User Manual

AWTK 移植及裁剪指南

用户手册

V1.0.1 Date:2025/10/31

类别	内容
关键词	AWTK、移植、裁剪、嵌入式
摘要	本文主要介绍了如何把 AWTK 移植到各种嵌入式平台上，并且还介绍了如何裁剪 AWTK。

修订历史

版本	日期	原因
1.0.1	2023/12/26	• 第七章硬件加速新增章节：图像专用内存。
1.0.0	2022/06/07	• first implement

目 录

文档导读

本文主要介绍了如何把 AWTK 移植到各种嵌入式平台上，并且还将说明 AWTK 的代码结构、堆栈需求以及相关功能模块等各种信息，帮助读者深入了解 AWTK。

第一部分

介绍篇

1. 简介

1.1 本文的目的

AWTK 全称 Toolkit AnyWhere，是 ZLG 开发的开源 GUI 引擎，详情可前往 官网[1] 查看。AWTK 的源码可前往 GitHub 仓库[2] 下载。

本文是为了介绍如何把 AWTK 移植到各种嵌入式平台上而编写的，并且还将说明 AWTK 代码结构、堆栈需求以及相关功能模块等各种信息，帮助读者全方面认知 AWTK。

本文基于 AWTK 1.6.1 编写，由于不同版本 AWTK 的代码结构和资源有所不同，所以其他版本可能会存在差异。

1.2 开发环境（编译器）

需要支持 gnu99 版本或者以上的 C/C++ 编译器。

1.3 AWTK 主要运行流程

启动 AWTK 程序时，运行流程如下图所示，其中标红的函数是在移植 AWTK 时需重点关注的函数，它们通常需要根据实际平台实现或者重载：

注: 流程图中的 platform_disaptch_input 函数是在使用 awtk/src/main_loop/main_loop_raw.inc 文件实现设备事件分发时才需要重载，具体方法详见下文第 5 章。

1、main 函数

AWTK 程序启动时，默认调用 AWTK 内置的 main 函数，函数实现请参考：awtk/src/awtk_main.inc，它会依次调用下列主要函数：

(1) tk_init 函数：初始化 AWTK。

(2) assets_init 函数：加载项目资源。

(3) application_init 函数：初始化应用程序，通常由用户实现。

(4) tk_run 函数：启动 AWTK GUI 主循环（main_loop）。

(5) application_exit 函数：退出应用程序，用于释放应用程序相关资源，通常由用户实现。

如果用户想要使用系统 main 函数，可以在 awtk_main.inc 文件前定义宏 USE_GUI_MAIN，并且在系统 main 函数中调用 gui_app_start 函数启动 AWTK，该函数与 AWTK 内置 main 函数的实现类似，示例代码如下：

```
#define USE_GUI_MAIN 1   /* 使用系统 main 函数时，请定义本宏*/

#include "awtk_main.inc"
int main(int argc, char* argv[]) {
  return gui_app_start(LCD_WIDTH, LCD_HEIGHT); /* 启动 AWTK（需指定 LCD 的宽高） */
}
```

