
RT-THREAD ULOG 日志组件应用 笔记 - 进阶篇

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 ©2019



WWW.RT-THREAD.ORG

Tuesday 9th October, 2018

目录

目录	i
1 本文的目的和结构	1
1.1 本文的目的和背景	1
1.2 本文的结构	1
2 问题阐述	1
3 问题的解决	2
3.1 日志后端	2
3.2 异步日志	4
3.2.1. 异步模式 VS 同步模式	4
3.2.2. 异步模式的配置	5
3.2.3. 异步模式例程	5
3.3 日志过滤器（动态过滤）	6
3.3.1. 按模块的级别过滤	6
3.3.2. 按级别进行全局过滤	7
3.3.3. 按标签进行全局过滤	7
3.3.4. 按关键词进行全局过滤	7
3.3.5. 运行例程	7
3.4 系统异常时的使用	10
3.4.1. 断言	10
3.4.2. CmBacktrace	11
3.5 syslog 模式	11
3.5.1. syslog 配置	11
3.5.2. 日志的格式	12

	3.5.3.	使用方法	12
3.6		如何输出更加直观的日志	13
	3.6.1.	日志标签分类	13
	3.6.2.	合理利用日志级别	13
	3.6.3.	避免重复性冗余日志	13
	3.6.4.	开启更多的日志格式	14
	3.6.5.	关闭不重要的日志	14
4		常见问题	14
5		参考	14
5.1		本文所有相关的 API	14
	5.1.1.	API 列表	14
	5.1.2.	核心 API 详解	15
	5.1.3.	注册后端设备	15
	5.1.4.	异步模式下输出全部日志	15
	5.1.5.	按模块/标签的级别过滤日志	15
	5.1.6.	按级别过滤日志（全局）	16
	5.1.7.	按标签过滤日志（全局）	16
	5.1.8.	按关键词过滤日志（全局）	17

!!! abstract “摘要” 本应用笔记在《RT-Thread ulog 日志组件应用笔记 - 基础篇》的基础上，讲解 RT-Thread ulog 组件的高级用法和使用技巧。帮助开发者更加深入的了解 ulog，并提升日志调试效率。

1 本文的目的和结构

1.1 本文的目的和背景

在了解了《RT-Thread ulog 日志组件应用笔记 - 基础篇》之后，ulog 的基本功能都可以掌握了。为了让大家更好的玩转 ulog，这篇应用笔记会重点跟大家介绍 ulog 的高级功能及一些日志调试的经验和技巧。学会这些高级用法以后，开发者也能很大程度上提升日志调试的效率。

同时还会介绍 ulog 的高级模式：syslog 模式，这个模式能做到从前端 API 到日志格式对于 Linux syslog 的完全兼容，极大的方便从 Linux 上的迁移过来的软件。

1.2 本文的结构

本应用笔记将从以下几个方面来介绍 RT-Thread ulog 的高级应用：

- ulog 的后端
- ulog 的异步模式
- ulog 的日志过滤器
- ulog 的 syslog 模式
- 日志调试的一些技巧

2 问题阐述

本应用笔记将围绕下面几个问题来介绍 RT-Thread ulog 组件。

- ulog 支持的后端有哪些？
- 异步模式、日志过滤器及 syslog 模式如何使用？
- 系统异常（例如：hardfault）时，ulog 该如何处理？
- 如何输出更加直观的日志？

想要解决这些问题，就需要对 RT-Thread ulog 组件的高级功能有一定的认识与了解，同时结合实际例程动手实验，各种功能也将会在 qemu 平台上进行演示。

3 问题的解决

3.1 日志后端

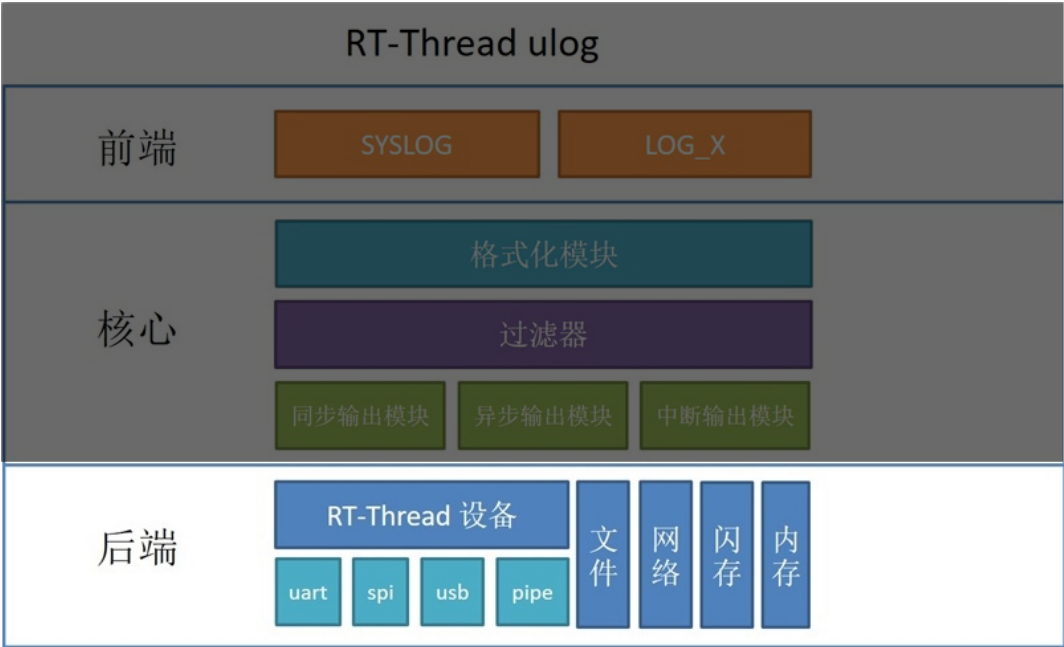


图 1: ulog 框架

讲到后端，我们来回顾下 ulog 的框架图。通过上图可以看出，ulog 是采用 前后端分离 的设计，前后端无依赖。并且支持的后端多样化，无论什么样后端，只要实现出来，都可以注册上去。

