
RT-THREAD 星火 1 号内核学习手册

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 ©2023



WWW.RT-THREAD.ORG

Tuesday 1st August, 2023

版本和修订

Date	Version	Author	Note
2023-07-25	v0.1.0	RT-Thread	内核初始版本

目录

版本和修订	i
目录	ii
1 内核入门	1
1.1 准备工作	1
1.1.1 RT-Thread Studio 环境搭建	2
1.1.2 工程目录介绍	5
1.1.3 FinSH 命令行中启动线程	5
1.2 内核入门实践例程	6
2 认识 RT-Thread main()	7
2.1 启动入口 rtthread_startup()	7
2.2 创建 main 线程 rt_application_init()	8
2.3 用户代码入口 main()	9
3 导出一个 shell 命令	11
3.1 背景	11
3.1.1 FinSH 简介	11
3.1.2 内置命令	13
3.2 MSH_CMD_EXPORT()	15
3.3 导出命令应用示例	16
3.3.1 不带参数的命令	16
3.3.2 带参数的命令	17
3.4 参考资料	18
4 RT-Thread 自动初始化机制	19
4.1 自动初始化示例	20
4.2 编译运行	20

5	线程的使用	22
5.1	代码设计	22
5.2	线程使用示例	23
5.3	编译运行	25
6	线程的时间片轮转调度	26
6.1	代码设计	26
6.2	时间片示例	27
6.3	编译运行	28
7	调度器	30
7.1	设置调度器钩子	30
7.2	调度器钩子示例	31
7.3	编译运行	31
8	空闲线程	33
8.1	空闲线程钩子函数	33
8.2	代码设计	34
8.3	空闲钩子函数使用示例	34
8.4	编译运行	35
9	临界区	37
9.1	全局开关中断	37
9.2	进入退出临界区	39
9.3	代码设计	39
9.4	临界区保护示例	39
9.5	编译运行	41
10	定时器的使用	42
10.1	代码设计	42
10.2	定时器使用示例	43
10.3	编译运行	45
11	信号量—生产者消费者问题	46
11.1	代码设计	46
11.2	生产者消费者示例	48
11.3	编译运行	51

12 互斥量——优先级继承	52
12.1 代码设计	52
12.2 源程序说明	52
12.3 编译运行	55
13 事件集的使用	56
13.1 代码设计	56
13.2 事件集使用示例	57
13.3 编译运行	59
14 邮箱的使用	61
14.1 代码设计	61
14.1.1 源程序说明	62
14.2 编译运行	65
15 消息队列的使用	66
15.1 代码设计	66
15.2 消息队列使用示例	67
15.3 编译运行	70
16 动态内存堆的使用	72
16.1 代码设计	72
16.2 动态内存堆示例	72
16.3 编译运行	74
17 内存池的使用	75
17.1 代码设计	75
17.2 mempool 使用示例	75
17.3 编译运行	77

第 1 章

内核入门

在准备工作之前，需要了解一些内核的基础知识，详见 [《RT-Thread 内核基础.pdf》](#)

注意：由于 RT-Thread 在不断更新中，且本文档针对 RT-Thread V4.1.1，所以某些细节可能与更高的发布版本有出入，在学习时请使用相应 V4.1.1 版本进行学习。

1.1 准备工作

准备工作的目的是让初学者了解 RT-Thread 运行环境，将以 RT-Thread Studio 为例，搭建 RT-Thread 运行环境。我们使用的硬件是星火 1 号开发板。

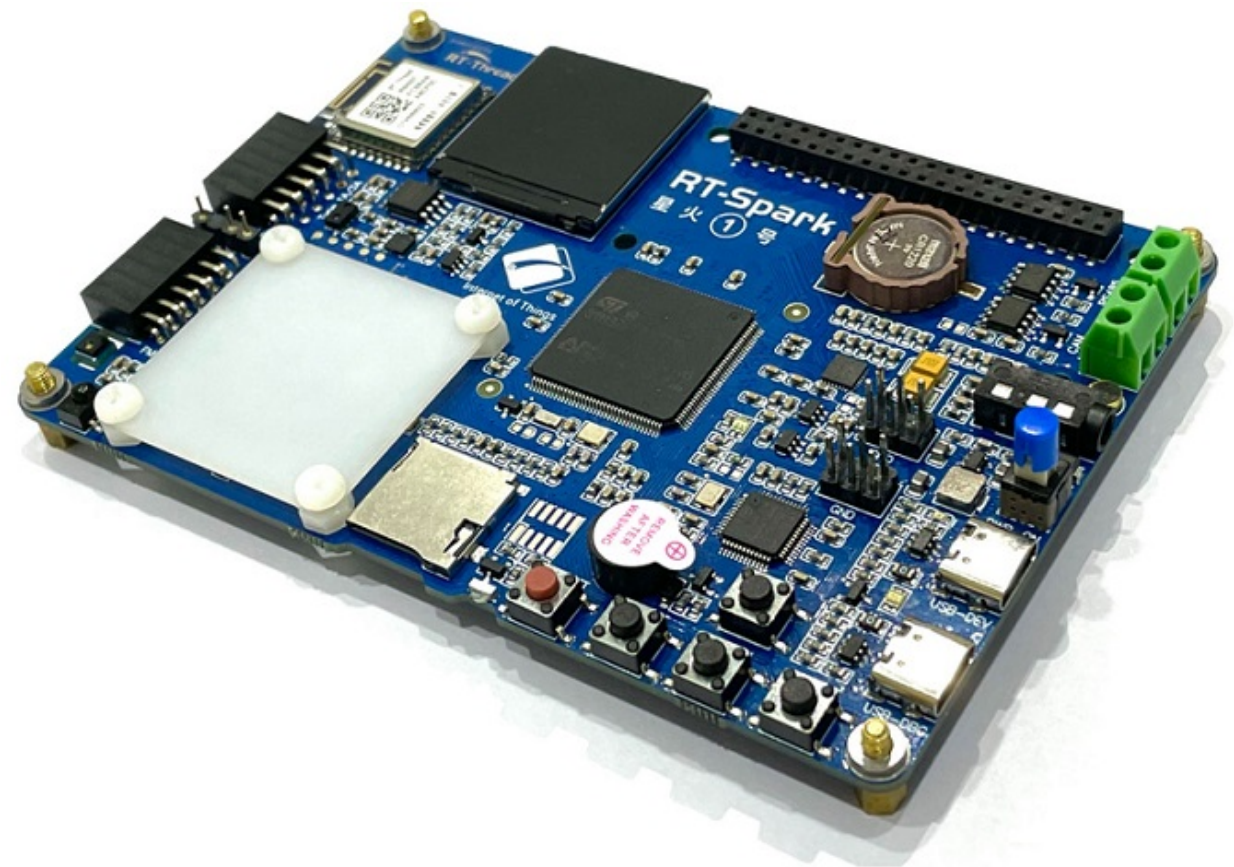


图 1.1: 星火 1 号开发板

1.1.1 RT-Thread Studio 环境搭建

- 1. 点击链接安装 RT-Thread Studio
- 2. 在 RT-Thread Studio 的 SDK 管理器中下载资开发板源包

RT-Thread Source Code		
4.1.1 (2022-08-22)	341 MB	Installed
4.1.0 (2022-04-01)	141 MB	Not installed
4.0.5 (2022-01-10)	122 MB	Not installed
4.0.4 (2021-10-28)	115 MB	Not installed
4.0.3 (2021-03-18)	104 MB	Installed
4.0.2 (2019-12-20)	65 MB	Installed

图 1.2: 资源下载











>	<input type="checkbox"/>  STM32L496-NUCLEO	<input type="radio"/> Not installed
>	<input type="checkbox"/>  STM32WB55-ST-NUCLEO	<input type="radio"/> Not installed
>	<input type="checkbox"/>  STM32WL55-ST-NUCLEO	<input type="radio"/> Not installed
>	<input type="checkbox"/>  STM32G474-ST-NUCLEO	<input type="radio"/> Not installed
>	<input type="checkbox"/>  STM32F103-100ASK-PRO	<input type="radio"/> Not installed
>	<input type="checkbox"/>  STM32F407-RT-SPARK	<input checked="" type="radio"/> Installed
▼	<input type="checkbox"/>  Synwit	
>	<input type="checkbox"/>  SWDM_LQ64_32SRE04	<input type="radio"/> Not installed
>	<input type="checkbox"/>  SWM320VET7-SYNWIT-SV	<input type="radio"/> Not installed
>	<input type="checkbox"/>  SWDM-QFP100-34SVEA1	<input type="radio"/> Not installed

图 1.3: 资源下载

3. 新建工程（新建工程可以基于模板工程，也可以基于示例工程），内核部分基于示例 01_kernel 完成工程的创建，

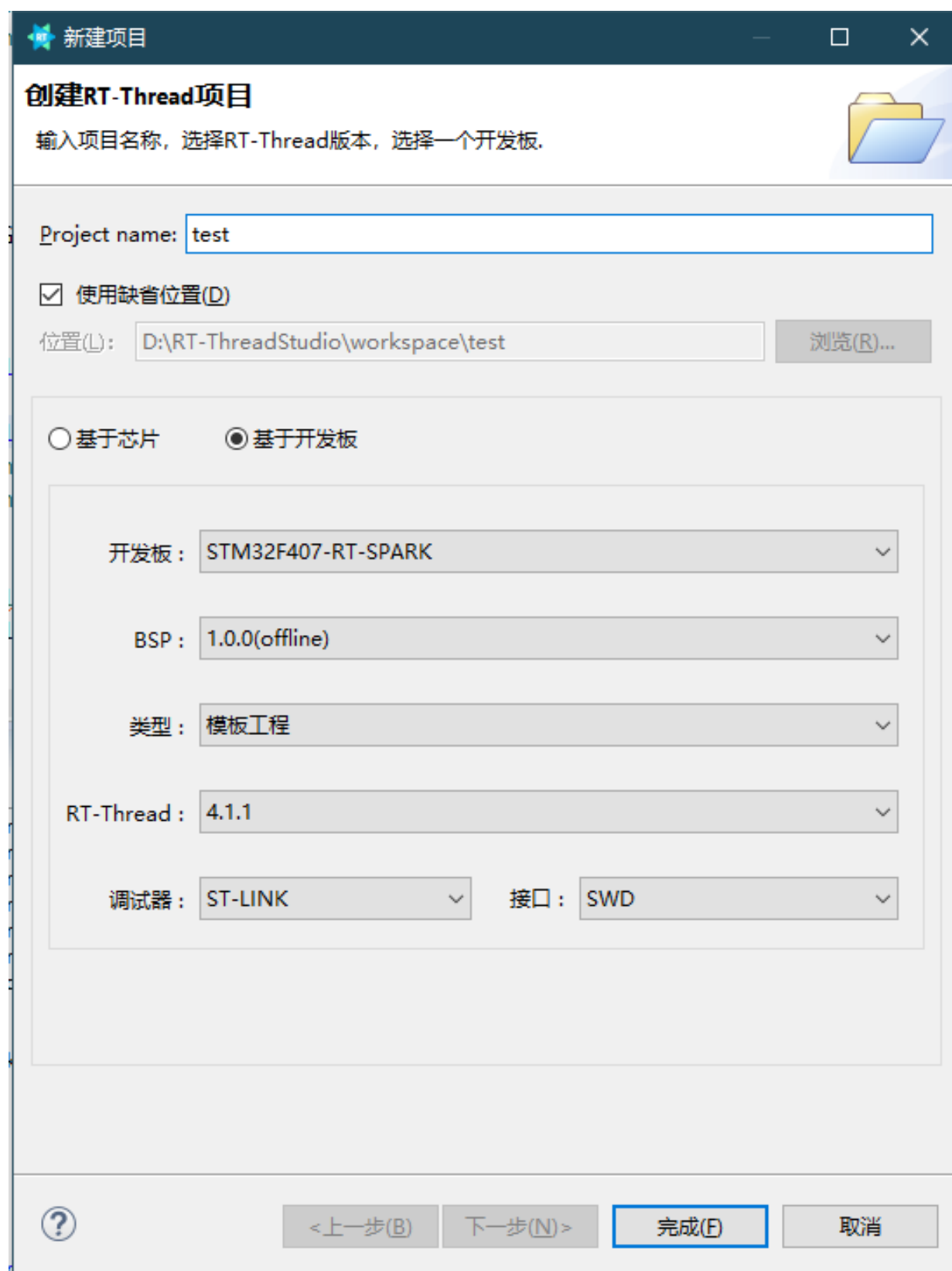


图 1.4: 创建项目

4. 创建成功之后编译
5. 编译无误, 连接开发板, 下载代码

1.1.2 工程目录介绍

在 RT-Thread Studio 中新建好工程后，可以看到工程中存在以下目录，分别是：

- **rt-thread Settings:** 配置文件
- **applications:** 应用代码，内核部分是一些用于学习的示例代码
- **board:** 板级相关的代码
- **libraries:** STM32F4 固件库、通用外接驱动程序
- **rt-thread:** rt-thread 源码

初学者可以将应用代码放在 **applications** 目录下。

1.1.3 FinSH 命令行中启动线程

RT-Thread 提供 FinSH 功能，用于调试或查看系统信息，**msh** 表示 FinSH 处于一种传统命令行模式，此模式下可以使用类似于 **dos/bash** 等传统的 **shell** 命令。

比如，我们可以通过输入“**help** + 回车”或者直接按下 **Tab** 键，输出当前系统所支持的所有命令，如下：

```
msh >help
RT-Thread shell commands:
clear          - clear the terminal screen
version        - show RT-Thread version information
list_thread    - list thread
list_sem       - list semaphore in system
list_event     - list event in system
list_mutex     - list mutex in system
list_mailbox   - list mail box in system
list_msgqueue  - list message queue in system
list_mempool   - list memory pool in system
list_timer     - list timer in system
list_device    - list device in system
list           - list objects
help           - RT-Thread shell help.
ps             - List threads in the system.
free           - Show the memory usage in the system.
pin            - pin [option]
dynmem_sample  - dynmem sample
event_sample   - event sample
idle_hook_sample - idle hook sample
interrupt_sample - interrupt sample
mailbox_sample - mailbox sample
mempool_sample - mempool sample
msgq_sample    - msgq sample
mutex_sample   - mutex sample
pri_inversion  - pri_inversion sample
producer_consume - producer_consumer sample
scheduler_hook - scheduler_hook sample
semaphore_sample - semaphore sample
```

```
thread_sample    - thread sample
timer_sample     - timer sample
timeslice_sample - timeslice sample
reboot           - Reboot System
msh >
```