[1] https://www.zlg.cn/index/pub/awtk.html
[2] https://github.com/zlgopen/awtk

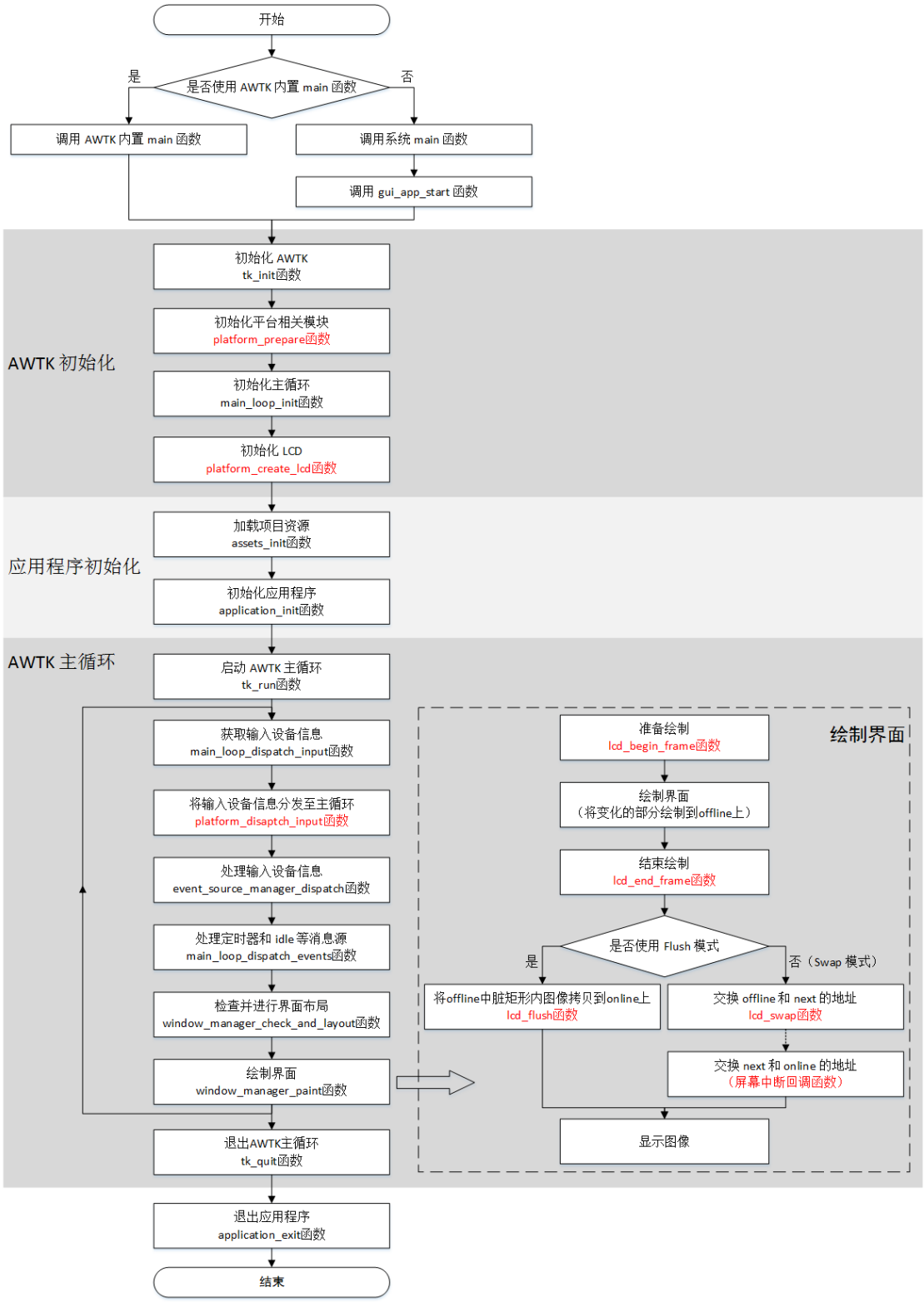

图 1.1　AWTK 运行流程图

2、tk_init 函数

tk_init 函数主要用于初始化 AWTK，它会依次调用下列主要函数：

(1) platform_prepare 函数：初始化平台相关的准备模块，比如内存管理器和时间管理器等。

(2) main_loop_init 函数：初始化 AWTK 的主循环，其中会调用 platform_create_lcd 函数初始化 LCD，通常用户会在该函数中设置 LCD 相关的 Framebuffer。

(3) 完成其他的一些初始化任务。

3、application_init 函数

application_init 函数一般用于应用程序初始化，用户通常在该函数中执行程序初始化代码，并打开第一个 GUI 窗口。

4、tk_run 函数

tk_run 函数用于启动 AWTK 的主循环，它会让 GUI 线程不断循环调用以下函数：

(1) main_loop_dispatch_input 函数：获取输入设备的信息，其中会调用 platform_disaptch_input 函数将输入设备信息分发至主循环，用户通常会在该函数中实现输入设备的移植。