目前 ulog 已集成控制台后端，即传统的输出 `rt_kprintf` 打印日志的设备。后期 ulog 还会增加文件后端、Flash 后端、网络后端等后端的实现。当然，如果有特殊需求，用户也可以自己来实现后端。下面以控制台后端为例，简单介绍后端的实现方法及注册方法。

打开 `rt-thread/components/utilities/ulog/backend/console_be.c` 文件，可以看到大致有如下内容：

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

/* 定义控制台后端设备 */
static struct ulog_backend console;
/* 控制台后端输出函数 */
void ulog_console_backend_output(struct ulog_backend *backend,
    rt_uint32_t level, const char *tag, rt_bool_t is_raw, const char *log,
    size_t len)
{
```

```

...
/* 输出日志到控制台 */
...
}
/* 控制台后端初始化 */
int ulog_console_backend_init(void)
{
    /* 设定输出函数 */
    console.output = ulog_console_backend_output;
    /* 注册后端 */
    ulog_backend_register(&console, "console", RT_TRUE);

    return 0;
}
INIT_COMPONENT_EXPORT(ulog_console_backend_init);

```

通过上面的代码可以看出控制台后端的实现非常简单，这里实现了后端设备的 `output` 函数，并将该后端注册到 `ulog` 里，之后 `ulog` 的日志都会输出到控制台上。

如果要实现一个比较复杂的后端设备，此时就需要了解后端设备结构体，具体如下：