此时可以输入列表中的命令，如输入 `list_thread` 命令显示系统当前正在运行的线程，结果显示为 `tshell`（shell 线程）线程与 `tidle`（空闲线程）线程：

```
msh >
msh >list_thread
thread  pri  status      sp      stack size max used left tick  error
-----  --  -
tshell   20  running 0x000000c8 0x00001000    14%  0x00000004 OK
tidle0   31  ready  0x0000005c 0x00000400    16%  0x0000001c OK
msh >
```

FinSH 具有命令自动补全功能，输入命令的部分字符（前几个字母，注意区分大小写），按下 **Tab** 键，则系统会根据当前已输入的字符，从系统中查找已经注册好的相关命令，如果找到与输入相关的命令，则会将完整的命令显示在终端上。

如：要使用 `version` 命令，可以先输入“`v`”，再按下 **Tab** 键，可以发现系统会在下方补全了有关“`v`”开头的命令：`version`，此时只需要回车，即可查看该命令的执行结果。

1.2 内核入门实践例程

有了以上的基础后，开始学习内核，以下是内核入门的实践教程，详见文件夹 `kernel_docs`，内核学习的内容如下：

1. 认识 RT-Thread `main()`
2. 导出一个 `shell` 命令
3. 自动初始化体验
4. 创建一个线程（静态、动态）
5. 线程时间片的轮转调度
6. 调度器钩子
7. 空闲线程钩子
8. 临界区
9. 认识系统定时器
10. 认识信号量：生产者消费者问题
11. 认识互斥量：优先级继承
12. 认识事件集
13. 认识邮箱
14. 认识消息队列
15. 使用动态内存堆
16. 使用动态内存池

第 2 章

认识 RT-Thread main()

main() 函数在 RT-Thread 中是 main 线程调用的函数，一般也称为 main 线程。

2.1 启动入口 rtthread_startup()

main 线程是在 rt-thread 启动时创建的，rt-thread 启动入口 rtthread_startup() 函数如下所示：

```
int rtthread_startup(void)
{
    rt_hw_interrupt_disable();

    /* 板级初始化 */
    rt_hw_board_init();

    /* 显示 RT-Thread 版本号 */
    rt_show_version();

    /* 调度器初始化 */
    rt_system_scheduler_init();

#ifdef RT_USING_SIGNALS
    /* 信号初始化 */
    rt_system_signal_init();
#endif /* RT_USING_SIGNALS */

    /* main 线程初始化（即创建 main 线程） */
    rt_application_init();

    /* timer 定时器线程初始化 */
    rt_system_timer_thread_init();

    /* idle 空闲线程初始化 */
    rt_thread_idle_init();
```

```

#ifdef RT_USING_SMP
    rt_hw_spin_lock(&cpus_lock);
#endif /* RT_USING_SMP */

    /* 启动调度器 */
    rt_system_scheduler_start();

    /* never reach here */
    return 0;
}

```

这部分启动代码，大致可以分为四个部分：

1. 初始化与系统相关的硬件；
2. 初始化系统内核对象，例如定时器、调度器、信号；
3. 创建 main 线程，在 main 线程中对各类模块依次进行初始化；
4. 初始化定时器线程、空闲线程，并启动调度器。

启动调度器之前，系统所创建的线程在执行 `rt_thread_startup()` 后并不会立马运行，它们会处于就绪状态等待系统调度；待启动调度器之后，系统才转入第一个线程开始运行，根据调度规则，选择的是就绪队列中优先级最高的线程。

`rt_hw_board_init()` 中完成系统时钟设置，为系统提供心跳、串口初始化，将系统输入输出终端绑定到这个串口，后续系统运行信息就会从串口打印出来。

2.2 创建 main 线程 rt_application_init()

我们将启动入口中调用的 `rt_application_init()` 函数展开，可以看到 main 线程到 `main()` 函数：

```

void rt_application_init(void)
{
    rt_thread_t tid;

    /* 动态或静态创建 main 线程，入口函数是 main_thread_entry */
#ifdef RT_USING_HEAP
    tid = rt_thread_create("main", main_thread_entry, RT_NULL,
                           RT_MAIN_THREAD_STACK_SIZE, RT_MAIN_THREAD_PRIORITY, 20);
    RT_ASSERT(tid != RT_NULL);
#else
    rt_err_t result;

    tid = &main_thread;
    result = rt_thread_init(tid, "main", main_thread_entry, RT_NULL,
                            main_stack, sizeof(main_stack), RT_MAIN_THREAD_PRIORITY,
                            20);
    RT_ASSERT(result == RT_EOK);

    /* if not define RT_USING_HEAP, using to eliminate the warning */

```

```

    (void)result;
#endif /* RT_USING_HEAP */

    /* 启动 main 线程 */
    rt_thread_startup(tid);
}

/* main 线程入口函数: main_thread_entry */
void main_thread_entry(void *parameter)
{
    extern int main(void);

#ifdef RT_USING_COMPONENTS_INIT
    rt_components_init();
#endif /* RT_USING_COMPONENTS_INIT */

    /* main 线程在不同平台下调用 main 函数 */
#ifdef __ARMCC_VERSION
    {
        extern int $Super$$main(void);
        $Super$$main(); /* for ARMCC. */
    }
#elif defined(__ICCARM__) || defined(__GNUC__) || defined(__TASKING__)
    main();
#endif
}

```

这样就调用到了 main.c 文件中的 main() 函数了，

2.3 用户代码入口 main()

main() 函数是 RT-Thread 的用户代码入口，用户可以在 main() 函数里添加自己的应用。打开基于 01_kernel 新建的工程，双击 applications 文件夹下的 main.c 文件，添加简单应用，如使用 rt_kprintf 进行打印。

```

int main(void)
{
    rt_kprintf("Hello RT-Thread!\n"); /* 新增本行打印 */
    return 0;
}

```

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志以及新增的打印“Hello RT-Thread!”：

```
\ | /  
- RT -   Thread Operating System  
/ | \    4.1.1 build Jul 17 2023 11:17:06  
2006 - 2022 Copyright by RT-Thread team  
Hello RT-Thread!  
msh >
```

图 2.1: 打印

第 3 章

导出一个 shell 命令

3.1 背景

在计算机发展的早期，图形系统出现之前，没有鼠标，甚至没有键盘。那时候人们如何与计算机交互呢？最早期的计算机使用打孔的纸条向计算机输入命令，编写程序。后来随着计算机的不断发展，显示器、键盘成为计算机的标准配置，但此时的操作系统还不支持图形界面，计算机先驱们开发了一种软件，它接受用户输入的命令，解释之后，传递给操作系统，并将操作系统执行的结果返回给用户。这个程序像一层外壳包裹在操作系统的外面，所以它被称为 **shell**。

嵌入式设备通常需要将开发板与 PC 机连接起来通讯，常见连接方式包括：串口、USB、以太网、Wi-Fi 等。一个灵活的 **shell** 也应该支持在多种连接方式上工作。有了 **shell**，就像在开发者和计算机之间架起了一座沟通的桥梁，开发者能很方便的获取系统的运行情况，并通过命令控制系统的运行。特别是在调试阶段，有了 **shell**，开发者除了能更快的定位到问题之外，也能利用 **shell** 调用测试函数，改变测试函数的参数，减少代码的烧录次数，缩短项目的开发时间。

FinSH 是 RT-Thread 的命令行组件（**shell**），正是基于上面这些考虑而诞生的，FinSH 的发音为 [ˈfɪnʃ]。读完本章，我们会对 FinSH 的工作方式以及如何导出自己的命令到 FinSH 有更加深入的了解。

3.1.1 FinSH 简介

FinSH 是 RT-Thread 的命令行组件，提供一套供用户在命令行调用的操作接口，主要用于调试或查看系统信息。它可以使用串口 / 以太网 / USB 等与 PC 机进行通信，硬件拓扑结构如下图所示：

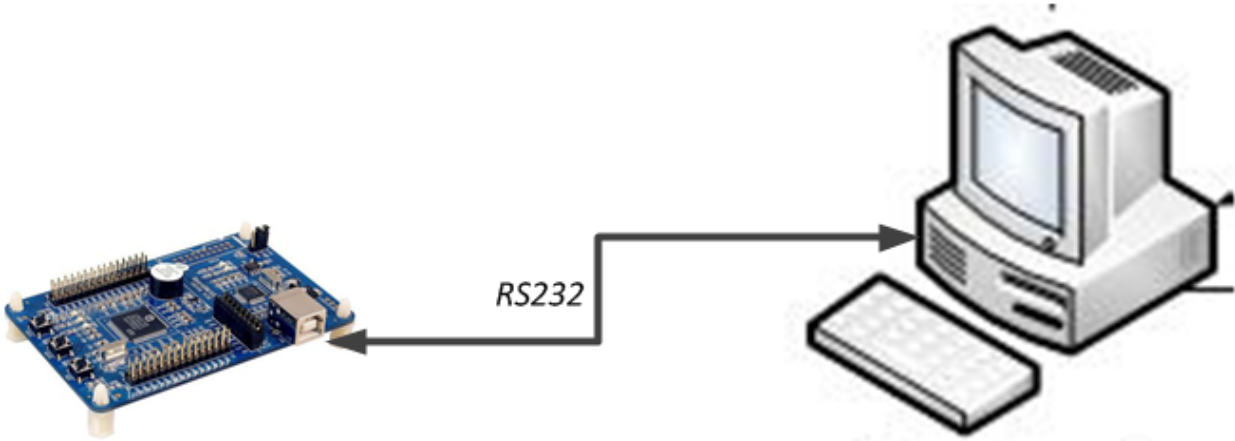


图 3.1: FinSH 硬件连接图

用户在控制终端输入命令，控制终端通过串口、USB、网络等方式将命令传给设备里的 FinSH，FinSH 会读取设备输入命令，解析并自动扫描内部函数表，寻找对应函数名，执行函数后输出回应，回应通过原路返回，将结果显示在控制终端上。

当使用串口连接设备与控制终端时，FinSH 命令的执行流程，如下图所示：

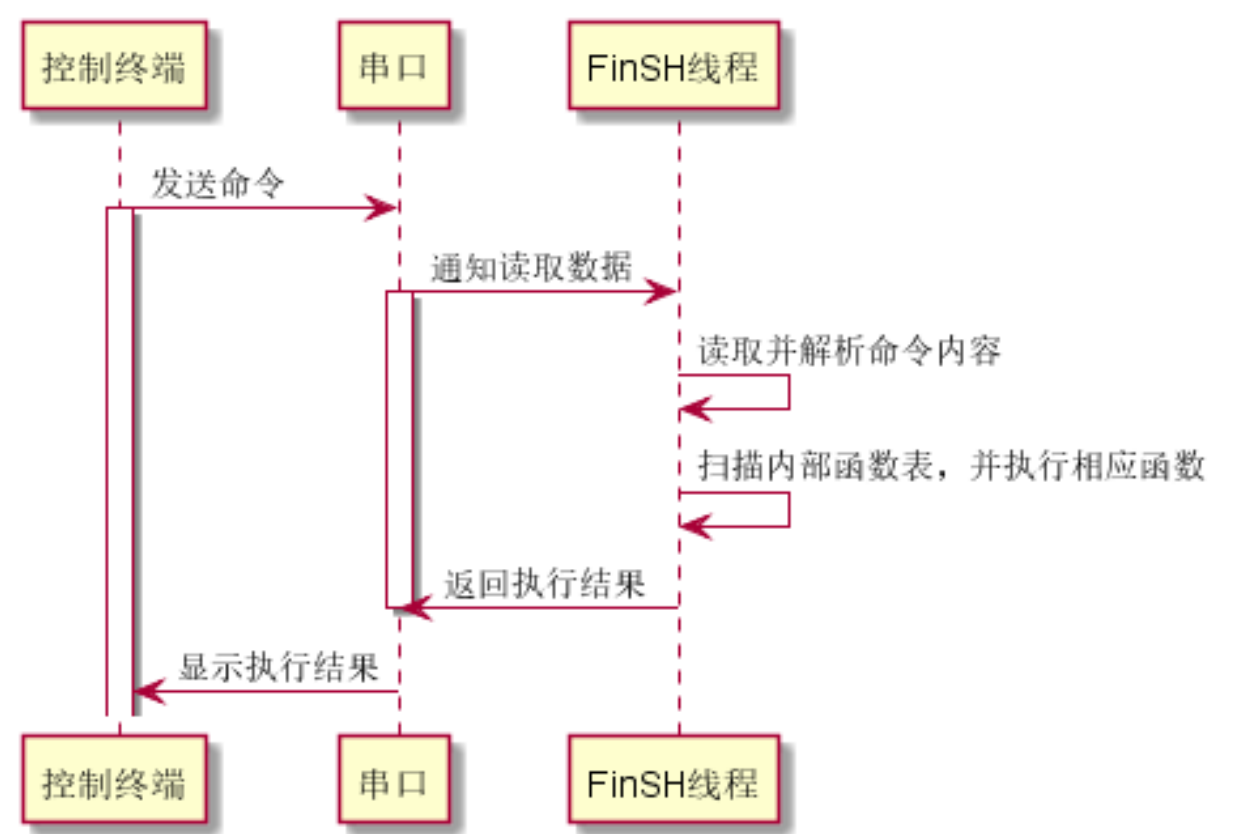


图 3.2: FinSH 命令执行流程图

FinSH 支持权限验证功能，系统在启动后会进行权限验证，只有权限验证通过，才会开启 FinSH 功能，提升系统输入的安全性。

FinSH 支持自动补全、查看历史命令等功能，通过键盘上的按键可以很方便的使用这些功能，FinSH 支持的按键如下表所示：

按键	功能描述
Tab 键	当没有输入任何字符时按下 Tab 键将会打印当前系统支持的所有命令。若已经输入部分字符时按下 Tab 键，将会查找匹配的命令，也会按照文件系统的当前目录下的文件名进行补全，并可以继续输入，多次补全
⏮ 键	上下翻阅最近输入的历史命令
退格键	删除符
⏮ 键	向左或向右移动标