(2) event_source_manager_dispatch 函数：处理输入设备信息。

(3) main_loop_dispatch_events 函数：处理定时器和 idle 等消息源。

(4) window_manager_check_and_layout 函数：完成 UI 布局的相关操作。

(6) window_manager_paint 函数：深度遍历树状结构的 UI 树，并且把相关的 UI 控件绘制到屏幕上面，早画的控件会被晚画的控件覆盖。

注: 在 UI 树中，窗口管理器（window_manager）是树顶，窗口管理器下可以有很多窗口，每个窗口下可以有很多控件。

5、tk_quit 函数

当程序调用 tk_quit 函数后，就会退出 AWTK 的主循环，并释放所有 AWTK 相关的资源。

6、application_exit 函数

application_exit 函数一般用于退出应用程序，该函数通常是给用户释放应用程序相关的资源。

1.4 AWTK 代码目录结构

了解 AWTK 在嵌入式下的核心代码以及相关功能模块有利于项目的移植和调试,AWTK 代码目录结构详见下表：

文件/目录	说明	备注
src/awtk_global.c	主要负责 AWTK 的初始化以及退出 AWTK 的相关操作	
src/base	AWTK 底层最基础的逻辑代码	
src/blend	AWTK 颜色融合的算法代码	
src/clip_board	剪切板模块	
src/dialog_highlighters	对话框高亮模块	
src/ext_widgets	扩展控件模块	
src/font_loader	字体解码器的代码	显示字体时使用
src/graphic_buffer	显示 buffer 对象的代码	
src/image_loader	图片解码器的代码	解码图片时使用
src/input_engines	输入法引擎模块	

续上表

文件/目录	说明	备注
src/input_methods	输入法模块	
src/layouters	布局模块	
src/lcd	AWTK 内置的 LCD 实现代码	
src/main_loop	平台相关的消息循环机制代码	
src/native_window	平台相关的物理窗口的实现代码	
src/platforms	通用平台下多线程和文件系统等与平台相关的代码	
src/tkc	AWTK 提供的通用函数库	
src/ui_loader	解析 AWTK 的 UI 文件的代码	
src/vgcanvas	矢量画布模块	
src/widget_animators	控件动画模块	
src/widgets	基础控件的代码	
src/window_animators	窗口动画模块	
src/window_manager	窗口管理器的代码	
3rd/agge	AGGE 矢量画布引擎的代码，AWTK 默认使用该引擎	支持 AGGE 矢量画布时使用
3rd/glad	Windows 的 OpenGL 转接层的代码	支持 OpenGL 使用
3rd/gpinyin	谷歌拼音输入法的代码	
3rd/libunibreak	UNICODE 换行算法代码	
3rd/nanovg	nanovg 库，AWTK 默认使用该引擎，通常与 AGGE 配合使用	支持矢量画布时使用
3rd/stb	stb 库，AWTK 默认使用该引擎，实现图片/字体的解码功能	解码图片时使用