```

struct ulog_backend
{
    char name[RT_NAME_MAX];
    rt_bool_t support_color;
    void (*init) (struct ulog_backend *backend);
    void (*output)(struct ulog_backend *backend, rt_uint32_t level, const
        char *tag, rt_bool_t is_raw, const char *log, size_t len);
    void (*flush) (struct ulog_backend *backend);
    void (*deinit)(struct ulog_backend *backend);
    rt_slist_t list;
};

```

从这个结构体的角度可以看出，实现后端设备的要求如下：

- `name` 以及 `support_color` 属性可以通过 `ulog_backend_register` 函数在注册时传入；
- `output` 为后端具体的输出函数，所有后端都必须实现接口；
- `init/deinit` 可选择性实现，`init` 会在 `register` 时调用，`deinit` 会在 `ulog_deinit` 时调用；
- `flush` 也是可选择性实现，一些内部输出带缓存的后端需要必须实现该接口。比如一些带 `RAM` 缓存的文件系统。后端的 `flush` 一般会在断言、`hardfault` 等异常情况下由 `ulog_flush` 完成调用。

3.2 异步日志

在 ulog 中，默认的输出模式是同步模式，在很多场景下用户可能还需要异步模式。用户在调用日志输出 API 时，会将日志缓存到缓冲区中，会有专门负责日志输出的线程取出日志，然后输出到后端。

3.2.1. 异步模式 VS 同步模式

两个模式针对用户而言，在日志 API 使用上是没有差异的，因为 ulog 在底层处理上会有区分。两者的工作原理区别大致如下图所示：

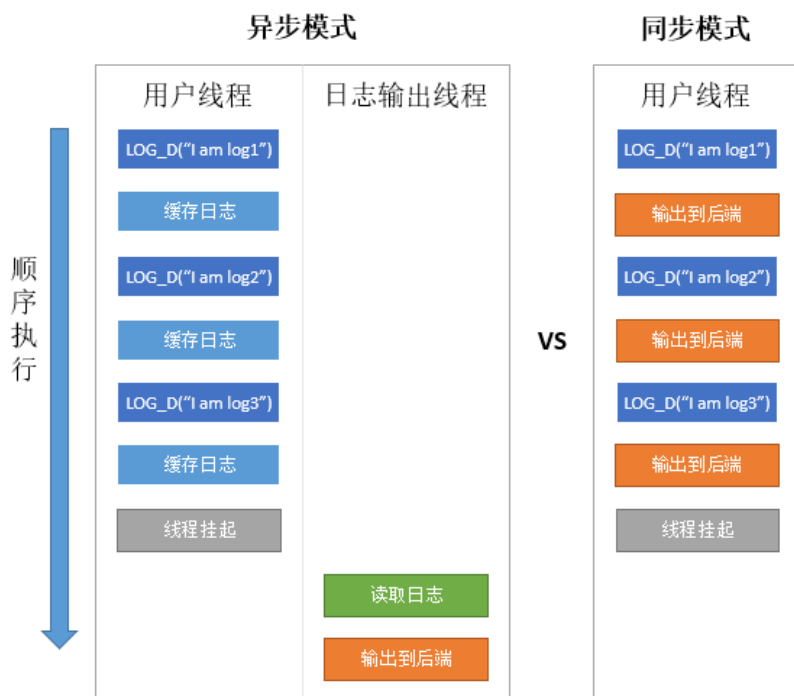


图 2: ulog 异步 VS 同步

再来看下异步模式的优缺点

- 优点:

- 首先日志输出时不会阻塞住当前线程，再加上有些后端输出速率低，所以使用同步输出模式可能影响当前线程的时序，异步模式不存在该问题。
- 其次，由于每个使用日志的线程省略了后端输出的动作，所以这些线程的堆栈开销可能也会减少，从这个角度也可以降低整个系统的资源占用。
- 同步模式下的中断日志只能输出到控制台后端，而异步模式下中断日志可以输出到所有后端去。

- 缺点: 首先异步模式需要日志缓冲区。再者异步日志的输出还需要有专门线程来完成，比如: idle 线程或者用户自定义的线程，用法上略显复杂。整体感觉异步模式资源占用会比同步模式要高。

3.2.2. 异步模式的配置

打开 env，进入 `rt-thread\bsp\qemu-vexpress-a9` 文件夹，打开 menuconfig 找到 ulog 的异步输出模式选项，打开的效果如下：

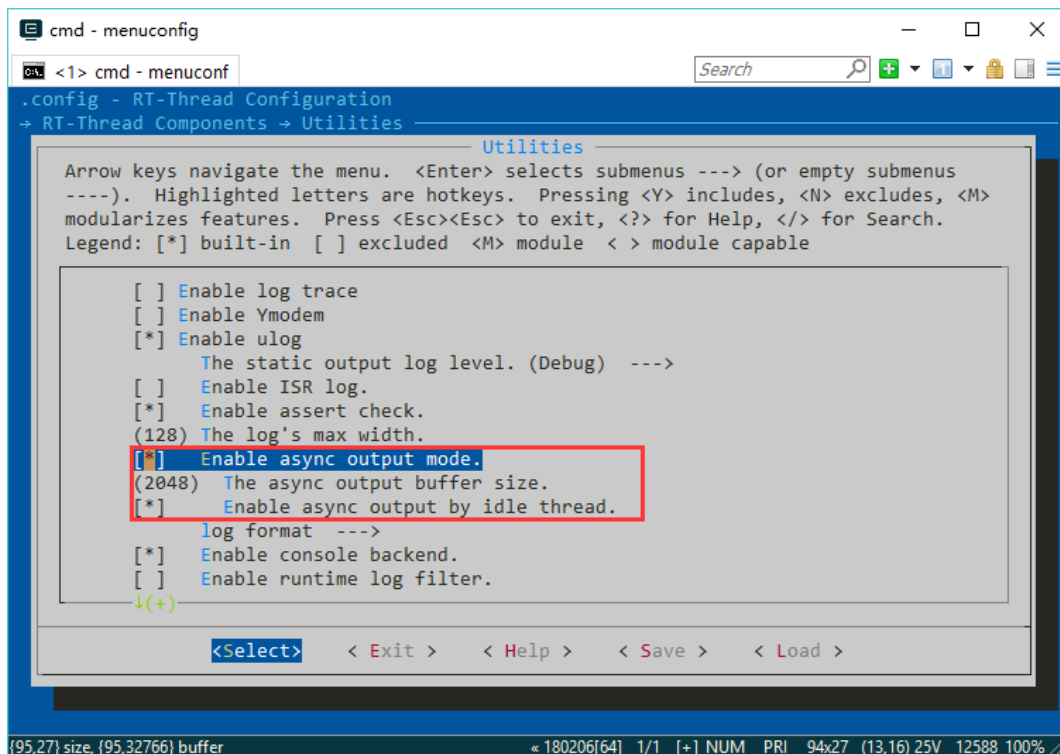


图 3: ulog 异步配置

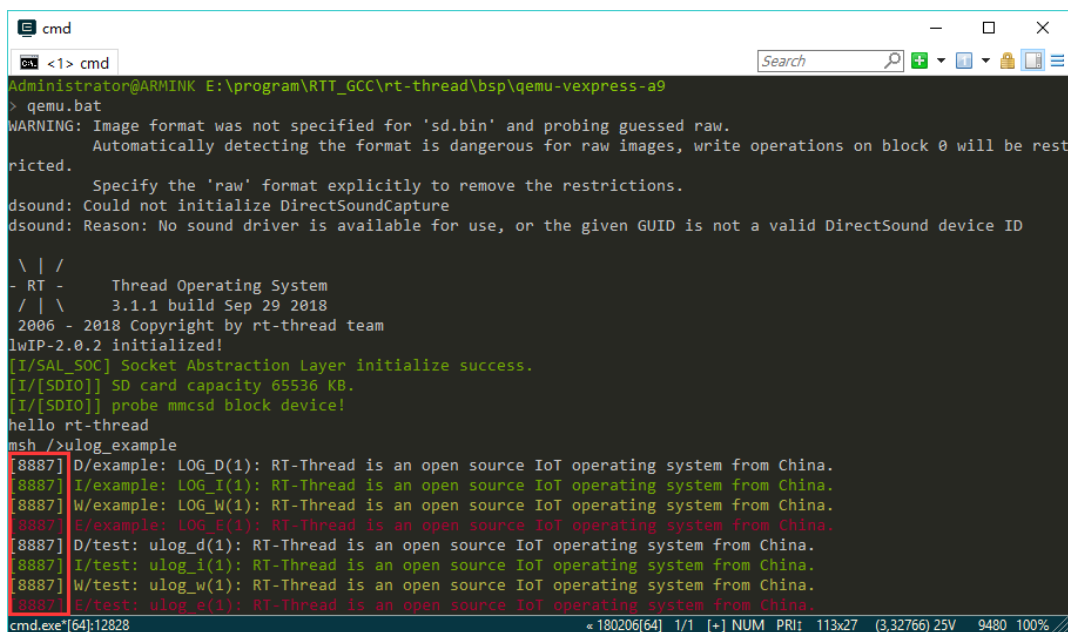
针对于异步模式，这里有两个选项

- The async output buffer size.: 异步缓冲区大小，默认为 2048
- Enable async output by idle thread.: 是否让 idle 线程作为异步日志输出线程，这样系统空闲时，可以自动完成日志的输出。该选项默认开启，如果想要修改其他线程，请先关闭此选项，然后在新线程中循环调用 `ulog_async_output()` 函数即可。

注意：使用 idle 线程输出时，务必确保 idle 线程堆栈大小至少为 384 字节。

3.2.3. 异步模式例程

- 保存异步输出选项配置
- 将 `rt-thread\examples\ulog_example.c` 拷贝至 `rt-thread\bsp\qemu-vexpress-a9\applications` 文件夹下
- 执行 `scons` 命令并等待编译完成
- 运行 `qemu.bat` 来打开 RT-Thread 的 qemu 模拟器
- 输入 `ulog_example` 命令，即可看到 ulog 例程运行结果，大致效果如下图