FinSH 支持命令行模式，此模式又称为 **msh(module shell)**，**msh** 模式下，FinSH 与传统 **shell**（**dos/bash**）执行方式一致，例如，可以通过 **cd /** 命令将目录切换至根目录。

msh 通过解析，将输入字符分解成以空格区分的命令和参数。其命令执行格式如下所示：

```
command [arg1] [arg2] [...]
```

其中 **command** 既可以是 RT-Thread 内置的命令，也可以是可执行的文件。

3.1.2 内置命令

当下载好代码之后，我们在终端键入 **help** 并回车，可以查询系统中的命令，如下

```

\ | /
- RT -   Thread Operating System
/ | \   4.1.1 build Jul 17 2023 13:46:06
2006 - 2022 Copyright by RT-Thread team

msh >
msh >help
RT-Thread shell commands:
clear          - clear the terminal screen
version        - show RT-Thread version information
list_thread    - list thread
list_sem       - list semaphore in system
list_event     - list event in system
list_mutex     - list mutex in system
list_mailbox   - list mail box in system
list_msgqueue  - list message queue in system
list_mempool   - list memory pool in system
list_timer     - list timer in system
list_device    - list device in system
list           - list objects
help           - RT-Thread shell help.
ps             - List threads in the system.
free           - Show the memory usage in the system.
pin            - pin [option]
pwm            - pwm [option]
reboot         - Reboot System

msh >

```

图 3.3: 系统命令

以上命令可以输入至终端中查看相应的信息，如：

```

msh >list_thread
thread  pri  status      sp      stack size max used left tick error
-----
tshell   20  running 0x00000270 0x00001000 15% 0x00000003 OK
tidle0   31  ready  0x00000060 0x00000400 14% 0x00000001 OK
main     10  suspend 0x000000bc 0x00000800 17% 0x00000013 OK
msh >
msh >list_sem
semaphor v suspend thread
-----
shrx      000 0
msh >

```

图 3.4: 输入系统命令

`list_thread` 返回字段的描述:

字段	描述
<code>thread</code>	线程的名称
<code>pri</code>	线程的优先级
<code>status</code>	线程当前的状态
<code>sp</code>	线程当前的栈位置
<code>stack size</code>	线程的栈大小
<code>max used</code>	线程历史中使用的最大栈位置
<code>left tick</code>	线程剩余的运行节拍数
<code>error</code>	线程的错误码

`list_sem` 返回字段的描述:

字段	描述
<code>semaphore</code>	信号量的名称
<code>v</code>	信号量当前的值
<code>suspend thread</code>	等待这个信号量的线程数目

3.2 MSH_CMD_EXPORT()

RT-Thread 使用 `MSH_CMD_EXPORT()` 将命令导出到 `msh` 命令列表中:

```
MSH_CMD_EXPORT(name, desc);
```

参数	描述
<code>name</code>	要导出的命令
<code>desc</code>	导出命令的描述

这个命令可以导出有参数的命令，也可以导出无参数的命令。

导出无参数命令时，函数的入参为 `void`，示例如下：

```
void hello(void)
{
    rt_kprintf("hello RT-Thread!\n");
}

MSH_CMD_EXPORT(hello , say hello to RT-Thread);
```

导出有参数的命令时，函数的入参为 `int argc` 和 `char**argv`。`argc` 表示参数的个数，`argv` 表示命令行参数字符串指针数组指针。导出有参数命令示例如下：

```
static void atcmd(int argc, char**argv)
{
    ....
}

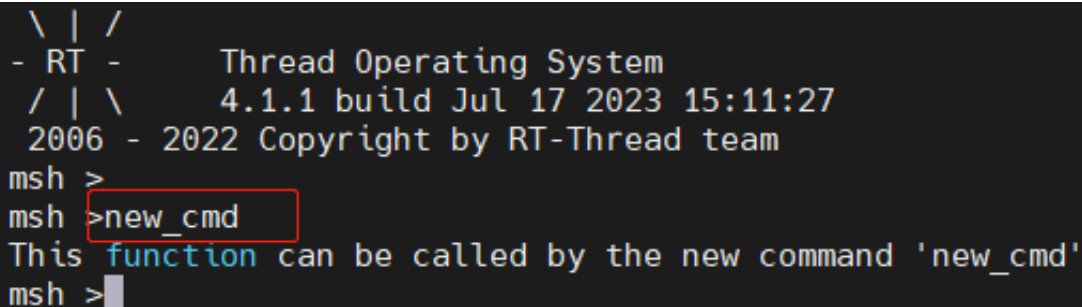
MSH_CMD_EXPORT(atcmd, atcmd sample: atcmd <server|client>);
```

例如：我们新增一个命令 `new_cmd`

```
int new_cmd(void)
{
    rt_kprintf("This function can be called by the new system command 'new_cmd'\n");
    return 0;
}

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

并在终端中执行该命令



```
\ | /
- RT -   Thread Operating System
/ | \   4.1.1 build Jul 17 2023 15:11:27
2006 - 2022 Copyright by RT-Thread team
msh >
msh >new_cmd
This function can be called by the new command 'new_cmd'
msh >
```

图 3.5: 新增命令

3.3 导出命令应用示例

以下代码示例可自行体验，代码详见 `shell_cmd_sample.c`。

3.3.1 不带参数的命令

这里将演示如何将一个自定义的命令导出，示例代码如下所示，代码中创建了 `hello` 函数，然后通过 `MSH_CMD_EXPORT` 命令即可将 `hello` 函数导出到 `FinSH` 命令列表中。

```
#include <rtthread.h>

void hello(void)
```

```
{
    rt_kprintf("hello RT-Thread!\n");
}

MSH_CMD_EXPORT(hello , say hello to RT-Thread);
```

系统运行起来后，在 FinSH 控制台按 **tab** 键可以看到导出的命令：

```
msh />
RT-Thread shell commands:
hello          - say hello to RT-Thread
version        - show RT-Thread version information
... ..
```

运行 **hello** 命令，运行结果如下所示：

```
msh />hello
hello RT_Thread!
msh />
```

3.3.2 带参数的命令

这里将演示如何将一个带参数的自定义的命令导出到 FinSH 中, 示例代码如下所示，代码中创建了 **atcmd()** 函数，然后通过 **MSH_CMD_EXPORT** 命令即可将 **atcmd()** 函数导出到命令列表中。

```
#include <rtthread.h>

static void atcmd(int argc, char**argv)
{
    if (argc < 2)
    {
        rt_kprintf("Please input'atcmd <server|client>'\n");
        return;
    }

    if (!rt_strcmp(argv[1], "server"))
    {
        rt_kprintf("AT server!\n");
    }
    else if (!rt_strcmp(argv[1], "client"))
    {
        rt_kprintf("AT client!\n");
    }
    else
    {
        rt_kprintf("Please input'atcmd <server|client>'\n");
    }
}
```

```
MSH_CMD_EXPORT(atcmd, atcmd sample: atcmd <server|client>);
```

系统运行起来后，在 FinSH 控制台按 **tab** 键可以看到导出的命令：

```
msh />
RT-Thread shell commands:
hello          - say hello to RT-Thread
atcmd          - atcmd sample: atcmd <server|client>
version        - show RT-Thread version information
... ..
```

运行 **atcmd** 命令，运行结果如下所示：

```
msh />atcmd
Please input 'atcmd <server|client>'
msh />
```

运行 **atcmd server** 命令，运行结果如下所示：

```
msh />atcmd server
AT server!
msh />
```

运行 **atcmd client** 命令，运行结果如下所示：

```
msh />atcmd client
AT client!
msh />
```

3.4 参考资料

FinSH 组件：[FinSH 控制台](#)

第 4 章

RT-Thread 自动初始化机制

自动初始化机制是指初始化函数不需要被显式调用，只需要在函数定义处通过宏定义的方式进行申明，就会在系统启动过程中被执行。

例如在串口驱动中调用一个宏定义告知系统初始化需要调用的函数，代码如下：

```
int rt_hw_usart_init(void) /* 串口初始化函数 */
{
    ... ..
    /* 注册串口 1 设备 */
    rt_hw_serial_register(&serial1, "uart1",
                          RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX,
                          uart);

    return 0;
}
INIT_BOARD_EXPORT(rt_hw_usart_init); /* 使用组件自动初始化机制 */
```

示例代码最后的 `INIT_BOARD_EXPORT(rt_hw_usart_init)` 表示使用自动初始化功能，按照这种方式，`rt_hw_usart_init()` 函数就会被系统自动调用，那么它是在哪里被调用的呢？

在系统启动流程图中，有两个函数：`rt_components_board_init()` 与 `rt_components_init()`，其后的带底色方框内部的函数表示被自动初始化的函数，其中：

1. “board init functions” 为所有通过 `INIT_BOARD_EXPORT(fn)` 申明的初始化函数。
2. “pre-initialization functions” 为所有通过 `INIT_PREV_EXPORT(fn)` 申明的初始化函数。
3. “device init functions” 为所有通过 `INIT_DEVICE_EXPORT(fn)` 申明的初始化函数。
4. “components init functions” 为所有通过 `INIT_COMPONENT_EXPORT(fn)` 申明的初始化函数。
5. “enviroment init functions” 为所有通过 `INIT_ENV_EXPORT(fn)` 申明的初始化函数。
6. “application init functions” 为所有通过 `INIT_APP_EXPORT(fn)` 申明的初始化函数。

`rt_components_board_init()` 函数执行的比较早，主要初始化相关硬件环境，执行这个函数时将会遍历通过 `INIT_BOARD_EXPORT(fn)` 申明的初始化函数表，并调用各个函数。

`rt_components_init()` 函数会在操作系统运行起来之后创建的 `main` 线程里被调用执行，这个时候硬件环境和操作系统已经初始化完成，可以执行应用相关代码。`rt_components_init()` 函数会遍历通过剩下的其他几个宏申明的初始化函数表。

RT-Thread 的自动初始化机制使用了自定义 RTI 符号段，将需要在启动时进行初始化的函数指针放到了该段中，形成一张初始化函数表，在系统启动过程中会遍历该表，并调用表中的函数，达到自动初始化的目的。

用来实现自动初始化功能的宏接口定义详细描述如下表所示：

初始化顺	宏接口	描述
1	INIT_BOARD_EXPORT(fn)	非常早期的初始化，此时调度器还未启动
2	INIT_PREV_EXPORT(fn)	主要是用于纯软件的初始化、没有太多依赖的函数
3	INIT_DEVICE_EXPORT(fn)	外设驱动初始化相关，比如网卡设备
4	INIT_COMPONENT_EXPORT(fn)	组件初始化，比如文件系统或者 LWIP
5	INIT_ENV_EXPORT(fn)	系统环境初始化，比如挂载文件系统
6	INIT_APP_EXPORT(fn)	应用初始化，比如 GUI 应用

初始化函数主动通过这些宏接口进行申明，如 INIT_BOARD_EXPORT(rt_hw_usart_init)，链接器会自动收集所有被申明的初始化函数，放到 RTI 符号段中，该符号段位于内存分布的 RO 段中，该 RTI 符号段中的所有函数在系统初始化时会被自动调用。

4.1 自动初始化示例

代码详见：auto_init_sample.c

使用自动初始化功能，初始化 lcd (伪代码)：

```
int lcd_init(void) /* 串口初始化函数 */
{
    // 此处将初始化的代码放入，如初始化 lcd
    rt_kprintf("test auto init: lcd init success!\n");
    return 0;
}
INIT_APP_EXPORT(lcd_init); /* 使用自动初始化机制 */
```

4.2 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，复位后，直接自动调用了 lcd_init() 函数，并打印了相应的 Log 信息。

```
\ | /  
- RT -   Thread Operating System  
/ | \    4.1.1 build Jul 17 2023 16:08:46  
2006 - 2022 Copyright by RT-Thread team  
test auto init: lcd init success!  
msh >
```

图 4.1: 自动初始化

第 5 章

线程的使用

线程，即任务的载体。一般被设计成 `while(1)` 的循环模式，但在循环中一定要有让出 CPU 使用权的动作。如果是可以执行完毕的线程，则系统会自动将执行完毕的线程进行删除 / 脱离。

参考：[文档中心——线程管理](#)

5.1 代码设计

本例程源码为：`thread_sample.c`

为了体现线程的创建、初始化与脱离，本例程设计了 `thread1`、`thread2` 两个线程。`thread1` 是创建的动态线程，优先级为 25；`Thread2` 初始化的静态线程，优先级为 24。

优先级较高的 `Thread2` 抢占低优先级的 `thread1`，执行完毕一段程序后自动被系统脱离。

优先级较低的 `thread1` 被设计成死循环，循环中有让出 CPU 使用权的动作 – 使用了 `delay` 函数。该线程在 `thread2` 退出运行之后开始运行，每隔一段时间运行一次，并一直循环运行下去。

用户可以清晰地了解到线程在本例程中的状态变迁情况。

整个运行过程如下图所示，描述如下：

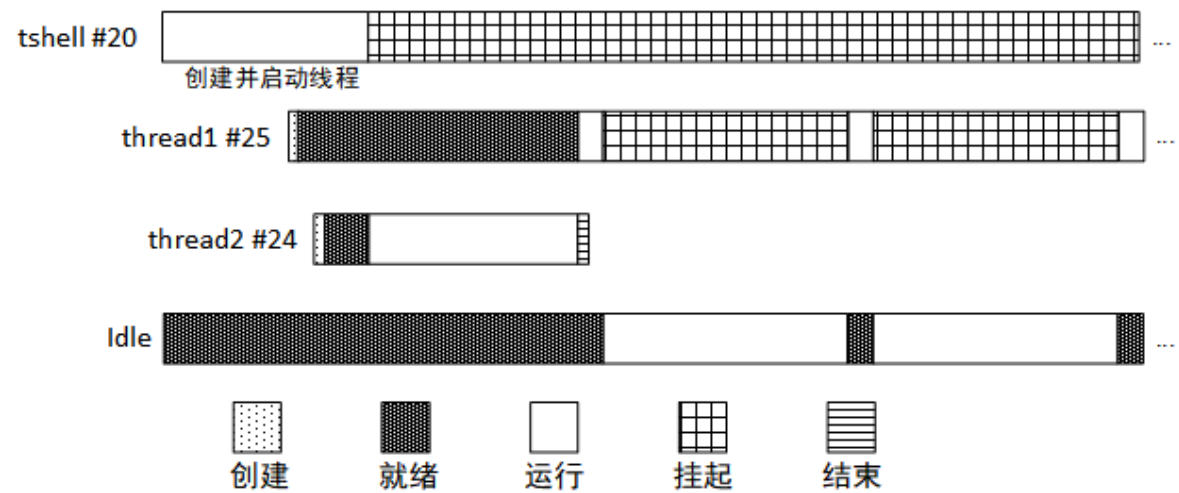


图 5.1: 运行过程

(1) 在 `tshell` 线程 (优先级 20) 中创建线程 `thread1` 和初始化 `thread2`, `thread1` 优先级为 25, `thread2` 优先级为 24;

(2) 启动线程 `thread1` 和 `thread2`, 使 `thread1` 和 `thread2` 处于就绪状态;

(3) 随后 `tshell` 线程挂起, 在操作系统的调度下, 优先级较高的 `thread2` 首先被投入运行;

(4) `thread2` 是可执行完毕线程, 运行完毕打印之后, 系统自动删除 `thread2`;

(5) `thread1` 得以运行, 打印信息之后执行延时将自己挂起;

(6) 系统中没有优先级更高的就绪队列, 开始执行空闲线程;

(7) 延时时间到, 执行 `thread1`;

(8) 循环 (5) ~ (7)。

5.2 线程使用示例

示例代码通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

定义了待创建线程需要用到的优先级, 栈空间, 时间片的宏, 定义线程 `thread1` 的线程句柄:

```
# 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 `thread1` 入口函数, 每 500ms 打印一次计数值

```
/* 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);
    }
}
```

线程 `thread2` 线程栈、控制块以及线程 2 入口函数的定义, 线程 2 打印计数, 10 次后退出。

```
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 */
}
```

例程代码，其中创建了线程 `thread1`，初始化了线程 `thread2`，并将函数使用 `MSH_CMD_EXPORT` 导出命令。

```
/* 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);
```

5.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 **thread_sample** 命令启动示例应用，示例输出结果如下：

```
msh >
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
...
```

第 6 章

线程的时间片轮转调度

线程的时间片轮转调度，可以了解到：

- 多线程时间片轮转的基本原理；
- 同优先级线程间的时间片轮转机制；

6.1 代码设计

使用的例程为：timeslice_sample.c

为了体现时间片轮转，本例程设计了 **thread1**、**thread2** 两个相同优先级的线程，**thread1** 时间片为 10，**thread2** 时间片为 5，如果就绪列表中该优先级最高，则这两个线程会按照时间片长短被轮番调度。两个线程采用同一个入口函数，分别打印一条带有累加计数的信息（每个线程进入一次入口函数会将计数 **count++**，**count>200** 时线程退出）遵循时间片轮转调度机制。

通过本例程，用户可以清晰地了解到，同优先级线程在时间片轮转调度时刻的状态变迁。

整个运行过程如下图所示，OS Tick 为系统滴答时钟（精度 10ms），下面以例程开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：

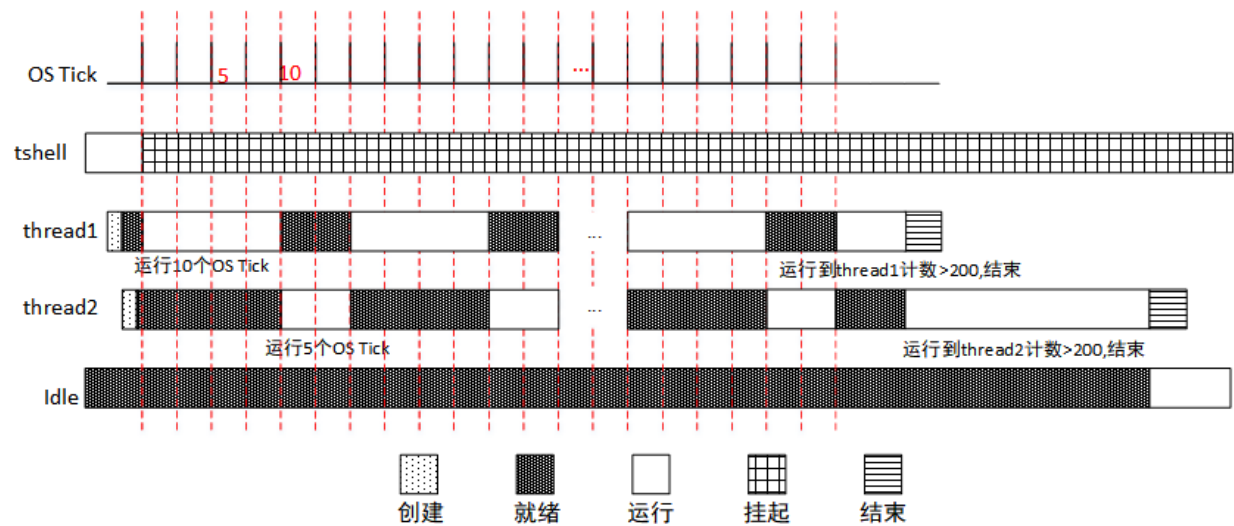


图 6.1: 运行过程

(1) 在 `tshell` 线程中创建线程 `thread1` 和 `thread2`, 优先级相同为 20, `thread1` 时间片为 10, `thread2` 时间片为 5;

(2) 启动线程 `thread1` 和 `thread2`, 使 `thread1` 和 `thread2` 处于就绪状态;

(3) 在操作系统的调度下, `thread1` 首先被投入运行;

(4) `thread1` 循环打印带有累计计数的信息, 当 `thread1` 运行到第 10 个时间片时, 操作系统调度 `thread2` 投入运行, `thread1` 进入就绪状态;

(5) `thread2` 开始运行后, 循环打印带有累计计数的信息, 直到第 15 个 OS Tick 到来, `thread2` 已经运行了 5 个时间片, 操作系统调度 `thread1` 投入运行, `thread2` 进入就绪状态;

(6) `thread1` 运行直到计数值 `count>200`, 线程 `thread1` 退出, 接着调度 `thread2` 运行直到计数值 `count>200`, `thread2` 线程退出; 之后操作系统调度空闲线程投入运行;

注意: 时间片轮转机制, 在 OS Tick 到来时, 正在运行的线程时间片减 1。

6.2 时间片示例

示例代码通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

例程定义了待创建线程需要用到的优先级, 栈空间, 时间片的宏:

```
#include <rtthread.h>

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

两个线程公共的入口函数, 线程 `thread1` 和 `thread2` 采用同一个入口函数, 但是变量分别存在不同的堆空间

```
/* 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++;
    }
}

```

线程时间片的示例函数，示例函数首先创建并启动了线程 **thread1**，然后创建并启动了线程 **thread2**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

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);

```

6.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 **timeslice_sample** 命令启动示例应用，示例输出结果如下：

```

msh >
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 = 20
...
thread 1 is running ,thread 1 count = 125
thread 1 is running ,thread 1 count = 1thread 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 = 60
thread 2 is running ,thread 2 co30
thread 1 is running ,thread 1 count = 135
thread 1 is running ,thread 1 count = 140
thread 1 is running ,thread 1 count = 145
...
thread 1 is running ,thread 1 count = 205
unt = 205thread 2 is running ,thread 2 count = 70
thread 2 is running ,thread 2 count = 75
thread 2 is running ,thread 2 count = 80
...
thread 2 is running ,thread 2 count = 200
thread 2 is running ,thread 2 count = 205
```

线程 `thread1` 在 10 个 OS Tick 中，可计数约 125 左右，计数 > 200 会退出，所以下一次执行不了 10 个 OS Tick 就会退出了。由于“计数 > 200 会退出”，`Thread1` 与 `thread2` 只会轮番调度一次就会先后退出了。

第 7 章

调度器

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。

7.1 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：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）。

7.2 调度器钩子示例

例程源码：scheduler_hook_sample.c

使用 `rt_scheduler_sethook` 设置调度器钩子函数，在钩子函数中打印线程的切换。

```
/* 设置调度器钩子 */
rt_scheduler_sethook(hook_of_scheduler);

/* 钩子函数 */
static void hook_of_scheduler(struct rt_thread *from, struct rt_thread *to)
{
    #if RT_VER_NUM >= 0x50001
        rt_kprintf("from: %s --> to: %s \n", from->parent.name, to->parent.name);
    #else
        rt_kprintf("from: %s --> to: %s \n", from->name, to->name);
    #endif
}
```

7.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `scheduler_hook` 命令启动示例应用，示例输出结果如下，可以看到系统中的线程都在执行切换。

```
msh >
msh >scheduler_hook
msh >from: tshell --> to: thread2
thread 2 is running
from: thread2 --> to: thread1
thread 1 is running
from: thread1 --> to: tidle0
from: tidle0 --> to: thread2
thread 2 is running
from: thread2 --> to: thread1
thread 1 is running
from: thread1 --> to: tidle0
```

```
from: tidle0 --> to: thread2  
thread 2 is running  
from: thread2 --> to: thread1  
thread 1 is running  
from: thread1 --> to: tidle0  
from: tidle0 --> to: thread2  
thread 2 is running  
from: thread2 --> to: thread1  
thread 1 is running  
from: thread1 --> to: tidle0  
from: tidle0 --> to: thread2  
thread 2 is running  
from: thread2 --> to: thread1
```

第 8 章

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。

8.1 空闲线程钩子函数

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：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。并且，由于 `malloc`、`free` 等内存相关的函数内部使用了信号量作为临界区保护，因此在钩子函数内部也不允许调用此类函数！

8.2 代码设计

本例程源码为：idlehook_sample.c

这个例程创建一个线程，通过延时进入空闲任务钩子，用于打印进入空闲钩子的次数

8.3 空闲钩子函数使用示例

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

#define THREAD_PRIORITY    20
#define THREAD_STACK_SIZE 1024
#define THREAD_TIMESLICE  5

/* 指向线程控制块的指针 */
static rt_thread_t tid = RT_NULL;

/* 空闲线程钩子函数执行次数 */
volatile static int hook_times = 0;

/* 空闲线程钩子函数 */
static void idle_hook()
{
    if (0 == (hook_times % 10000))
    {
        rt_kprintf("enter idle hook %d times.\n", hook_times);
    }

    rt_enter_critical();
    hook_times++;
    rt_exit_critical();
}
```

```

}

/* 线程入口 */
static void thread_entry(void *parameter)
{
    int i = 5;
    while (i--)
    {
        rt_kprintf("enter thread1.\n");
        rt_enter_critical();
        hook_times = 0;
        rt_exit_critical();

        /* 休眠500ms */
        rt_kprintf("thread1 delay 500ms.\n");
        rt_thread_mdelay(500);
    }
    rt_kprintf("delete idle hook.\n");

    /* 删除空闲线程钩子函数 */
    rt_thread_idle_delhook(idle_hook);
    rt_kprintf("thread1 finish.\n");
}

int idle_hook_sample(void)
{
    /* 设置空闲线程钩子 */
    rt_thread_idle_sethook(idle_hook);

    /* 创建线程 */
    tid = rt_thread_create("thread1",
                           thread_entry, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY, THREAD_TIMESLICE);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(idle_hook_sample, idle hook sample);

```

8.4 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `idle_hook_sample` 命令启动示例应用，示例输出结果如下：


```
msh >
msh >idle_hook_sample
msh >enter thread1.
thread1 delay 500ms.
enter idle hook 0 times.
enter idle hook 10000 times.
enter idle hook 20000 times.
enter idle hook 30000 times.
...
enter idle hook 200000 times.
enter thread1.
thread1 delay 500ms.
enter idle hook 0 times.
...
enter idle hook 200000 times.
...
enter thread1.
thread1 delay 500ms.
enter idle hook 0 times.
enter idle hook 10000 times.
...
enter idle hook 200000 times.
delete idle hook.
thread1 finish.
```

第 9 章

临界区

多个线程操作 / 访问同一块区域（代码），这块代码就称为临界区，上述例子中的共享内存块就是临界区。线程互斥是指对于临界区资源访问的排它性。当多个线程都要使用临界区资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程的同步方式有很多种，其核心思想都是：在访问临界区的时候只允许一个（或一类）线程运行。进入 / 退出临界区的方式有很多种：

- 1) 调用 `rt_hw_interrupt_disable()` 进入临界区，调用 `rt_hw_interrupt_enable()` 退出临界区。
- 2) 调用 `rt_enter_critical()` 进入临界区，调用 `rt_exit_critical()` 退出临界区。

9.1 全局开关中断

全局中断开关也称为中断锁，是禁止多线程访问临界区最简单的一种方式，即通过关闭中断的方式，来保证当前线程不会被其他事件打断（因为整个系统已经不再响应那些可以触发线程重新调度的外部事件），也就是当前线程不会被抢占，除非这个线程主动放弃了处理器控制权。当需要关闭整个系统的中断时，可调用下面的函数接口：

```
rt_base_t rt_hw_interrupt_disable(void);
```

下表描述了此函数的返回值：

`rt_hw_interrupt_disable()` 的返回值

返回	描述
中断状态	<code>rt_hw_interrupt_disable</code> 函数运行前的中断状态

恢复中断也称开中断。`rt_hw_interrupt_enable()` 这个函数用于“使能”中断，它恢复了调用 `rt_hw_interrupt_disable()` 函数前的中断状态。如果调用 `rt_hw_interrupt_disable()` 函数前是关中断状态，那么调用此函数后依然是关中断状态。恢复中断往往是和关闭中断成对使用的，调用的函数接口如下：