注：AWTK 的代码主要放在 awtk/src 目录以及 awtk/3rd 目录，其中 3rd 目录主要用于存放第三方的类库。

上面提到的所有文件夹和文件都可以直接添加到移植项目中，只需要配置好相关的宏就可以使用了，功能模块相关的宏定义详见本文附录二，但是有些模块需要注意：

1. 需要确定程序是基于 OpenGL 还是基于 Framebuffer 开发，如果基于 Framebuffer 开发，需要添加 lcd_mem_xxx 的所有文件和 native_window_raw.c 文件，如果基于 OpenGL 开发，就需要添加 lcd_nanovg.c 和 native_window_fb_gl.c 文件。
2. 如果是裸系统平台，一般会使用 main_loop_raw.inc 文件实现 AWTK 主循环，而在嵌入式 Linux 平台一般会使用 main_loop_simple.c 文件。
3. platforms 文件夹下提供了一些常见平台的平台相关的移植模块，可以根据具体平台选择性添加。
4. 3rd/nanovg 文件夹中分别提供了 agge、gl 和 base 的相关代码，如果支持矢量画布功能，则 base 中的代码必须全部添加到项目中，并且基于 OpenGL 开发的程序需要添加 gl 相关代码，而基于 Framebuffer 开发的程序需要添加 agge 相关代码。
5. 在移植的过程中，添加头文件路径，awtk 的话只需要增加 awtk/src 和 awtk/src/ext_widgets 的路径即可，如果使用到 3rd 的功能模块的话，还需要添加相关的 3rd 文件夹的路径（例如：使用到 AGGE 矢量画布的话，就需要添加 awtk/3rd/agge，awtk/3rd/nanovg 和 awtk/3rd/nanovg/base 目录）。

此处主要列出了 AWTK 代码结构并提到了一些添加文件时的注意事项，如果读者想要了解更详细的构建移植工程的过程，可以参考 STM32H743 移植笔记[3] 以及 NXP LPC1768

[3] https://github.com/zlgopen/awtk-stm32h743iitx-tencentos/blob/master/docs/stm32h743iitx_port.md

移植笔记[4]，里面详细记录了 Keil 工程的构建过程。

1.5 官方移植案例

AWTK 提供了部分常见平台和操作系统的移植案例，如 PC Windows、PC Linux、嵌入式 Linux、嵌入式无操作系统平台等，用户可以直接使用或参考这些工程做实际开发。详情请浏览《AWTK 生态共建计划》[5]文档。

[4] https://github.com/zlgopen/awtk-lpc1768-raw/blob/master/docs/lpc1768_port.md

[5] https://github.com/zlgopen/awtk/blob/master/docs/awtk_ecology.md

第二部分

移植篇

2. 移植篇导读

AWTK 的可移植性很高，移植层通常需要实现平台初始化、时钟和睡眠以及显示设备（LCD），它们的作用如下：

1. 平台初始化通常用于初始化平台相关的准备模块，比如 AWTK 的内存管理器和时间管理器等。
2. 时钟一般用来计算时间间隔，实现定时器和动画功能；睡眠一般用于 AWTK 主循环限制 GUI 帧率。
3. 显示设备（LCD）用于显示 AWTK 绘制的 GUI 界面。

这些模块的移植过程请参考本文 3、4 章节的内容。除此之外，一般还需要进行输入设备的移植，其步骤请参考本文第 5 章。其他平台相关接口均可根据实际需求决定是否进行移植，比如文件系统、多线程相关资源等，它们的移植过程请参考本文第 6 章。

3. 适配准备工作

3.1 平台堆栈配置

AWTK 最低的栈 (Stack) 要求为 9KB，如果需要支持 jpg/png 图片解码，则栈至少需要 32KB。

不同平台设置栈大小的方法不一样，比如在 AWTK 针对 STM32F767igtx 平台的 移植层[6] 中，需要修改 startup_stm32f767xx.s 文件中的 Stack_Size 属性值。在实时操作系统的平台上，AWTK GUI 线程使用的栈就是开辟线程时定义的栈，所以需要修改线程栈大小。

堆 (Heap) 用于程序运行过程中使用 malloc 开辟内存空间，对 AWTK 来讲，堆就是 AWTK 的动态运行内存，通常用于创建控件、定时器、idle 等各种对象以及保存图片、字模等缓存。其初始化方法详见下一个小节，堆推荐配置范围是 64KB~8MB，具体需根据模块裁剪和应用程序的实际情况调整，计算方法请参考下文裁剪篇的内容。

3.2 平台初始化

AWTK 程序启动时，会在 tk_init 函数中调用 platform_prepare 函数初始化平台相关的准备模块，AWTK 运行流程详见本文 1.3 章节。