```

cmd
Administrator@ARMINK E:\program\RTT_GCC\rt-thread\bsp\qemu-vexpress-a9
> qemu.bat
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
dsound: Could not initialize DirectSoundCapture
dsound: Reason: No sound driver is available for use, or the given GUID is not a valid DirectSound device ID

\ | /
- RT -      Thread Operating System
/ | \      3.1.1 build Sep 29 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_S0C] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcsd block device!
hello rt-thread
msh />ulog_example
[8887] D/example: LOG_D(1): RT-Thread is an open source IoT operating system from China.
[8887] I/example: LOG_I(1): RT-Thread is an open source IoT operating system from China.
[8887] W/example: LOG_W(1): RT-Thread is an open source IoT operating system from China.
[8887] E/example: LOG_E(1): RT-Thread is an open source IoT operating system from China.
[8887] D/test: ulog_d(1): RT-Thread is an open source IoT operating system from China.
[8887] I/test: ulog_i(1): RT-Thread is an open source IoT operating system from China.
[8887] W/test: ulog_w(1): RT-Thread is an open source IoT operating system from China.
[8887] E/test: ulog_e(1): RT-Thread is an open source IoT operating system from China.
cmd.exe [64]:12828

```

图 4: ulog 异步例程

大家如果细心观察可以发现，开启异步模式后，这一些在代码上离得非常近的日志的时间信息几乎是相同的。但在同步模式下，日志使用用户线程来输出，由于日志输出要花一定时间，所以每条日志的时间会有一定的间隔。这里也充分说明了异步日志的输出效率很高，几乎不占用调用者的时间。

3.3 日志过滤器（动态过滤）

在 ulog 应用笔记的基础篇中有介绍过一些日志的静态过滤功能，静态过滤有其优点比如：节省资源，但很多时候，用户需要在软件运行时动态调整日志的过滤方式，这就可以使用到 ulog 的动态过滤器功能。使用动态过滤器功能需在 menuconfig 中开启 `Enable runtime log filter` 选项，该选项 默认关闭。

ulog 支持的动态过滤方式有以下 4 种，并且都有对应的 API 函数及 Finsh/MSH 命令，下面将会逐一介绍。

3.3.1. 按模块的级别过滤

- 函数: `int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level)`
- 命令格式: `ulog_tag_lvl <tag> <level>`

这里指的模块在应用笔记的基础篇中有介绍过，代表一类具有相同标签属性的日志代码。有些时候需要在运行时动态的修改某一个模块的日志输出级别，例如：关闭 `wifi` 模块的日志，此时设定 `wifi` 模块的日志输出级别为 `LOG_FILTER_LVL_SILENT` 即可。

3.3.2. 按级别进行全局过滤

- 函数: `void ulog_global_filter_lvl_set(rt_uint32_t level)`
- 命令格式: `ulog_lvl <level>`, level 分别为
 - 0: Assert
 - 3: Error
 - 4: Warning
 - 6: Info
 - 7: Debug

通过函数或者命令设定好全局的过滤级别以后, 低于设定级别的日志都将停止输出。

3.3.3. 按标签进行全局过滤

- 函数: `void ulog_global_filter_tag_set(const char *tag)`
- 命令格式: `ulog_tag [tag]`, tag 为空时, 则取消标签过滤

该过滤方式可以对所有日志执行按标签过滤, 只有 包含标签信息的日志才允许输出。

例如: 有 `wifi.driver`、`wifi.mgmt`、`audio.driver` 3 种标签的日志, 当设定过滤标签为 `wifi` 时, 只有标签为 `wifi.driver` 及 `wifi.mgmt` 的日志会输出。同理, 当设置过滤标签为 `driver` 时, 只有标签为 `wifi.driver` 及 `audio.driver` 的日志会输出。

3.3.4. 按关键词进行全局过滤

- 函数: `void ulog_global_filter_kw_set(const char *keyword)`
- 命令格式: `ulog_kw [keyword]`, keyword 为空时, 则取消关键词过滤

该过滤方式可以对所有日志执行按关键词过滤, 包含关键词信息的日志才允许输出。

3.3.5. 运行例程

依然是在 qemu BSP 中执行, 首先在 menuconfig 开启动态过滤, 然后保存配置并编译、运行例程, 在日志输出约 20 次后, 会执行 `ulog_example.c` 里对应的如下过滤代码:

```
if (count == 20)
{
    /* Set the global filter level is INFO. All of DEBUG log will stop
       output */
    ulog_global_filter_lvl_set(LOG_LVL_INFO);
    /* Set the test tag's level filter's level is ERROR. The DEBUG, INFO,
       WARNING log will stop output. */
    ulog_tag_lvl_filter_set("test", LOG_LVL_ERROR);
}
...
```

此时全局的过滤级别由于被设定到了 INFO 级别，所以无法再看到比 INFO 级别低的日志。同时，又将 `test` 标签的日志输出级别设定为 ERROR，此时 `test` 标签里比 ERROR 低的日志也都停止输出了。在每条日志里都有当前日志输出次数的计数值，对比的效果如下：

```

cmd - qemu.bat
<1> cmd - qemu.bat
[4191] D/example: LOG_D(20): RT-Thread is an open source IoT operating system from China.
[4191] I/example: LOG_I(20): RT-Thread is an open source IoT operating system from China.
[4191] W/example: LOG_W(20): RT-Thread is an open source IoT operating system from China.
[4191] E/example: LOG_E(20): RT-Thread is an open source IoT operating system from China.
[4191] D/test: ulog_d(20): RT-Thread is an open source IoT operating system from China.
[4191] I/test: ulog_i(20): RT-Thread is an open source IoT operating system from China.
[4191] W/test: ulog_w(20): RT-Thread is an open source IoT operating system from China.
[4191] E/test: ulog_e(20): RT-Thread is an open source IoT operating system from China.
[4278] I/example: LOG_I(21): RT-Thread is an open source IoT operating system from China.
[4278] W/example: LOG_W(21): RT-Thread is an open source IoT operating system from China.
[4278] E/example: LOG_E(21): RT-Thread is an open source IoT operating system from China.
[4278] E/test: ulog_e(21): RT-Thread is an open source IoT operating system from China.
[4376] I/example: LOG_I(22): RT-Thread is an open source IoT operating system from China.
[4376] W/example: LOG_W(22): RT-Thread is an open source IoT operating system from China.
[4376] E/example: LOG_E(22): RT-Thread is an open source IoT operating system from China.
[4376] E/test: ulog_e(22): RT-Thread is an open source IoT operating system from China.
[4399] I/example: LOG_I(23): RT-Thread is an open source IoT operating system from China.
[4399] W/example: LOG_W(23): RT-Thread is an open source IoT operating system from China.
[4399] E/example: LOG_E(23): RT-Thread is an open source IoT operating system from China.
[4399] E/test: ulog_e(23): RT-Thread is an open source IoT operating system from China.
[4454] I/example: LOG_I(24): RT-Thread is an open source IoT operating system from China.
[4454] W/example: LOG_W(24): RT-Thread is an open source IoT operating system from China.
[4454] E/example: LOG_E(24): RT-Thread is an open source IoT operating system from China.
[4454] E/test: ulog_e(24): RT-Thread is an open source IoT operating system from China.
[4514] I/example: LOG_I(25): RT-Thread is an open source IoT operating system from China.
[4514] W/example: LOG_W(25): RT-Thread is an open source IoT operating system from China.
[4514] E/example: LOG_E(25): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe[32]:13844  180206[64] 1/1 [+] NUM PRI: 113x27 (7,32766) 25V 9480 100%
  
```

图 5: ulog 过滤器例程 20

在日志输出约 30 次后，会执行 `ulog_example.c` 里对应的如下过滤代码：

```

...
else if (count == 30)
{
    /* Set the example tag's level filter's level is
       LOG_FILTER_LVL_SILENT, the log enter silent mode. */
    ulog_tag_lvl_filter_set("example", LOG_FILTER_LVL_SILENT);
    /* Set the test tag's level filter's level is WARNING. The DEBUG,
       INFO log will stop output. */
    ulog_tag_lvl_filter_set("test", LOG_LVL_WARNING);
}
...
  
```

此时又新增了 `example` 模块的过滤器，并且是将这个模块的所有日志都停止输出，所以接下来将看不到该模块日志。同时，又将 `test` 标签的日志输出级别降低为 WARNING，此时就只能看到 `test` 标签的 WARNING 与 ERROR 级别日志。效果如下：

```

cmd - qemu.bat
[4666] I/example: LOG_I(30): RT-Thread is an open source IoT operating system from China.
[4666] W/example: LOG_W(30): RT-Thread is an open source IoT operating system from China.
[4666] E/example: LOG_E(30): RT-Thread is an open source IoT operating system from China.
[4666] E/test: ulog_e(30): RT-Thread is an open source IoT operating system from China.
[4729] W/test: ulog_w(31): RT-Thread is an open source IoT operating system from China.
[4729] E/test: ulog_e(31): RT-Thread is an open source IoT operating system from China.
[4793] W/test: ulog_w(32): RT-Thread is an open source IoT operating system from China.
[4793] E/test: ulog_e(32): RT-Thread is an open source IoT operating system from China.
[4885] W/test: ulog_w(33): RT-Thread is an open source IoT operating system from China.
[4885] E/test: ulog_e(33): RT-Thread is an open source IoT operating system from China.
[4980] W/test: ulog_w(34): RT-Thread is an open source IoT operating system from China.
[4980] E/test: ulog_e(34): RT-Thread is an open source IoT operating system from China.
[5011] W/test: ulog_w(35): RT-Thread is an open source IoT operating system from China.
[5011] E/test: ulog_e(35): RT-Thread is an open source IoT operating system from China.
[5098] W/test: ulog_w(36): RT-Thread is an open source IoT operating system from China.
[5098] E/test: ulog_e(36): RT-Thread is an open source IoT operating system from China.
[5147] W/test: ulog_w(37): RT-Thread is an open source IoT operating system from China.
[5147] E/test: ulog_e(37): RT-Thread is an open source IoT operating system from China.
[5184] W/test: ulog_w(38): RT-Thread is an open source IoT operating system from China.
[5184] E/test: ulog_e(38): RT-Thread is an open source IoT operating system from China.
[5215] W/test: ulog_w(39): RT-Thread is an open source IoT operating system from China.
[5215] E/test: ulog_e(39): RT-Thread is an open source IoT operating system from China.
[5286] W/test: ulog_w(40): RT-Thread is an open source IoT operating system from China.
[5287] E/test: ulog_e(40): RT-Thread is an open source IoT operating system from China.
[5311] D/test: ulog_d(41): RT-Thread is an open source IoT operating system from China.
[5311] I/test: ulog_i(41): RT-Thread is an open source IoT operating system from China.
[5311] W/test: ulog_w(41): RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe[32]:13844
  
```

图 6: ulog 过滤器例程 30

在日志输出约 40 次后，会执行 ulog_example.c 里对应的如下过滤代码：