```
void rt_hw_interrupt_enable(rt_base_t level);
```

下表描述了此函数的输入参数：

`rt_hw_interrupt_enable()` 的输入参数

参数	描述
level	前一次 <code>rt_hw_interrupt_disable</code> 返回的中断状态

1) 使用中断锁来操作临界区的方法可以应用于任何场合，且其他几类同步方式都是依赖于中断锁而实现的，可以说中断锁是最强大的和最高效的同步方法。只是使用中断锁最主要的问题在于，在中断关闭期间系统将不再响应任何中断，也就不能响应外部的事件。所以中断锁对系统的实时性影响非常巨大，当使用不当的时候会导致系统完全无实时性可言（可能导致系统完全偏离要求的时间需求）；而使用得当，则会变成一种快速、高效的同步方式。

例如，为了保证一行代码（例如赋值）的互斥运行，最快速的方法是使用中断锁而不是信号量或互斥量：

```
/* 关闭中断 */
level = rt_hw_interrupt_disable();
a = a + value;
/* 恢复中断 */
rt_hw_interrupt_enable(level);
```

在使用中断锁时，需要确保关闭中断的时间非常短，例如上面代码中的 `a = a + value;` 也可换成另外一种方式，例如使用信号量：

```
/* 获得信号量锁 */
rt_sem_take(sem_lock, RT_WAITING_FOREVER);
a = a + value;
/* 释放信号量锁 */
rt_sem_release(sem_lock);
```

这段代码在 `rt_sem_take`、`rt_sem_release` 的实现中，已经存在使用中断锁保护信号量内部变量的行为，所以对于简单如 `a = a + value;` 的操作，使用中断锁将更为简洁快速。

2) 函数 `rt_base_t rt_hw_interrupt_disable(void)` 和函数 `void rt_hw_interrupt_enable(rt_base_t level)` 一般需要配对使用，从而保证正确的中断状态。

在 RT-Thread 中，开关全局中断的 API 支持多级嵌套使用，简单嵌套中断的代码如下代码所示：

简单嵌套中断使用

```
#include <rthw.h>

void global_interrupt_demo(void)
{
    rt_base_t level0;
    rt_base_t level1;
```

```

/* 第一次关闭全局中断，关闭之前的全局中断状态可能是打开的，也可能是关闭的 */
level0 = rt_hw_interrupt_disable();
/* 第二次关闭全局中断，关闭之前的全局中断是关闭的，关闭之后全局中断还是关闭的 */
level1 = rt_hw_interrupt_disable();

do_something();

/* 恢复全局中断到第二次关闭之前的状态，所以本次 enable 之后全局中断还是关闭的 */
rt_hw_interrupt_enable(level1);
/* 恢复全局中断到第一次关闭之前的状态，这时候的全局中断状态可能是打开的，也可能是关闭的 */
rt_hw_interrupt_enable(level0);
}

```

这个特性可以给代码的开发带来很大的便利。例如在某个函数里关闭了中断，然后调用某些子函数，再打开中断。这些子函数里面也可能存在开关中断的代码。由于全局中断的 API 支持嵌套使用，用户无需为这些代码做特殊处理。

9.2 进入退出临界区

调用 `rt_enter_critical()` 进入临界区，调用 `rt_exit_critical()` 退出临界区，函数原型如下所示：

```

void rt_enter_critical(void);
void rt_exit_critical(void);

```

两个函数均没有参数和返回，`rt_enter_critical` 此函数将锁定线程调度程序。

9.3 代码设计

本例程源码为：`interrupt_sample.c`

使用开关全局中断的方式保护临界区 `cnt` 的安全。

9.4 临界区保护示例

```

#include <rthw.h>
#include <rtthread.h>

#define THREAD_PRIORITY      20
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 同时访问的全局变量 */
static rt_uint32_t cnt;

```

```

static void thread_entry(void *parameter)
{
    rt_uint32_t no;
    rt_uint32_t level;

    no = (rt_uint32_t) parameter;
    while (1)
    {
        /* 关闭中断 */
        level = rt_hw_interrupt_disable();
        cnt += no;
        /* 恢复中断 */
        rt_hw_interrupt_enable(level);

        rt_kprintf("protect thread[%d]'s counter is %d\n", no, cnt);
        rt_thread_mdelay(no * 10);
    }
}

int interrupt_sample(void)
{
    rt_thread_t thread;

    /* 创建thread1线程 */
    thread = rt_thread_create("thread1", thread_entry, (void *)10,
                              THREAD_STACK_SIZE,
                              THREAD_PRIORITY, THREAD_TIMESLICE);

    if (thread != RT_NULL)
        rt_thread_startup(thread);

    /* 创建thread2线程 */
    thread = rt_thread_create("thread2", thread_entry, (void *)20,
                              THREAD_STACK_SIZE,
                              THREAD_PRIORITY, THREAD_TIMESLICE);

    if (thread != RT_NULL)
        rt_thread_startup(thread);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(interrupt_sample, interrupt sample);

```

9.5 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `interrupt_sample` 命令启动示例应用，示例输出结果如下：

```
msh >interrupt_sample
msh >protect thread[20]'s counter is 20
protect thread[10]'s counter is 30
protect thread[10]'s counter is 40
protect thread[20]'s counter is 60
protect thread[10]'s counter is 70
protect thread[10]'s counter is 80
protect thread[20]'s counter is 100
protect thread[10]'s counter is 110
protect thread[10]'s counter is 120
protect thread[20]'s counter is 140
protect thread[10]'s counter is 150
protect thread[10]'s counter is 160
protect thread[20]'s counter is 180
protect thread[10]'s counter is 190
protect thread[10]'s counter is 200
protect thread[20]'s counter is 220
```

第 10 章

定时器的使用

RT-Thread 定时器由操作系统提供的一类系统接口（函数），它构建在芯片的硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。

RT-Thread 定时器分为 `HARD_TIMER` 与 `SOFT_TIMER`，可以设置为单次定时与周期定时，这些属性均可在创建 / 初始化定时器时设置；而如果没有设置 `HARD_TIMER` 或 `SOFT_TIMER`，则默认使用 `HARD_TIMER`。

参考：[文档中心——定时器管理](#)

10.1 代码设计

本例程源码为：`timer_sample.c`

为了体现动态定时器的单次定时与周期性定时，本例程设计了 `timer1`、`timer2` 两定时器。

周期性定时器 1 的超时函数，每 5 个 OS Tick 运行 1 次，共运行 5 次（5 次后调用 `rt_timer_stop` 使定时器 1 停止运行）；单次定时器 2 的超时函数在第 15 个 OS Tick 时运行一次。

通过本例程，用户可以清晰地了解到定时器的工作过程，以及使用定时器相关 API 动态更改定时器属性。

整个运行过程如下图所示，描述如下：

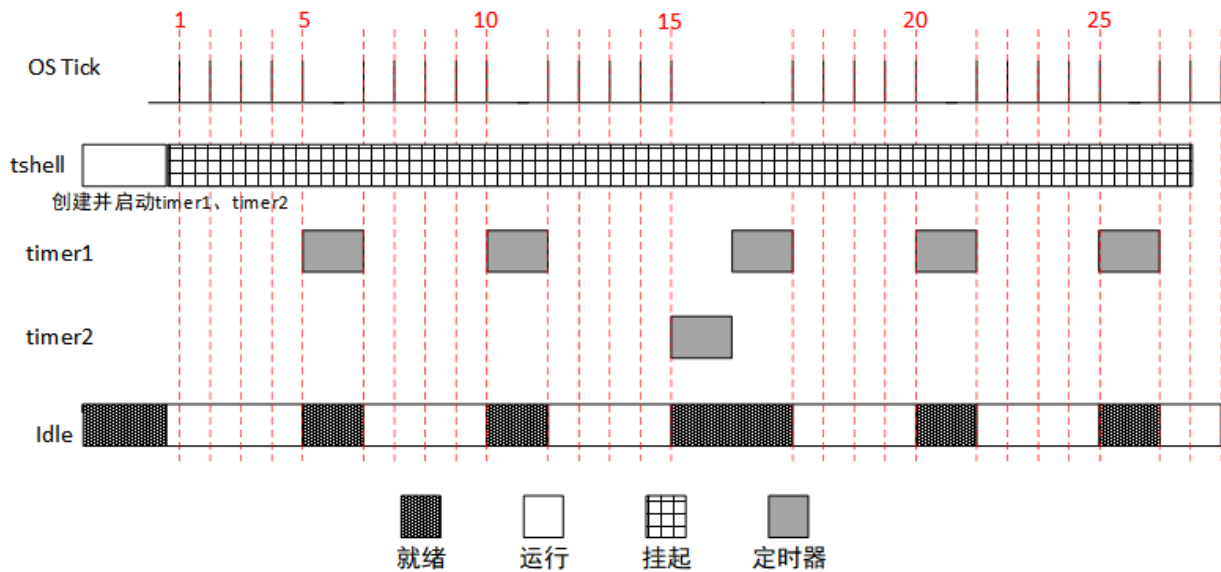


图 10.1: 运行过程

(1) 在 **tshell** 线程中创建定时器 **timer1** 和 **timer2**，**timer1** 周期定时 5 OS Tick，**timer2** 单次定时 15 OS Tick；启动定时器 **timer1**、**timer2**；

(2) 定时器的定时时间均为到，在操作系统的调度下，**Idle** 投入运行；

(3) 每 5 个 OS Tick 到来时，定时器 **timer1** 定时时间到，调用超时函数打印一段信息，**timer1** 定时器重置；

(4) 在第 15 个 OS Tick 到来时，**timer1** 第 3 次超时，调用超时函数打印一段信息；**timer2** 第一次超时，调用超时函数打印一段信息且超时函数运行完删除；

(5) 在第 25 个 OS Tick 到来时，定时器 **timer1** 第 5 次超时，调用超时函数打印一段信息，并使用 **rt_timer_stop()** 接口将定时器停止，超时函数运行完后自行删除；

10.2 定时器使用示例

示例代码通过 **MSH_CMD_EXPORT** 将示例初始函数导出到 **msh** 命令，可以在系统运行过程中，通过在控制台输入命令来启动。头文件以及定义了待创建定时器控制块以及例程需要用到的变量

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

```
/* 定时器的控制块 */
```

```
static rt_timer_t timer1;
```

```
static rt_timer_t timer2;
```

```
static int cnt = 0;
```

周期定时器 **timer1** 的超时函数，**timer1** 定时时间到会执行次函数，10 次之后停止定时器 **timer1**。

```
/* 定时器 1 超时函数 */
```

```
static void timeout1(void *parameter)
```

```
{
```



```

    rt_kprintf("periodic timer is timeout %d\n", cnt);

    /* 运行第 10 次，停止周期定时器 */
    if (cnt++>= 9)
    {
        rt_timer_stop(timer1);
        rt_kprintf("periodic timer was stopped! \n");
    }
}

```

单次定时器 **timer2** 的超时函数，**timer2** 定时时间到会执行次函数

```

/* 定时器 2 超时函数 */
static void timeout2(void *parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}

```

定时器的示例代码，示例函数首先创建并启动了线程 **timer1**，然后创建并启动了线程 **timer2**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

int timer_sample(void)
{
    /* 创建定时器 1 周期定时器 */
    timer1 = rt_timer_create("timer1", timeout1,
                             RT_NULL, 10,
                             RT_TIMER_FLAG_PERIODIC);

    /* 启动定时器 1 */
    if (timer1 != RT_NULL)
        rt_timer_start(timer1);

    /* 创建定时器 2 单次定时器 */
    timer2 = rt_timer_create("timer2", timeout2,
                             RT_NULL, 30,
                             RT_TIMER_FLAG_ONE_SHOT);

    /* 启动定时器 2 */
    if (timer2 != RT_NULL)
        rt_timer_start(timer2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(timer_sample, timer sample);

```

以上为示例函数，可以看到将函数使用 **MSH_CMD_EXPORT** 导出命令，示例函数首先创建并启动了定时器 **timer1**，然后创建并启动了定时器 **timer2**。

10.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 **timer_sample** 命令启动示例应用，示例输出结果如下：

```
msh >
msh >timer_sample
msh >periodic timer is timeout 0
periodic timer is timeout 1
one shot timer is timeout
periodic timer is timeout 2
periodic timer is timeout 3
periodic timer is timeout 4
periodic timer was stopped!
```

第 11 章

信号量—生产者消费者问题

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。

本例程通过信号量—生产者消费者问题：

- 理解信号量的基本原理；
- 使用信号量来达到线程间同步；
- 理解资源计数适合于线程间工作处理速度不匹配的场所；

Tip: 信号量在大于 0 时才能获取，在中断、线程中均可释放信号量。

参考：[文档中心——IPC 管理之信号量](#)

11.1 代码设计

本例程源码为：producer_consumer_sample.c

为了体现使用信号量来达到线程间的同步，本例程设计了 producer、consumer 两个线程，producer 优先级为 24，consumer 优先级为 26。线程 producer 每生产一个数据进入 20ms 延时，生产 10 个数据后结束。线程 consumer 每消费一个数据进入 50ms 延时，消费 10 个数据后结束。通过本例程，用户可以清晰地了解到，信号量在线程同步以及资源计数时起到的作用。

整个运行过程如下图所示，OS Tick 为系统滴答时钟，下面以例程开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：