用户可以选择使用系统内存堆或让 AWTK 自行管理内存。如果用户希望 AWTK 使用系统标准 malloc 管理内存，则应该在工程中定义 **HAS_STD_MALLOC** 宏；如果用户希望 AWTK 自行管理堆内存，则可以在 platform_prepare 函数中调用 tk_mem_init 函数给 GUI 分配一块内存。代码如下：

```
/* 给 AWTK 开辟一块大小为 TK_MEM_SIZE 的内存 s_mem */
#define TK_MEM_SIZE 8 * 1024 * 1024
ret_t platform_prepare(void) {
/* 定义宏 HAS_STD_MALLOC 将使用系统标准 malloc 函数，具体详见本文附录二 */
#ifndef HAS_STD_MALLOC
  static uint8_t s_mem[TK_MEM_SIZE];  /* 此处以静态数组为例 */
  tk_mem_init(s_mem, sizeof(s_mem));  /* 初始化 AWTK 内存管理器 */
#endif
  return RET_OK;
}
```

其他平台相关的准备模块，比如时间管理模块，同样也可以在 platform_prepare 函数中初始化，具体请参考：awtk/src/tkc/date_time.h。

[6] https://github.com/zlgopen/awtk-stm32f767igtx-raw

3.3 时钟和睡眠移植

在 AWTK 中，时钟一般用来计算时间间隔，实现定时器和动画等功能；睡眠一般用于 AWTK 主循环限制 GUI 帧率。移植时钟和睡眠通常只需实现 awtk/src/tkc/platform.h 文件中的 get_time_ms64 函数和 sleep_ms 函数即可，代码如下：

```
uint64_t get_time_ms64() {
  /* 返回平台当前时间（单位：毫秒） */
  return system_time_ms;
}

void sleep_ms(uint32_t ms) {
  /* 调用平台相关接口实现睡眠 */
  system_sleep_ms(ms);
}
```

需要注意的是，移植层需要确保 get_time_ms64 函数返回的时间是 **64 位毫秒计数器**。如果平台没有提供 64 位计数器，则需要用户自行通过系统中断或硬件定时器实现。如果强行在 get_time_ms64 函数返回 32 位计数器，GUI 长时间运行后可能存在计数溢出风险和不可预测的问题。

此外，在裸系统平台上也可以通过 SysTick 来实现以上函数，只需初始化 SyeTick 并且在编译时添加 awtk/src/platforms/raw/sys_tick_handler.c 文件即可，具体请参考 AWTK 针对 STM32f103ze 裸系统的 移植层[7]。

注: AWTK 提供了常见 RTOS 的时钟与睡眠实现，具体详见 awtk/src/platforms 目录下对应平台的实现代码，用户可直接加入工程使用。

3.4 AWTK 的宏配置

在 PC 或嵌入式 Linux 系统中，AWTK 的宏配置直接在 SCons 脚本（awtk_config.py）中定义。在其他嵌入式平台上，为方便用户配置，AWTK 相关的宏配置通常写在 awtk_config.h 文件中，它主要用于设置平台相关的配置以及开关某些功能模块，示例详见：awtk/src/base/awtk_config_sample.h，代码如下：

```
/* awtk_config.h */
#ifndef AWTK_CONFIG_H
#define AWTK_CONFIG_H

/* 嵌入式系统有自己的main函数时，请定义本宏 */
#define USE_GUI_MAIN 1

/* 如果支持png/jpeg图片，请定义本宏 */
#define WITH_STB_IMAGE 1

...

/* 如果启用矢量画布 VGCANVAS，请定义本宏 */
```

[7] https://github/zlgopen/awtk-stm32f103ze-raw

```
#define WITH_VGCANVAS 1

...

#endif /*AWTK_CONFIG_H*/
```

需要注意的是，使用 awtk_config.h 文件需要在配置工程时定义宏 **HAS_AWTK_CONFIG**，例如，在 Keil 项目中设置编译宏如下图所示：