```

...
else if (count == 40)
{
    /* Set the test tag's level filter's level is LOG_FILTER_LVL_ALL. All
       level log will resume output. */
    ulog_tag_lvl_filter_set("test", LOG_FILTER_LVL_ALL);
    /* Set the global filter level is LOG_FILTER_LVL_ALL. All level log
       will resume output */
    ulog_global_filter_lvl_set(LOG_FILTER_LVL_ALL);
}
  
```

此时将 `test` 模块的日志输出级别调整为 `LOG_FILTER_LVL_ALL`，即不再过滤该模块任何级别的日志。同时，又将全局过滤级别设定为 `LOG_FILTER_LVL_ALL`，所以接下来 `test` 模块的全部日志将恢复输出。效果如下：

图 7: ulog 过滤器例程 40

3.4 系统异常时的使用

由于 ulog 的异步模式具有缓存机制，注册进来的后端内部也可能具有缓存。如果系统出现了 hardfault、断言等错误情况，但缓存中还有日志没有输出出来，这可能会导致日志丢失的问题，对于查找异常的原因会无从入手。

针对这种场景，ulog 提供了统一的日志 flush 函数：`void ulog_flush(void)`，当出现异常时，输出异常信息日志时，同时再调用该函数，即可保证缓存中剩余的日志也能够输出到后端中去。

下面以 RT-Thread 的断言及 CmBacktrace 进行举例：

3.4.1. 断言

RT-Thread 的断言支持断言回调 (hook)，我们定义一个类似如下的断言 hook 函数，然后通过 `rt_assert_set_hook(rtt_user_assert_hook);` 函数将其设置到系统中即可。

```
static void rtt_user_assert_hook(const char* ex, const char* func,
                                rt_size_t line)
{
    rt_enter_critical();

    ulog_output(LOG_LVL_ASSERT, "rtt", "(%s) has assert failed at %s:%ld."
                , ex, func, line);

    /* flush all log */
    ulog_flush();
    while(1);
}
```

```
}
```

3.4.2. CmBacktrace

CmBacktrace 是一个 ARM Cortex-M 系列 MCU 的错误诊断库,它也有对应 RT-Thread 软件包,并且最新版的软件包已经做好了针对于 ulog 的适配。里面适配代码位于 `cmb_cfg.h` :

```
...
/* print line, must config by user */
#include <rtthread.h>
#ifndef RT_USING_ULOG
#define cmb_println(...)          rt_kprintf(__VA_ARGS__);rt_kprintf
    ("\r\n")
#else
#include <ulog.h>
#define cmb_println(...)          ulog_e("cmb", __VA_ARGS__);
    ulog_flush()
#endif /* RT_USING_ULOG */
...
```

由此可以看出,当启用了 ulog 以后, CmBacktrace 的每一条日志输出时都会使用错误级别,并且会同时执行 `ulog_flush`, 用户无需再做任何修改。

3.5 syslog 模式

在 Unix 类操作系统上, syslog 广泛应用于系统日志。syslog 常见的后端有文件和网络, syslog 日志可以记录在本地文件中, 也可以通过网络发送到接收 syslog 的服务器。

ulog 提供了 syslog 模式的支持, 不仅仅前端 API 与 syslog API 完全一致, 日志的格式也符合 RFC 标准。但需要注意的是, 在开启 syslog 模式后, 不管使用哪一种日志输出 API, 整个 ulog 的日志输出格式都会采用 syslog 格式。

3.5.1. syslog 配置

只需要开启 `Enable syslog format log and API`. 选项即可。

3.5.2. 日志的格式

ulog syslog 格式



图 8: ulog syslog 格式

如上图所示，ulog syslog 日志格式分为下面 4 个部分：

- **PRI**：PRI 部分由尖括号包含的一个数字构成，这个数字包含了程序模块（Facility）、严重性（Severity）信息，是由 Facility 乘以 8，然后加上 Severity 得来。Facility 和 Severity 由 syslog 函数的入参传入，具体数值详见 syslog.h；
- **Header**：Header 部分主要是时间戳，指示当前日志的时间；
- **TAG**：当前日志的标签，可以通过 openlog 函数入参传入，如果不指定将会使用 rtt 作为默认标签；
- **Content**：日志的具体内容。

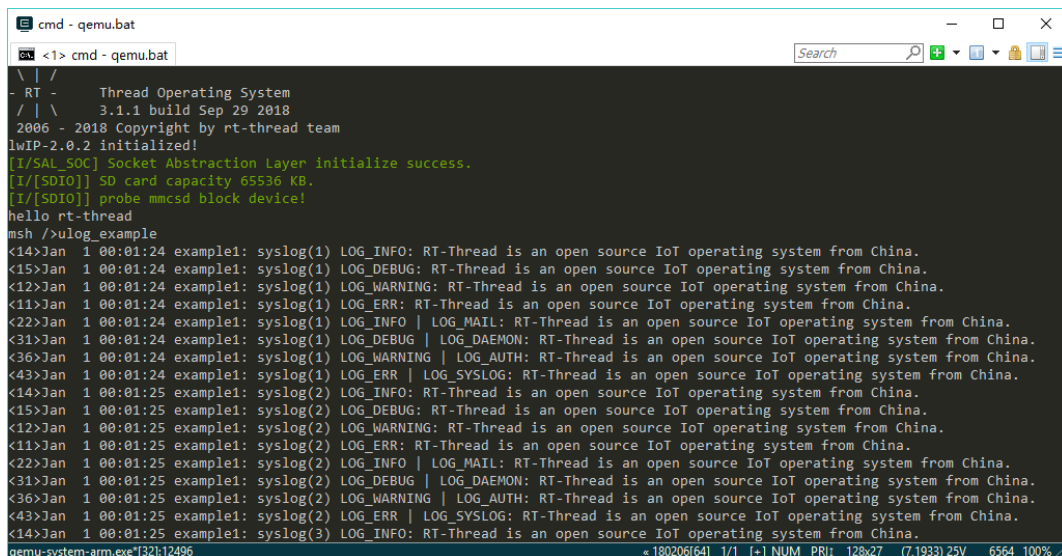
3.5.3. 使用方法

使用前需要在 menuconfig 中开启 syslog 选项，主要常用的 API 有：

- 打开 syslog: void openlog(const char *ident, int option, int facility)
- 输出 syslog 日志: void syslog(int priority, const char *format, ...)

提示：调用 openlog 是可选的。如果不调用 openlog，则在第一次调用 syslog 时，自动调用 openlog

syslog() 函数的使用方法也非常简单，其入参格式与 printf 函数一致。在 ulog_example.c 中也有 syslog 的例程，在 qemu 中的运行效果大致如下：