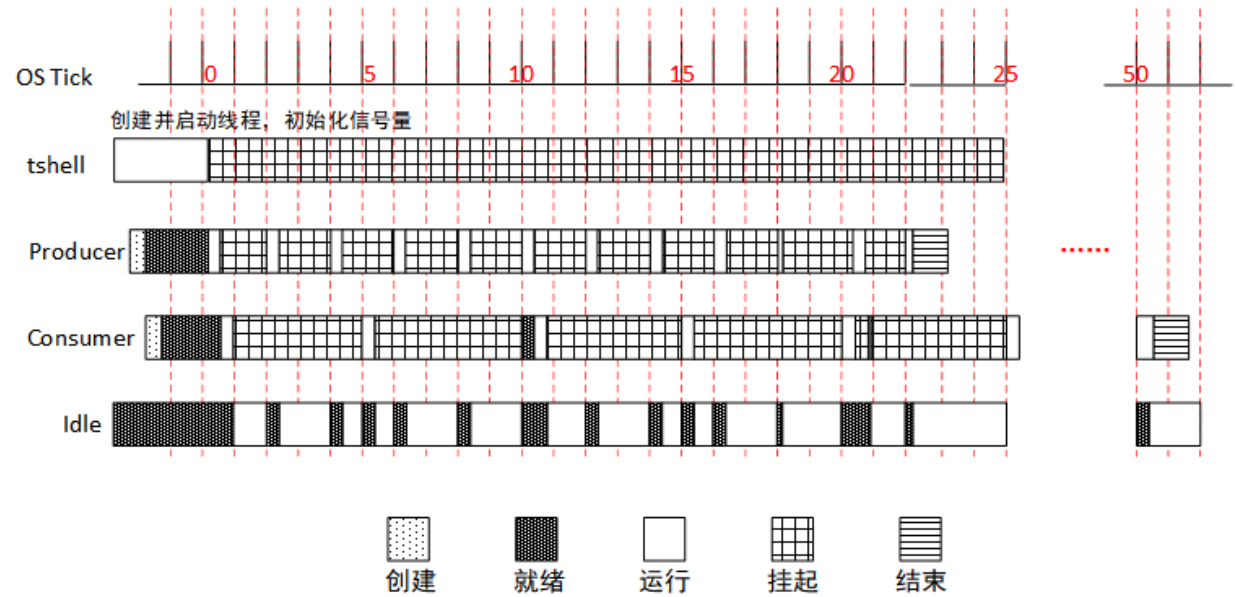


图 11.1: 运行过程

(1) 在 tshell 线程中初始化 3 个信号量，lock 初始化为 1（用作保护临界区，保护数组），empty 初始化为 5，full 初始化为 0；信号量情况：

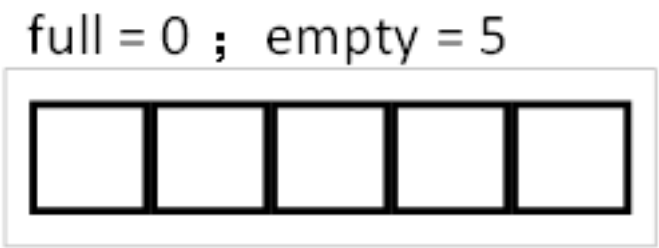


图 11.2: 信号量情况 1

- (2) 创建并启动线程 producer，优先级为 24；创建并启动线程和 consumer，优先级为 26；
- (3) 在操作系统的调度下，producer 优先级高，首先被投入运行；
- (4) producer 获取一个 empty 信号量，产生一个数据放入数组，再释放一个 full 信号量，然后进入 2 OS Tick 延时；之后的信号量情况：

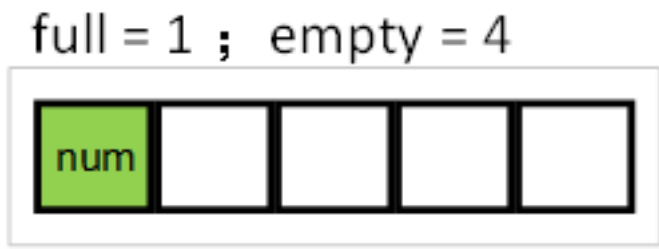


图 11.3: 信号量情况 2

- (5) 随后 consumer 投入运行，获取一个 full 信号量，消费一个数据用于累加，再释放一个 empty 信

号量，然后进入 5 OS Tick 延时；之后的信号量情况：

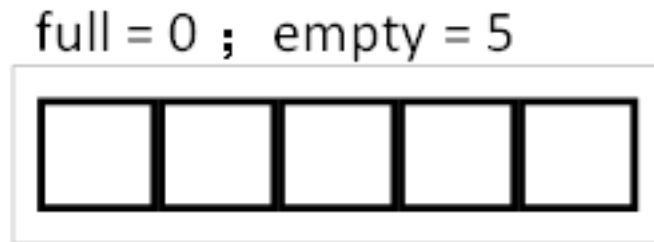


图 11.4: 信号量情况 3

(6) 由于生产速度 > 消费速度，所以在某一时刻会存在 full = 5 / empty = 0 的情况，如下：

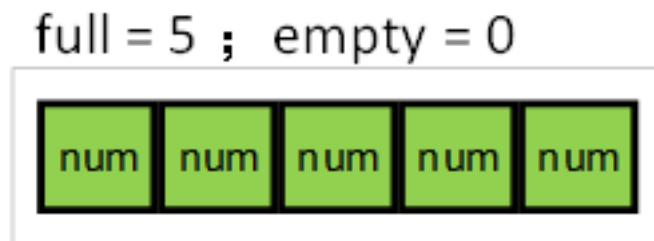


图 11.5: 信号量情况 4

比如第 18 个 OS Tick 时，producer 延时结束，操作系统调度 producer 投入运行，获取一个 empty 信号量，由于此时 empty 信号量为 0，producer 由于获取不到信号量挂起；等待有 empty 信号时，才可以继续生产。

(7) 直到 producer 产生 10 个 num 后，producer 线程结束，被系统删除。

(8) 直到 consumer 消费 10 个 num 后，consumer 线程结束，被系统删除。

11.2 生产者消费者示例

示例代码通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

定义了待创建线程需要用到的优先级，栈空间，时间片的宏，以及生产消费过程中用于存放产生数据的数字和相关变量、线程句柄、信号量控制块。

```
#include <rtthread.h>

#define THREAD_PRIORITY      6
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 定义最大 5 个元素能够被产生 */
#define MAXSEM 5
```

```

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

/* 指向生产者、消费者在 array 数组中的读写位置 */
static rt_uint32_t set, get;

/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;

```

生产者 **producer** 线程的入口函数，每 20ms 就获取一个空位（获取不到时挂起），上锁，产生一个数字写入数组，解锁，释放一个满位，10 次后结束。

```

/* 生产者线程入口 */
void producer_thread_entry(void *parameter)
{
    int cnt = 0;

    /* 运行 10 次 */
    while (cnt < 10)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改 array 内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set % MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n", array[set % MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_mdelay(20);
    }

    rt_kprintf("the producer exit!\n");
}

```

消费者 **consumer** 线程的入口函数，每 50ms 获取一个满位（获取不到时挂起），上锁，将数组中的内容相加，解锁，释放一个空位，10 次后结束。

```

/* 消费者线程入口 */

```

```

void consumer_thread_entry(void *parameter)
{
    rt_uint32_t sum = 0;

    while (1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get % MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", (get % MAXSEM), array[get
            % MAXSEM]);
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到 10 个数目，停止，消费者线程相应停止 */
        if (get == 10) break;

        /* 暂停一小会时间 */
        rt_thread_mdelay(50);
    }

    rt_kprintf("the consumer sum is: %d\n", sum);
    rt_kprintf("the consumer exit!\n");
}

```

生产者与消费者问题的示例函数，示例函数首先初始化了 3 个信号量，创建并启动生产者线程 producer，然后创建、启动消费者线程 consumer。并将函数使用 MSH_CMD_EXPORT 导出命令。

```

int producer_consumer(void)
{
    set = 0;
    get = 0;

    /* 初始化 3 个信号量 */
    rt_sem_init(&sem_lock, "lock", 1, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty", MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full, "full", 0, RT_IPC_FLAG_FIFO);

    /* 创建生产者线程 */
    producer_tid = rt_thread_create("producer",
                                    producer_thread_entry, RT_NULL,
                                    THREAD_STACK_SIZE,
                                    THREAD_PRIORITY - 1, THREAD_TIMESLICE);
}

```

```
if (producer_tid != RT_NULL)
    rt_thread_startup(producer_tid);

/* 创建消费者线程 */
consumer_tid = rt_thread_create("consumer",
                                consumer_thread_entry, RT_NULL,
                                THREAD_STACK_SIZE,
                                THREAD_PRIORITY + 1, THREAD_TIMESLICE);

if (consumer_tid != RT_NULL)
    rt_thread_startup(consumer_tid);

return 0;
}

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

11.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 **producer_consumer** 命令启动示例应用，示例输出结果如下：

```
msh >
msh >producer_consumer
the producer generates a number: 1
the consumer[0] get a number: 1
msh >the producer generates a number: 2
the producer generates a number: 3
the consumer[1] get a number: 2
the producer generates a number: 4
the producer generates a number: 5
the producer generates a number: 6
the consumer[2] get a number: 3
the producer generates a number: 7
the producer generates a number: 8
the consumer[3] get a number: 4
the producer generates a number: 9
the consumer[4] get a number: 5
the producer generates a number: 10
the producer exit!
the consumer[0] get a number: 6
the consumer[1] get a number: 7
the consumer[2] get a number: 8
the consumer[3] get a number: 9
the consumer[4] get a number: 10
the consumer sum is: 55
the consumer exit!
```


第 12 章

互斥量——优先级继承

互斥量是一种特殊的二值信号量。它和信号量不同的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。

注意：需要切记的是互斥量不能在中断服务例程中使用。

参考：[文档中心——IPC 管理之互斥量](#)

12.1 代码设计

本例程源码为：priority_inheritance_sample.c。

为了体现使用互斥量来达到线程间的同步，并体现优先级继承的现象，本例程设计了 threadC、threadA 两个线程，优先级分别为 9、8，设计了一个互斥量 mutex。

线程 threadC 进入后先打印自己的优先级，然后进入 100ms 延时，延时结束后获取互斥量 mutex，获取到互斥量之后持有一段时间，再释放互斥量 mutex。

线程 threadA 进入后先打印自己的优先级，然后尝试获取互斥量 mutex，获取到互斥量之后持有一段时间，之后将互斥量释放。

其中优先级继承的情况就是：低优先级线程 C 先持有互斥量，而后高优先级线程 A 试图持有互斥量，此时线程 C 的优先级应该被提升为和线程 A 的优先级相同。

通过本例程，用户可以清晰地了解到，互斥量在线程间同步的作用、互斥量的优先级继承性以及互斥量连续获取不会造成死锁。

12.2 源程序说明

示例代码通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

定义了待创建线程需要用到的优先级，栈空间，时间片的宏，以及线程控制块句柄和互斥量控制块句柄

```
#include <rtthread.h>

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

#define THREAD_PRIORITY      9
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5
```

线程 **thread C** 的入口函数，打印优先级信息，之后获取互斥量并长时间运行，让高优先级的 **thread A** 主动抢占 CPU，然后打印优先级，最后释放互斥量。

```
/* 线程 C 入口 */
static void thread_c_entry(void *parameter)
{
    rt_tick_t tick;

    /* 预留创建并启动线程 A 的时间 */
    rt_thread_delay(100);
    rt_kprintf("thread C priority is: %d\n", tidc->current_priority);

    /* 获取互斥量 */
    rt_mutex_take(mutex, RT_WAITING_FOREVER);
    rt_kprintf("thread C priority is: %d, take the mutex\n", tidc->current_priority);
    ;

    /* 持续持有互斥量 3000ms，线程 A 优先级高会主动抢占 CPU */
    tick = rt_tick_get();
    while (rt_tick_get() - tick < 3000) ;

    rt_kprintf("thread C priority is: %d, running...\n", tidc->current_priority);

    /* 释放互斥锁 */
    rt_kprintf("thread C release the mutex\n\n");
    rt_mutex_release(mutex);
    rt_kprintf("thread C priority is: %d\n", tidc->current_priority);
    rt_kprintf("thread C exit\n");
}
```

线程 **thread A** 的入口函数，先打印自身优先级信息，然后获取互斥量，获取到互斥量之后进行 500ms 的长时间循环，使 **thread3** 运行 500ms 左右，之后释放互斥量。

```
/* 线程 A 入口 */
static void thread_a_entry(void *parameter)
{
    rt_tick_t tick;
```

```

/* 让出更多的时间让线程 C先运行 */
rt_thread_delay(200);

/* 线程 A 尝试获取互斥量 */
rt_kprintf("thread A priority is: %d, try to take the mutex\n", tida->
    current_priority);
rt_mutex_take(mutex, RT_WAITING_FOREVER);
rt_kprintf("thread A take the mutex\n");

rt_kprintf("thread A running...\n");

/* 持续持有互斥量 500ms, 线程 A 优先级高会一直运行 */
tick = rt_tick_get();
while (rt_tick_get() - tick < 500) ;

rt_kprintf("thread A release the mutex\n");
rt_mutex_release(mutex);
rt_kprintf("thread A exit\n\n");
}

```

互斥量优先级继承的例子，解决优先级翻转问题。示例函数首先创建互斥量，再创建、启动了线程 threadC、threadA。并将函数使用 MSH_CMD_EXPORT 导出命令。

```

int pri_inheritance(void)
{
    /* 创建互斥量 mutex */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_PRIO);
    if (mutex == RT_NULL)
    {
        rt_kprintf("create dynamic mutex failed.\n");
        return -1;
    }

    /* 创建线程 C 并启动 */
    tidc = rt_thread_create("threadc",
        thread_c_entry,
        RT_NULL,
        2048,
        THREAD_PRIORITY,
        5);
    RT_ASSERT(tidc != RT_NULL);
    rt_thread_startup(tidc);

    /* 创建线程 A 并启动 */
    tida = rt_thread_create("threada",
        thread_a_entry,
        RT_NULL,
        2048,

```

```
        THREAD_PRIORITY - 1,
        5);
    RT_ASSERT(tida != RT_NULL);
    rt_thread_startup(tida);

    return RT_EOK;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pri_inheritance, pri inheritance);
```

12.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `pri_inversion` 命令启动示例应用，示例输出结果如下：