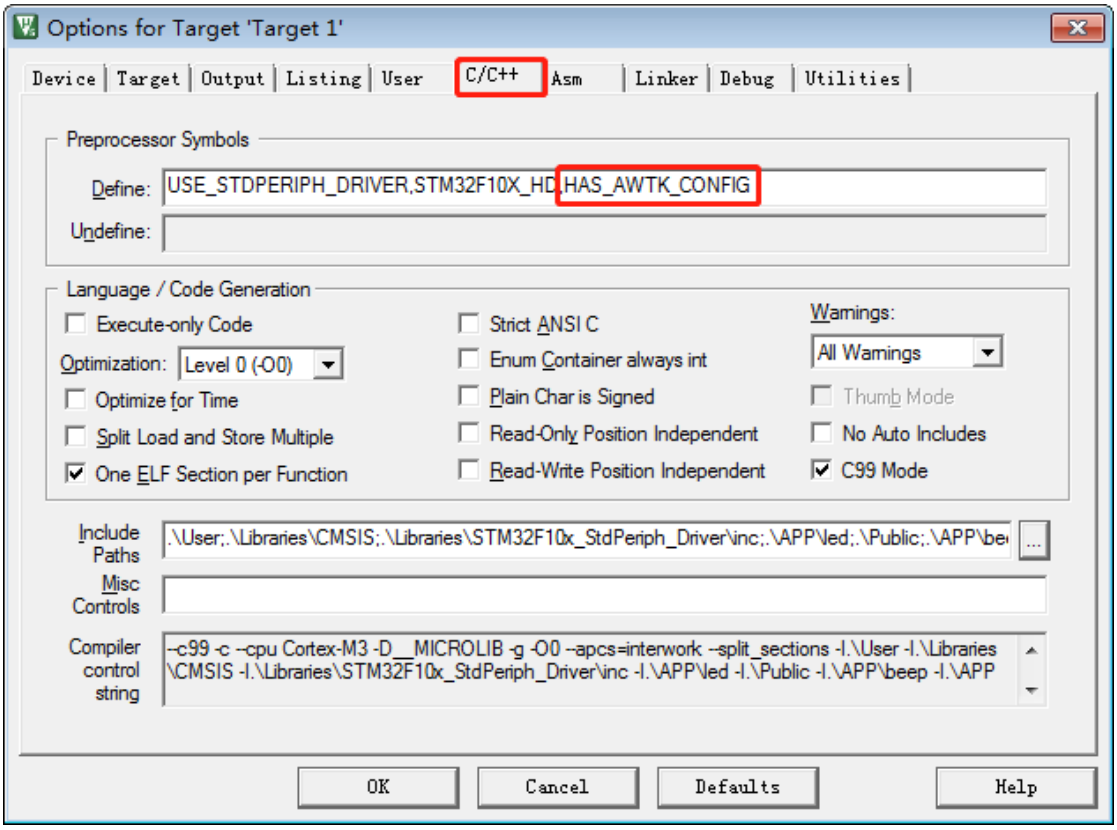

图 3.1 定义宏 HAS_AWTK_CONFIG

注: 各个功能模块的介绍及其宏定义详见文本附录二。

3.5 将 AWTK 代码添加到工程

AWTK 的核心代码结构详见本文 1.4 章节中的表格，根据项目裁剪需求，添加对应的代码文件到工程中，详见本文 10.3 章节。

3.6 自带例程及运行异常问题排查

移植完成后，请尝试在板子上运行 AWTK 自带例程 demoui，验证移植效果。用户需要将 awtk/demos 文件夹下的 assets.c（或 assets-1m.c）、demo_ui_app.c 加入到工程中编译，如运行出现莫名其妙的异常或崩溃，请按如下步骤排查：

1. 增加系统 Stack 大小，如果使用 RTOS 线程，则调整 GUI 线程 Stack 大小。建议尝试设置 Stack 大小 >=32KB；
2. 增加系统 Heap 大小，如果使用 AWTK 自己管理的 Heap，则调整对应的数组大小。建