```
cmd - qemu.bat
<1> cmd - qemu.bat
- RT - Thread Operating System
/ | \ 3.1.1 build Sep 29 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/SAL_SOC] Socket Abstraction Layer initialize success.
[I/[SDIO]] SD card capacity 65536 KB.
[I/[SDIO]] probe mmcblk0 block device!
hello rt-thread
msh />ulog_example
<14>Jan 1 00:01:24 example1: syslog(1) LOG_INFO: RT-Thread is an open source IoT operating system from China.
<15>Jan 1 00:01:24 example1: syslog(1) LOG_DEBUG: RT-Thread is an open source IoT operating system from China.
<12>Jan 1 00:01:24 example1: syslog(1) LOG_WARNING: RT-Thread is an open source IoT operating system from China.
<11>Jan 1 00:01:24 example1: syslog(1) LOG_ERR: RT-Thread is an open source IoT operating system from China.
<22>Jan 1 00:01:24 example1: syslog(1) LOG_INFO | LOG_MAIL: RT-Thread is an open source IoT operating system from China.
<31>Jan 1 00:01:24 example1: syslog(1) LOG_DEBUG | LOG_DAEMON: RT-Thread is an open source IoT operating system from China.
<36>Jan 1 00:01:24 example1: syslog(1) LOG_WARNING | LOG_AUTH: RT-Thread is an open source IoT operating system from China.
<43>Jan 1 00:01:24 example1: syslog(1) LOG_ERR | LOG_SYSLOG: RT-Thread is an open source IoT operating system from China.
<14>Jan 1 00:01:25 example1: syslog(2) LOG_INFO: RT-Thread is an open source IoT operating system from China.
<15>Jan 1 00:01:25 example1: syslog(2) LOG_DEBUG: RT-Thread is an open source IoT operating system from China.
<12>Jan 1 00:01:25 example1: syslog(2) LOG_WARNING: RT-Thread is an open source IoT operating system from China.
<11>Jan 1 00:01:25 example1: syslog(2) LOG_ERR: RT-Thread is an open source IoT operating system from China.
<22>Jan 1 00:01:25 example1: syslog(2) LOG_INFO | LOG_MAIL: RT-Thread is an open source IoT operating system from China.
<31>Jan 1 00:01:25 example1: syslog(2) LOG_DEBUG | LOG_DAEMON: RT-Thread is an open source IoT operating system from China.
<36>Jan 1 00:01:25 example1: syslog(2) LOG_WARNING | LOG_AUTH: RT-Thread is an open source IoT operating system from China.
<43>Jan 1 00:01:25 example1: syslog(2) LOG_ERR | LOG_SYSLOG: RT-Thread is an open source IoT operating system from China.
<14>Jan 1 00:01:25 example1: syslog(3) LOG_INFO: RT-Thread is an open source IoT operating system from China.
qemu-system-arm.exe[32]:12496
```

图 9: ulog syslog 例程

3.6 如何输出更加直观的日志

有了日志工具后，如果使用不当，也会造成日志被滥用、日志信息无法突出重点等问题。这里重点与大家分享下日志组件在使用时的一些技巧，让日志信息更加直观。主要关注点有：

3.6.1. 日志标签分类

合理利用标签功能，每个模块代码在使用日志前，先明确好模块、子模块名称。这样也能让日志在最开始阶段就做好分类，为后期日志过滤也做好了准备。

3.6.2. 合理利用日志级别

刚开始使用日志库时，大家会经常遇到警告与错误日志无法区分，信息与调试日志无法区分，导致日志级别选择不合适。一些重要日志可能看不到，不重要的日志满天飞等问题。所以，在使用前务必仔细阅读《RT-Thread ulog 日志组件应用笔记 - 基础篇》的日志级别章节，针对各个级别划分，里面有明确的标准。

3.6.3. 避免重复性冗余日志

在一些情况下会出现代码的重复调用或者循环执行，多次输出相同、相似的日志问题。这样的日志不仅会占用很大的系统资源，还会影响开发人员对于问题的定位。所以，在遇到这种情况时，建议增加对于重复性日志特殊处理，比如：让上层来输出一些业务有关的日志，底层只返回具体结果状态；同一个时间点下相同的日志，是否可以增加去重处理，在错误状态没有变化时，只输出一次等等。

3.6.4. 开启更多的日志格式

ulog 默认的日志格式中没有开启时间戳及线程信息。这两个日志信息，在 RTOS 上挺实用。它们能帮助开发者直观的了解各个日志的运行时间点、时间差，还能清晰的看到是在哪个线程执行当前代码。所以如果条件允许，还是建议开启。

3.6.5. 关闭不重要的日志

ulog 提供了多种维度的日志开关、过滤的功能，完全能够做到精细化控制，所以如果在调试某个功能模块时，可以适当关闭其他无关模块的日志输出，这样就可以聚焦在当前调试的模块上，更多关于日志过滤功能请阅读《RT-Thread ulog 日志组件应用笔记 - 基础篇》设定级别的分类章节及本应用笔记的日志过滤器章节。

4 常见问题

- 1、运行出现警告提示: Warning: There is no enough buffer for saving async log, please increase the ULOG_ASYNC_OUTPUT_BUF_SIZE option.

当遇到该提示时，说明了在异步模式下的缓冲区出现了溢出的情况，这会导致一部分日志丢失，增大 ULOG_ASYNC_OUTPUT_BUF_SIZE 选项可以解决该问题。

- 2、编译时提示: The idle thread stack size must more than 384 when using async output by idle (ULOG_ASYNC_OUTPUT_BY_IDLE)

在使用 idle 线程作为输出线程时，idle 线程的堆栈大小需要提高，这也取决于具体的后端设备，例如：控制台后端时，idle 线程至少得 384 字节。

5 参考

5.1 本文所有相关的 API

5.1.1. API 列表

API	位置
rt_err_t ulog_backend_register(ulog_backend_t backend, const char *name, rt_bool_t support_color)	ulog.c
void ulog_async_output(void)	ulog.c
int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level)	ulog.c
void ulog_global_filter_lvl_set(rt_uint32_t level)	ulog.c
void ulog_global_filter_tag_set(const char *tag)	ulog.c

API	位置
void ulog_global_filter_kw_set(const char *keyword)	ulog.c
void openlog(const char *ident, int option, int facility)	syslog.c
void syslog(int priority, const char *format, ...)	syslog.c
int setlogmask(int mask)	syslog.c

5.1.2. 核心 API 详解

5.1.3. 注册后端设备