```
msh >
msh >pri_inheritance
thread C priority is: 9          # 线程 C 的优先级为 9
thread C priority is: 9, take the mutex
thread A priority is: 8, try to take the mutex
thread C priority is: 8, running... # 线程 C 继承线程 A 的优先级，变为 8
thread C release the mutex

thread A take the mutex
thread A running...
thread A release the mutex
thread A exit

thread C priority is: 9          # 线程 C 的优先级恢复为 9
thread C exit
```

例程演示了互斥量的优先级继承特性。线程 C 先持有互斥量，而后线程 A 试图持有互斥量，此时线程 C 的优先级被提升为和线程 A 的优先级相同。

第 13 章

事件集的使用

事件集主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程与多个事件的关系可设置为：其中任意一个事件唤醒线程，或几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件。

参考：[文档中心——IPC 管理之事件集](#)

13.1 代码设计

本例程源码为：`event_sample.c`

为了体现使用事件集来达到线程间的同步，本例程设计了 `thread1`、`thread2` 两个线程，优先级分别为 8、9，设计了一个事件集 `event`。

线程 `thread1` 进入后接收事件组合“事件 3 或事件 5”，接收到事件时候进行 100ms 延时，然后接收事件组合“事件 3 与事件 5”，接收完成后结束线程。

线程 `thread2` 进入后发送事件 3，延时 200ms；发送事件 5，延时 200ms；发送事件 3，完成后结束线程。

整体情况：`thread1` 首先等待“事件 3 或事件 5”的到来，`thread2` 发送事件 3，唤醒 `thread1` 接收事件，之后 `thread1` 等待“事件 3 与事件 5”；`thread2` 再发送事件 5，进行延时，`thread2` 发送事件 3，等 `thread1` 延时结束就能接收事件组合“事件 3 与事件 5”。

通过本例程，用户可以清晰地了解到，线程在同时接收多个事件和接收多个事件中的一个时的运行情况。

整个运行过程如下图所示，过程描述如下：

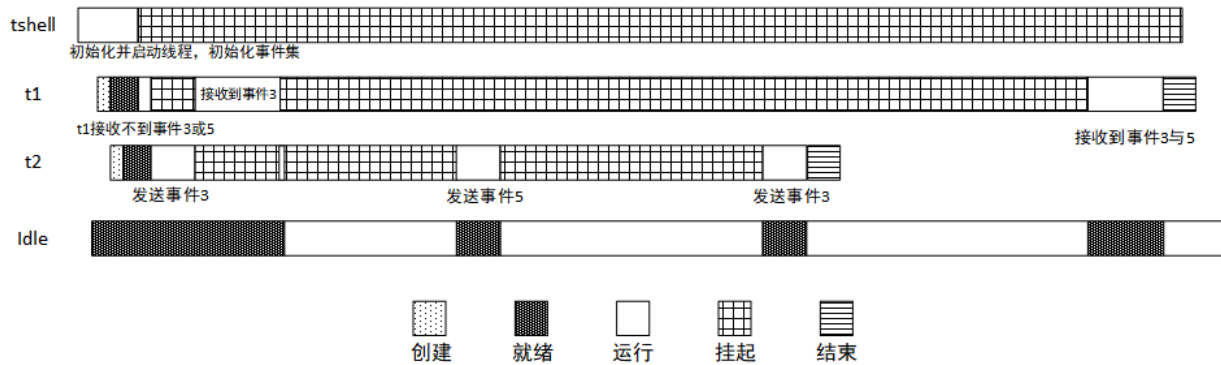


图 13.1: 运行过程

1. 在 tshell 线程中初始化一个事件集 event，初始化为先进先出型；并分别初始化、启动线程 thread1、thread2，优先级分别为 8、9；
2. 在操作系统的调度下，thread1 优先级高，首先被投入运行；thread1 开始运行后接收事件集 3 或 5，由于未接收到事件集 3 或 5，线程 thread1 挂起；
3. 随后操作系统调度 thread2 投入运行，thread2 发送事件 3，然后执行延时将自己挂起 200ms；
4. thread1 接收到事件 3，打印相关信息，并开始等待“事件 3 与事件 5”；
5. thread2 的延时时间到，发送事件 5；延时，发送事件 3；
6. 等待 thread1 的延时结束后，可以马上接收到“事件 3 与事件 5”，打印信息，结束运行；

13.2 事件集使用示例

示例代码通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

例子中初始化一个事件集，初始化两个静态线程。一个线程等待自己关心的事件的发生，另外一个线程发生事件。

以下定义了待创建线程需要用到的优先级，栈空间，时间片的宏，事件控制块。

```
#include <rtthread.h>
#define THREAD_PRIORITY      9
#define THREAD_TIMESLICE    5
#define EVENT_FLAG3 (1 << 3)
#define EVENT_FLAG5 (1 << 5)

/* 事件控制块 */
static struct rt_event event;
```

线程 thread1 的栈空间、线程控制块以及线程 thread1 的入口函数，共接收两次事件，第一次永久等待“事件 3 或事件 5”，第二次永久等待“事件 3 与事件 5”

```
ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口函数 */
```

```
static void thread1_rcv_event(void *param)
{
    rt_uint32_t e;

    /* 第一次接收事件，事件 3 或事件 5 任意一个可以触发线程 1，接收完后清除事件标志 */
    if (rt_event_rcv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                    RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                    RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR rcv event 0x%x\n", e);
    }

    rt_kprintf("thread1: delay 1s to prepare the second event\n");
    rt_thread_mdelay(1000);

    /* 第二次接收事件，事件 3 和事件 5 均发生时才可以触发线程 1，接收完后清除事件标志 */
    if (rt_event_rcv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                    RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                    RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: AND rcv event 0x%x\n", e);
    }
    rt_kprintf("thread1 leave.\n");
}
}
```

线程 `thread2` 的栈空间、线程控制块以及线程 `thread1` 的入口函数，发送 3 次事件，发送事件 3，延时 200ms；发送事件 5，延时 200ms；发送事件 3，结束。

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_send_event(void *param)
{
    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event5\n");
    rt_event_send(&event, EVENT_FLAG5);
    rt_thread_mdelay(200);

    rt_kprintf("thread2: send event3\n");
    rt_event_send(&event, EVENT_FLAG3);
    rt_kprintf("thread2 leave.\n");
}
}
```

事件的示例代码，初始化一个事件对象，初始化并启动线程 `thread1`、`thread2`，并将函数使用 `MSH_CMD_EXPORT` 导出命令。

```
int event_sample(void)
{
    rt_err_t result;

    /* 初始化事件对象 */
    result = rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                   "thread1",
                   thread1_recv_event,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),
                   THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
                   "thread2",
                   thread2_send_event,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),
                   THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

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

13.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `event_sample` 命令启动示例应用，示例输出结果如下：

```
msh >
msh >event_sample
thread2: send eventthread3
thread1: OR recv event 0x8
```



```
thread1: delay 1s to prepare the second event
msh >thread2: send event5
thread2: send eventthread3
thread2 leave.
thread1: AND recv event 0x28
thread1 leave.
```

例程演示了事件集的使用方法。**thread1** 前后两次接收事件，分别使用了“逻辑或”与“逻辑与”的方法。

第 14 章

邮箱的使用

邮箱是一种简单的线程间消息传递方式，特点是开销比较低，效率较高。在 **RT-Thread** 操作系统的实现中能够一次传递一个 4 字节大小的邮件，并且邮箱具备一定的存储功能，能够缓存一定数量的邮件数（邮件数由创建、初始化邮箱时指定的容量决定）。邮箱中一封邮件的最大长度是 4 字节，所以邮箱能够用于不超过 4 字节的消息传递。

参考：[文档中心——IPC 管理之邮箱](#)

14.1 代码设计

本例程源码为：`mailbox_sample.c`

为了体现使用邮箱来达到线程间的通信，本例程设计了 `thread1`、`thread2` 两个线程，优先级同为 10，设计了一个邮箱 `mbt`。

线程 `thread1` 每 100ms 尝试接收一次邮件，如果接收到邮件就将邮件内容打印出来。在接收到结束邮件时，打印邮件信息，线程结束。

线程 `thread2` 每 200ms 发送一次邮件，发送 10 次之后，发送结束邮件（线程 2 共发送 11 封邮件），线程运行结束。

通过本例程，用户可以清晰地了解到，线程在使用邮箱时候的线程调度。

整个运行过程如下图所示，下面以 `thread2` 开始运行时为开始时间，过程描述如下：

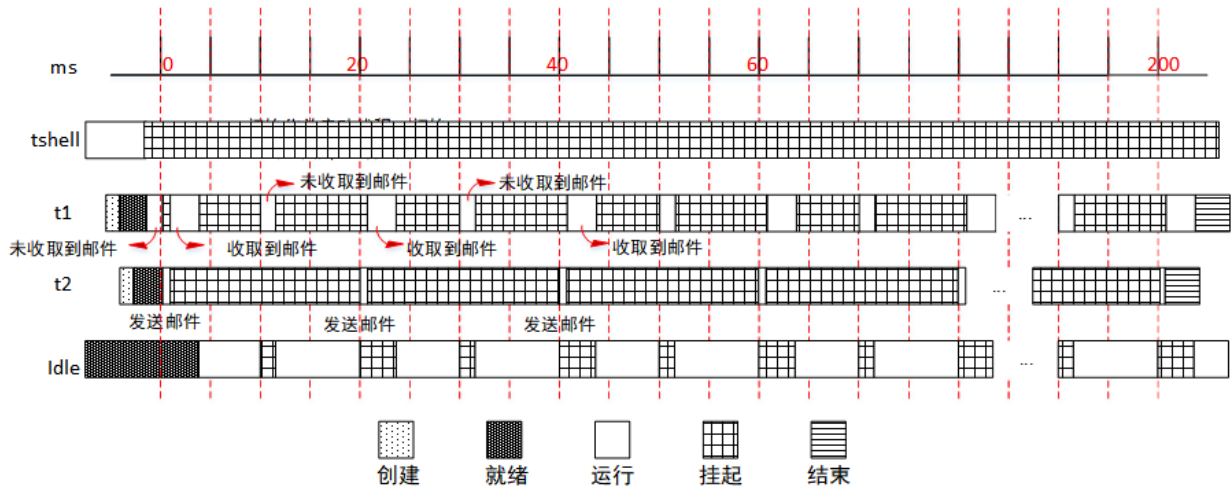


图 14.1: 运行过程

- (1) 在 `tshell` 线程中初始化一个邮箱 `mbt`, 采用 FIFO 方式进行线程等待; 初始化并启动线程 `thread1`、`thread2`, 优先级同为 10;
- (2) 在操作系统的调度下, `thread1` 首先被投入运行;
- (3) `thread1` 开始运行, 首先打印一段信息, 然后尝试获取邮件, 邮箱暂时没有邮件, `thread1` 挂起;
- (4) 随后操作系统调度 `thread2` 投入运行, 发送一封邮件, 随后进入 200ms 延时;
- (5) 此时线程 `thread1` 被唤醒, 接收到邮件, 继续打印一段信息, 然后进入 100ms 延时;
- (6) `thread2` 在发送 10 次邮件后, 发送一封结束内容的邮件, 线程结束。
- (7) `thread1` 一直接收邮件, 当接收到来自 `thread2` 的结束邮件后, 脱离邮箱, 线程结束。

14.1.1 源程序说明

示例代码通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

以下定义了线程需要用到的优先级, 栈空间, 时间片的宏, 邮箱控制块, 存放邮件的内存池、3 份邮件内容。

```
#include <rtthread.h>

#define THREAD_PRIORITY      10
#define THREAD_TIMESLICE    5

/* 邮箱控制块 */
static struct rt_mailbox mb;

/* 用于放邮件的内存池 */
static char mb_pool[128];

static char mb_str1[] = "I'm a mail!";
static char mb_str2[] = "this is another mail!";
```

```
static char mb_str3[] = "over";
```

线程 **thread1** 使用的栈空间、线程控制块，以及线程 **thread1** 的入口函数，每 100ms 收取一次邮件并打印邮件内容，当收取到结束邮件的时候，脱离邮箱，结束运行。

```
ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口 */
static void thread1_entry(void *parameter)
{
    char *str;

    while (1)
    {
        rt_kprintf("thread1: try to recv a mail\n");

        /* 从邮箱中收取邮件 */
        if (rt_mb_recv(&mb, (rt_uint32_t *)&str, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: get a mail from mailbox, the content:%s\n", str);
            if (str == mb_str3)
                break;

            /* 延时 100ms */
            rt_thread_mdelay(100);
        }
    }
    /* 执行邮箱对象脱离 */
    rt_mb_detach(&mb);
}
```

线程 **thread2** 使用的栈空间、线程控制块，以及线程 **thread2** 的入口函数，每 200ms 发送一封邮件，10 次后发送结束邮件，结束运行

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    rt_uint8_t count;

    count = 0;
    while (count < 10)
    {
        count ++;
```

```

    if (count & 0x1)
    {
        /* 发送 mb_str1 地址到邮箱中 */
        rt_mb_send(&mb, (rt_uint32_t)&mb_str1);
    }
    else
    {
        /* 发送 mb_str2 地址到邮箱中 */
        rt_mb_send(&mb, (rt_uint32_t)&mb_str2);
    }

    /* 延时 200ms */
    rt_thread_mdelay(200);
}

/* 发送邮件告诉线程 1，线程 2 已经运行结束 */
rt_mb_send(&mb, (rt_uint32_t)&mb_str3);
}

```

邮箱的示例代码，初始化了邮箱，初始化并启动了线程 thread1 与 thread2。并将函数使用 MSH_CMD_EXPORT 导出命令

```

int mailbox_sample(void)
{
    rt_err_t result;

    /* 初始化一个 mailbox */
    result = rt_mb_init(&mb,
                        "mbt",                /* 名称是 mbt */
                        &mb_pool[0],         /* 邮箱用到的内存池是 mb_pool */
                        sizeof(mb_pool) / 4,   /* 邮箱中的邮件数目，因为一封邮件占 4 字节 */
                        RT_IPC_FLAG_FIFO);     /* 采用 FIFO 方式进行线程等待 */

    if (result != RT_EOK)
    {
        rt_kprintf("init mailbox failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
                  "thread1",
                  thread1_entry,
                  RT_NULL,
                  &thread1_stack[0],
                  sizeof(thread1_stack),
                  THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,

```

```
        "thread2",
        thread2_entry,
        RT_NULL,
        &thread2_stack[0],
        sizeof(thread2_stack),
        THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);
    return 0;
}

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

14.2 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 **mailbox_sample** 命令启动示例应用，示例输出结果如下：

