上下文切换。这是因为 A 线程在尝试挂起 B 线程时，A 线程并不清楚 B 线程正在运行什么程序，一旦 B 线程正在使用例如互斥量、信号量等影响、阻塞其他线程（如 C 线程）的内核对象，如果此时其他线程也在等待这个内核对象，那么 A 线程尝试挂起 B 线程的操作将会引发其他线程（如 C 线程）的饥饿，严重危及系统的实时性。

恢复线程就是让挂起的线程重新进入就绪状态，并将线程放入系统的就绪队列中；如果被恢复线程在所有就绪态线程中，位于最高优先级链表的第一位，那么系统将进行线程上下文的切换。线程恢复使用下面的函数接口：

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

线程恢复接口 rt_thread_resume() 的参数和返回值见下表：

参数	描述
thread	线程句柄
返回	——
RT_EOK	线程恢复成功
-RT_ERROR	线程恢复失败，因为该个线程的状态并不是 RT_THREAD_SUSPEND 状态

3.8 控制线程

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下函数接口：

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

线程控制接口 rt_thread_control() 的参数和返回值见下表：

函数参数	描述
thread	线程句柄
cmd	指示控制命令
arg	控制参数
返回	——
RT_EOK	控制执行正确
-RT_ERROR	失败

指示控制命令 cmd 当前支持的命令包括：

- RT_THREAD_CTRL_CHANGE_PRIORITY：动态更改线程的优先级；
- RT_THREAD_CTRL_STARTUP：开始运行一个线程，等同于 rt_thread_startup() 函数调用；
- RT_THREAD_CTRL_CLOSE：关闭一个线程，等同于 rt_thread_delete() 或 rt_thread_detach() 函数调用。

3.9 设置和删除空闲钩子

空闲钩子函数是空闲线程的钩子函数，如果设置了空闲钩子函数，就可以在系统执行空闲线程时，自动执行空闲钩子函数来做一些其他事情，比如系统指示灯。设置 / 删除空闲钩子的接口如下：

```
rt_err_t rt_thread_idle_sethook(void (*hook)(void));
rt_err_t rt_thread_idle_delhook(void (*hook)(void));
```

设置空闲钩子函数 rt_thread_idle_sethook() 的输入参数和返回值如下表所示：

函数参数	描述
hook	设置的钩子函数
返回	——
RT_EOK	设置成功
-RT_EFULL	设置失败

删除空闲钩子函数 rt_thread_idle_delhook() 的输入参数和返回值如下表所示：

函数参数	描述
hook	删除的钩子函数
返回	——
RT_EOK	删除成功
-RT_ENOSYS	删除失败

注意：空闲线程是一个线程状态永远为就绪态的线程，因此设置的钩子函数必须保证空闲线程在任何时刻都不会处于挂起状态，例如 rt_thread_delay()，rt_sem_take() 等可能会导致线程挂起的函数都不能使用。

3.10 设置调度器钩子

在整个系统的运行时，系统都处于线程运行、中断触发 - 响应中断、切换到其他线程，甚至是线程间的切换过程中，或者说系统的上下文切换是系统中最普遍的事件。有时用户可能会想知道在一个时刻发生了什么样的线程切换，可以通过调用下面的函数接口设置一个相应的钩子函数。在系统线程切换时，这个钩子函数将被调用：

```
void rt_scheduler_sethook(void (*hook)(struct rt_thread* from, struct rt_thread* to)
    );
```

设置调度器钩子函数的输入参数如下表所示：

函数参数	描述
hook	表示用户定义的钩子函数指针

钩子函数 hook() 的声明如下：

```
void hook(struct rt_thread* from, struct rt_thread* to);
```

调度器钩子函数 hook() 的输入参数如下表所示：

函数参数	描述
from	表示系统所要切换出的线程控制块指针
to	表示系统所要切换到的线程控制块指针

注意：请仔细编写你的钩子函数，稍有不慎将很可能导致整个系统运行不正常（在这个钩子函数中，基本上不允许调用系统 API，更不应该导致当前运行的上下文挂起）。

4 线程应用示例

下面给出在 Keil 模拟器环境下的应用示例。

4.1 创建线程示例

这个例子创建一个动态线程初始化一个静态线程，一个线程在运行完毕后自动被系统删除，另一个线程一直打印计数，如下代码：

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

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

static rt_thread_t tid1 = RT_NULL;

/* 线程 1 的入口函数 */
static void thread1_entry(void *parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程 1 采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread1 count: %d\n", count ++);
        rt_thread_mdelay(500);
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
/* 线程 2 入口 */
static void thread2_entry(void *param)
{
    rt_uint32_t count = 0;

    /* 线程 2 拥有较高的优先级，以抢占线程 1 而获得执行 */
    for (count = 0; count < 10 ; count++)
    {
        /* 线程 2 打印计数值 */
        rt_kprintf("thread2 count: %d\n", count);
    }
    rt_kprintf("thread2 exit\n");
    /* 线程 2 运行结束后也将自动被系统脱离 */
}

/* 线程示例 */
int thread_sample(void)
{
    /* 创建线程 1，名称是 thread1，入口是 thread1_entry*/
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