```
rt_err_t ulog_backend_register(ulog_backend_t backend, const char *name,
                               rt_bool_t support_color)
```

将后端设备注册到 ulog 中，注册前确保后端设备结构体中的函数成员已设置。

参数	描述
backend	AT 客户端使用设备名称
name	AT 客户端最大支持接收数据长度
support_color	是否支持彩色日志
返回	描述
>=0	成功

5.1.4. 异步模式下输出全部日志

```
void ulog_async_output(void)
```

在异步模式下，如果想要使用其他非 idle 线程作为日志输出线程时，则需要在输出线程中循环调用该 API，输出缓冲区中日志至所有的后端设备

5.1.5. 按模块/标签的级别过滤日志

```
int ulog_tag_lvl_filter_set(const char *tag, rt_uint32_t level)
```

通过这个 API 可以按照标签的级别来过滤日志，例如：

```
/* 停止输出 wifi.driver 模块的日志 */
ulog_tag_lvl_filter_set("wifi.driver", LOG_FILTER_LVL_SILENT);
/* 停止 wifi.mgmt 模块 INFO 级别以下日志的输出 */
```

```
ulog_tag_lvl_filter_set("wifi.mgmt", LOG_LVL_INFO);
```

参数	描述
tag	日志的标签
level	设定的日志级别，详见 ulog_def.h
返回	描述
>=0	成功
-5	失败，没有足够的内存

5.1.6. 按级别过滤日志（全局）

```
void ulog_global_filter_lvl_set(rt_uint32_t level)
```

设定全局的日志过滤器级别，低于这个级别的日志都将停止输出。可设定的级别包括：

级别	名称
LOG_LVL_ASSERT	断言
LOG_LVL_ERROR	错误
LOG_LVL_WARNING	警告
LOG_LVL_INFO	信息
LOG_LVL_DBG	调试
LOG_FILTER_LVL_SILENT	静默（停止输出）
LOG_FILTER_LVL_ALL	全部

参数	描述
level	设定的级别，详见上面的表格，也可见 ulog_def.h

5.1.7. 按标签过滤日志（全局）

```
void ulog_global_filter_tag_set(const char *tag)
```

设定全局的日志过滤器标签，只有日志的标签内容中 包含该设定字符串时，才会允许输出。

参数	描述
tag	设定的过滤标签

5.1.8. 按关键词过滤日志（全局）

```
void ulog_global_filter_kw_set(const char *keyword)
```

设定全局的日志过滤器关键词，只有内容中包含该关键词的日志才会允许输出。

参数	描述
keyword	设定的过滤关键词