```
msh >
msh > mailbox_sample
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
msh > thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
...
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:over
```

第 15 章

消息队列的使用

消息队列能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

参考：[文档中心——IPC 管理之消息队列](#)

15.1 代码设计

本例程源码为：`msgq_sample.c`

为了体现使用消息队列来达到线程间的通信，本例程设计了 `thread1`、`thread2` 两个线程，优先级同为 25，设计了一个消息队列 `mqt`。

线程 `thread1` 每 50ms 从消息队列接收一次消息，并打印接收到的消息内容，在接收 20 次消息之后，将消息队列脱离、结束线程。

线程 `thread2` 每 10ms 向 `mqt` 消息队列依次发送 20 次消息，分别是消息“A”-“T”，第 9 次发送的是一个紧急消息“T”，发送 20 次后线程运行结束。（注，虽然设置的是 5ms，但是该工程设置的一个 OS Tick 是 10ms，是最小精度）。

通过本例程，用户可以清晰地了解到，线程在使用消息队列时候的线程调度。

整个运行过程如下图所示，OS Tick 为系统滴答时钟，下面以例程开始后第一个到来的 OS Tick 为第 1 个 OS Tick，过程描述如下：



图 15.1: 运行过程

- (1) 在 `tshell` 线程中初始化一个消息队列 `mqt`, 采用 FIFO 方式进行线程等待; 初始化并启动线程 `thread1`、`thread2`, 优先级同为 25;
- (2) 在操作系统的调度下, `thread1` 首先被投入运行, 尝试从消息队列获取消息, 消息队列暂时没有消息, 线程挂起;
- (3) 随后操作系统调度 `thread2` 投入运行, `thread2` 发送一个消息“A”, 并打印发送消息内容, 随后每 10ms 发送一条消息;
- (4) 此时线程 `thread1` 接收到消息, 打印消息内容“A”, 然后每 50ms 接收一次消息;
- (5) 在第 100ms 时, `thread1` 本应接收消息“C”, 但由于队列中有紧急消息, 所以 `thread1` 先接收紧急消息“I”, 之后再顺序接收其他消息。
- (6) `thread2` 发送 20 条消息后, 结束线程。
- (7) `thread1` 接收 20 条消息后, 结束线程。

15.2 消息队列使用示例

示例代码通过 `MSH_CMD_EXPORT` 将示例初始函数导出到 `msh` 命令, 可以在系统运行过程中, 通过在控制台输入命令来启动。

以下定义了待创建线程需要用到的优先级、时间片的宏, 消息队列控制块以及存放消息用到的内存池。

```
#include <rtthread.h>

#define THREAD_PRIORITY    25
#define THREAD_TIMESLICE  5

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static rt_uint8_t msg_pool[2048];
```


线程 **thread1** 使用的栈空间、线程控制块，以及线程 **thread1** 的入口函数，每 50ms 从消息队列中收取消息，并打印消息内容，20 次后结束。

```
ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* 线程 1 入口函数 */
static void thread1_entry(void *parameter)
{
    char buf = 0;
    rt_uint8_t cnt = 0;

    while (1)
    {
        /* 从消息队列中接收消息 */
        if (rt_mq_rcv(&mq, &buf, sizeof(buf), RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: rcv msg from msg queue, the content:%c\n", buf);
            if (cnt == 19)
            {
                break;
            }
        }
        /* 延时 50ms */
        cnt++;
        rt_thread_mdelay(50);
    }
    rt_kprintf("thread1: detach mq \n");
    rt_mq_detach(&mq);
}
```

线程 **thread2** 使用的栈空间、线程控制块，以及线程 **thread2** 的入口函数，每 5ms 向消息队列中发送消息，并打印消息内容，20 次后结束

```
ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;

/* 线程 2 入口 */
static void thread2_entry(void *parameter)
{
    int result;
    char buf = 'A';
    rt_uint8_t cnt = 0;

    while (1)
    {
        if (cnt == 8)
        {

```

```

    /* 发送紧急消息到消息队列中 */
    result = rt_mq_urgent(&mq, &buf, 1);
    if (result != RT_EOK)
    {
        rt_kprintf("rt_mq_urgent ERR\n");
    }
    else
    {
        rt_kprintf("thread2: send urgent message - %c\n", buf);
    }
}
else if (cnt >= 20) /* 发送 20 次消息之后退出 */
{
    rt_kprintf("message queue stop send, thread2 quit\n");
    break;
}
else
{
    /* 发送消息到消息队列中 */
    result = rt_mq_send(&mq, &buf, 1);
    if (result != RT_EOK)
    {
        rt_kprintf("rt_mq_send ERR\n");
    }

    rt_kprintf("thread2: send message - %c\n", buf);
}
buf++;
cnt++;
/* 延时 5ms */
rt_thread_mdelay(5);
}
}

```

消息队列的示例代码，初始化了一个消息队列，初始化并启动了 **thread1** 与 **thread2**。并将函数使用 **MSH_CMD_EXPORT** 导出命令。

```

/* 消息队列示例的初始化 */
int msgq_sample(void)
{
    rt_err_t result;

    /* 初始化消息队列 */
    result = rt_mq_init(&mq,
                        "mq",
                        &msg_pool[0],
                        1,
                        sizeof(msg_pool),
                        小 /*
                                /* 内存池指向 msg_pool */
                                /* 每个消息的大小是 1 字节 */
                                /* 内存池的大小是 msg_pool 的大

```

```

        RT_IPC_FLAG_FIFO);          /* 如果有多个线程等待，按照先来
        先得到的方法分配消息 */

    if (result != RT_EOK)
    {
        rt_kprintf("init message queue failed.\n");
        return -1;
    }

    rt_thread_init(&thread1,
        "thread1",
        thread1_entry,
        RT_NULL,
        &thread1_stack[0],
        sizeof(thread1_stack),
        THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread1);

    rt_thread_init(&thread2,
        "thread2",
        thread2_entry,
        RT_NULL,
        &thread2_stack[0],
        sizeof(thread2_stack),
        THREAD_PRIORITY, THREAD_TIMESLICE);
    rt_thread_startup(&thread2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(msgq_sample, msgq sample);

```

15.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `msgq_sample` 命令启动示例应用，示例输出结果如下：

```

msh >
msh > msgq_sample
msh > thread2: send message - A
thread1: rcv msg from msg queue, the content:A
thread2: send message - B
thread2: send message - C
thread2: send message - D
thread2: send message - E
thread1: rcv msg from msg queue, the content:B
thread2: send message - F

```

```
thread2: send message - G
thread2: send message - H
thread2: send urgent message - I
thread2: send message - J
thread1: recv msg from msg queue, the content:I
thread2: send message - K
thread2: send message - L
thread2: send message - M
thread2: send message - N
thread2: send message - O
thread1: recv msg from msg queue, the content:C
thread2: send message - P
thread2: send message - Q
thread2: send message - R
thread2: send message - S
thread2: send message - T
thread1: recv msg from msg queue, the content:D
message queue stop send, thread2 quit
thread1: recv msg from msg queue, the content:E
thread1: recv msg from msg queue, the content:F
thread1: recv msg from msg queue, the content:G
...
thread1: recv msg from msg queue, the content:T
thread1: detach mq
```

第 16 章

动态内存堆的使用

动态堆管理根据具体内存设备划分为以下三种情况，本例程针对于第一种情况，前提是要开启系统 heap 功能。

第一种是针对小内存块的分配管理（小堆内存管理算法），小内存管理算法主要针对系统资源比较少，一般用于小于 2MB 内存空间的系统。

第二种是针对大内存块的分配管理（slab 管理算法），slab 内存管理算法则主要是在系统资源比较丰富时，提供了一种近似多内存池管理算法的快速算法。

第三种是针对多内存块的分配情况（memheap 管理算法），memheap 方法适用于系统存在多个内存堆的情况，它可以将多个内存“粘贴”在一起，形成一个大的内存堆，用户使用起来会感到格外便捷。

16.1 代码设计

本例程源码为：dynmem_sample.c

例程设计一个动态的线程，这个线程会动态申请内存并释放，每次申请更大的内存，当申请不到的时候就结束。

16.2 动态内存堆示例

示例代码通过 MSH_CMD_EXPORT 将示例初始函数导出到 msh 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

以下定义了线程所用的优先级、栈大小以及时间片的宏。

```
#include <rtthread.h>

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

线程入口函数，一直申请内存，申请到之后就释放内存，每次会申请更大的内存，申请不到时，将结束，申请的内存大小信息也会打印出来。

```

/* 线程入口 */
void thread1_entry(void *parameter)
{
    int i;
    char *ptr = RT_NULL; /* 内存块的指针 */

    for (i = 0; ; i++)
    {
        /* 每次分配 (1 << i) 大小字节数的内存空间 */
        ptr = rt_malloc(1 << i);

        /* 如果分配成功 */
        if (ptr != RT_NULL)
        {
            rt_kprintf("get memory :%d byte\n", (1 << i));
            /* 释放内存块 */
            rt_free(ptr);
            rt_kprintf("free memory :%d byte\n", (1 << i));
            ptr = RT_NULL;
        }
        else
        {
            rt_kprintf("try to get %d byte memory failed!\n", (1 << i));
            return;
        }
    }
}

```

动态内存管理的示例代码，创建 thread1 并启动。并将函数使用 MSH_CMD_EXPORT 导出命令。

```

int dynmem_sample(void)
{
    rt_thread_t tid = RT_NULL;

    /* 创建线程 1 */
    tid = rt_thread_create("thread1",
                           thread1_entry, RT_NULL,
                           THREAD_STACK_SIZE,
                           THREAD_PRIORITY,
                           THREAD_TIMESLICE);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(dynmem_sample, dynmem sample);

```

16.3 编译运行

编译工程，然后下载运行。使用终端工具打开相应的 COM 口（波特率 115200），可以看到系统的启动日志，输入 `dynmem_sample` 命令启动示例应用，示例输出结果如下：

```
msh >
msh > dynmem_sample
msh > get memory :1 byte
free memory :1 byte
get memory :2 byte
free memory :2 byte
get memory :4 byte
free memory :4 byte
get memory :8 byte
free memory :8 byte
get memory :16 byte
free memory :16 byte
get memory :32 byte
free memory :32 byte
get memory :64 byte
free memory :64 byte
get memory :128 byte
free memory :128 byte
get memory :256 byte
...
get memory :16384 byte
free memory :16384 byte
get memory :32768 byte
free memory :32768 byte
get memory :65536 byte
free memory :65536 byte
try to get 131072 byte memory failed!
```

例程中分配内存成功并打印信息；当试图申请 131072 byte 即 128KB 内存时，由于 RAM 总大小只有 128K，而可用 RAM 小于 128K，所以分配失败。

第 17 章

内存池的使用

内存池是一种内存分配方式，用于分配大量大小相同的小内存块，它可以极大地加快内存分配与释放的速度，且能尽量避免内存碎片化。此外，**RT-Thread** 的内存池支持线程挂起功能，当内存池中无空闲内存块时，申请线程会被挂起，直到内存池中有新的可用内存块，再将挂起的申请线程唤醒。

参考：[文档中心——内存池](#)

17.1 代码设计

本例程的源码为：**memp_sample.c**，这个程序会创建一个静态的内存池对象，2 个动态线程。一个线程会试图从内存池中获得内存块，另一个线程释放内存块。

17.2 mempool 使用示例

示例代码通过 **MSH_CMD_EXPORT** 将示例初始函数导出到 **msh** 命令，可以在系统运行过程中，通过在控制台输入命令来启动。

```
#include <rtthread.h>

static rt_uint8_t *ptr[50];
static rt_uint8_t mempool[4096];
static struct rt_mempool mp;

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

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

/* 线程1入口 */
static void thread1_mp_alloc(void *parameter)
```



```

{
    int i;
    for (i = 0 ; i < 50 ; i++)
    {
        if (ptr[i] == RT_NULL)
        {
            /* 试图申请内存块50次，当申请不到内存块时，
               线程1挂起，转至线程2运行 */
            ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
            if (ptr[i] != RT_NULL)
                rt_kprintf("allocate No.%d\n", i);
        }
    }
}

/* 线程2入口，线程2的优先级比线程1低，应该线程1先获得执行。*/
static void thread2_mp_release(void *parameter)
{
    int i;

    rt_kprintf("thread2 try to release block\n");
    for (i = 0; i < 50 ; i++)
    {
        /* 释放所有分配成功的内存块 */
        if (ptr[i] != RT_NULL)
        {
            rt_kprintf("release block %d\n", i);
            rt_mp_free(ptr[i]);
            ptr[i] = RT_NULL;
        }
    }
}

int mempool_sample(void)
{
    int i;
    for (i = 0; i < 50; i++) ptr[i] = RT_NULL;

    /* 初始化内存池对象 */
    rt_mp_init(&mp, "mp1", &mempool[0], sizeof(mempool), 80);

    /* 创建线程1：申请内存池 */
    tid1 = rt_thread_create("thread1", thread1_mp_alloc, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

#ifdef RT_USING_SMP
    /* 绑定线程到同一个核上，避免启用多核时的输出混乱 */
    rt_thread_control(tid1, RT_THREAD_CTRL_BIND_CPU, (void*)0);
#endif
}

```

```

    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 创建线程2: 释放内存池*/
    tid2 = rt_thread_create("thread2", thread2_mp_release, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY + 1, THREAD_TIMESLICE);
#ifdef RT_USING_SMP
    /* 绑定线程到同一个核上, 避免启用多核时的输出混乱 */
    rt_thread_control(tid2, RT_THREAD_CTRL_BIND_CPU, (void*)0);
#endif
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    return 0;
}

/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(mempool_sample, mempool sample);

```

17.3 编译运行

编译工程, 然后下载运行。使用终端工具打开相应的 COM 口 (波特率 115200), 可以看到系统的启动日志, 输入 `mempool_sample` 命令启动示例应用, 示例输出结果如下:

```

msh >
msh >mempool_sample
msh >allocate No.0
allocate No.1
allocate No.2
allocate No.3
allocate No.4
allocate No.5
...
allocate No.45
allocate No.46
allocate No.47
thread2 try to release block
release block 0
allocate No.48
release block 1
allocate No.49
release block 2
release block 3
...
release block 49

```