    /* 如果获得线程控制块，启动这个线程 */
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
```

```
    /* 初始化线程 2，名称是 thread2，入口是 thread2_entry */
    rt_thread_init(&thread2,
                   "thread2",
                   thread2_entry,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(thread_sample, thread sample);
```

仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 24 2018
 2006 - 2018 Copyright by rt-thread team
msh >thread_sample
msh >thread2 count: 0
thread2 count: 1
thread2 count: 2
thread2 count: 3
thread2 count: 4
thread2 count: 5
thread2 count: 6
thread2 count: 7
thread2 count: 8
thread2 count: 9
thread2 exit
thread1 count: 0
thread1 count: 1
thread1 count: 2
thread1 count: 3
...
```

线程 2 计数到一定值会执行完毕，线程 2 被系统自动删除，计数停止。线程 1 一直打印计数。

注意：关于删除线程：大多数线程是循环执行的，无需删除；而能运行完毕的线程，RT-Thread 在线程运行完毕后，自动删除线程，在 rt_thread_exit() 里完成删除动作。用户只需要了解该接口的作用，不推荐使用该接口（可以由其他线程调用此接口或在定时器超时函数中调用此接口删除一个线程，但是这种使用非常少）。

4.2 线程时间片轮转调度示例

这个例子创建两个线程，在执行时会一直打印计数，如下代码：

```
#include <rtthread.h>

#define THREAD_STACK_SIZE   1024
#define THREAD_PRIORITY     20
#define THREAD_TIMESLICE    10

/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;
    rt_uint32_t count = 0;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        if(0 == (count % 5))
        {
            rt_kprintf("thread %d is running ,thread %d count = %d\n", value , value
                 , count);

            if(count> 200)
                return;
        }
         count++;
     }
}

int timeslice_sample(void)
{
    rt_thread_t tid = RT_NULL;
    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                           thread_entry, (void*)1,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid != RT_NULL)
        rt_thread_startup(tid);


    /* 创建线程 2 */
    tid = rt_thread_create("thread2",
                           thread_entry, (void*)2,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE-5);
    if (tid != RT_NULL)
```

```
        rt_thread_startup(tid);
    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(timeslice_sample, timeslice sample);
```

仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 27 2018
 2006 - 2018 Copyright by rt-thread team
msh >timeslice_sample
msh >thread 1 is running ,thread 1 count = 0
thread 1 is running ,thread 1 count = 5
thread 1 is running ,thread 1 count = 10
thread 1 is running ,thread 1 count = 15
...
thread 1 is running ,thread 1 count = 125
thread 1 is rthread 2 is running ,thread 2 count = 0
thread 2 is running ,thread 2 count = 5
thread 2 is running ,thread 2 count = 10
thread 2 is running ,thread 2 count = 15
thread 2 is running ,thread 2 count = 20
thread 2 is running ,thread 2 count = 25
thread 2 is running ,thread 2 count = 30
thread 2 is running ,thread 2 count = 35
thread 2 is running ,thread 2 count = 40
thread 2 is running ,thread 2 count = 45
thread 2 is running ,thread 2 count = 50
thread 2 is running ,thread 2 count = 55
thread 2 is running ,thread 2 count = 60
thread 2 is running ,thread 2 cunning ,thread 2 count = 65
thread 1 is running ,thread 1 count = 135
...
thread 2 is running ,thread 2 count = 205
```

由运行的计数结果可以看出，线程 2 的运行时间是线程 1 的一半。

4.3 线程调度器钩子示例

在线程进行调度切换时，会执行调度，我们可以设置一个调度器钩子，这样可以在线程切换时，做一些额外的事情，这个例子是在调度器钩子函数中打印线程间的切换信息，如下代码：

```
#include <rtthread.h>

#define THREAD_STACK_SIZE   1024
#define THREAD_PRIORITY     20
```

```
#define THREAD_TIMESLICE    10

/* 针对每个线程的计数器 */
volatile rt_uint32_t count[2];

/* 线程 1、2 共用一个入口，但入口参数不同 */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        rt_kprintf("thread %d is running\n", value);
        rt_thread_mdelay(1000); // 延时一段时间
    }
}

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

static void hook_of_scheduler(struct rt_thread* from, struct rt_thread* to)
{
    rt_kprintf("from: %s -->  to: %s \n", from->name , to->name);
}

int scheduler_hook(void)
{
    /* 设置调度器钩子 */
    rt_scheduler_sethook(hook_of_scheduler);

    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                            thread_entry, (void*)1,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 创建线程 2 */
    tid2 = rt_thread_create("thread2",
                            thread_entry, (void*)2,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY,THREAD_TIMESLICE - 5);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    return 0;
}
```

```
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(scheduler_hook, scheduler_hook sample);
```

仿真运行结果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Aug 27 2018
 2006 - 2018 Copyright by rt-thread team
msh > scheduler_hook
msh >from: tshell -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle
from: tidle -->  to: thread1
thread 1 is running
from: thread1 -->  to: tidle
from: tidle -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle
...
```

由仿真的结果可以看出，对线程进行切换时，设置的调度器钩子函数是在正常工作的，一直在打印线程切换的信息，包含切换到空闲线程。