议尝试设置 Heap 大小 >=4MB；

3. 降低工程编译优化等级。

4. 显示设备适配

系统中存放 LCD 显示数据的显存被称为 Framebuffer（简称 fb），AWTK 设计了一个名为 lcd_t 的类型，把 fb 和 lcd_t 对象关联起来，AWTK 操作 lcd_t 对象等同于直接操作 LCD 显示的数据，达到控制 LCD 显示画面的目的。

注: 大部分嵌入式平台都可以直接操作 fb，特殊小资源的平台没有 fb，则需要通过 SPI 或 I2C 等手段去控制 LCD 显示。

显示设备适配的主要步骤如下图所示：

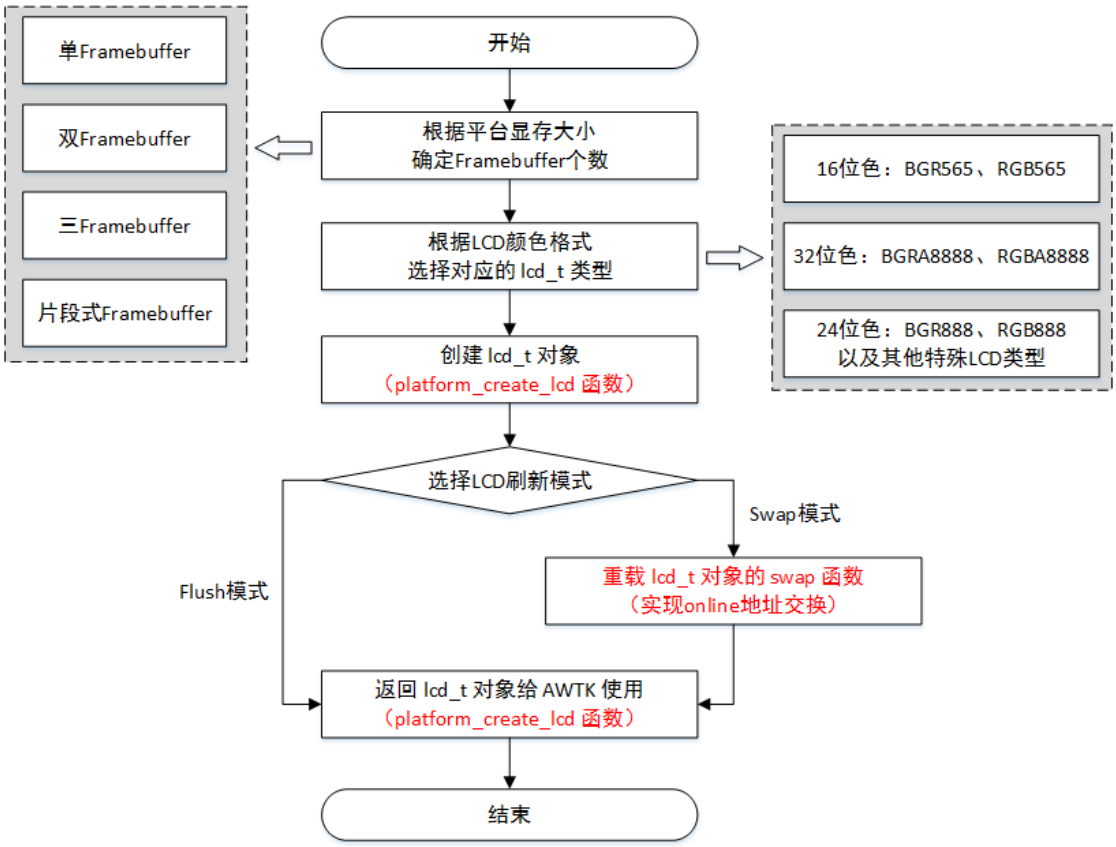

图 4.1　显示设备适配流程

1. 根据平台显存的大小，选择开辟多少块 Framebuffer。
2. 根据 LCD 的颜色格式和 Framebuffer 的个数，确定相关的 lcd_t 类型。
3. **实现 platform_create_lcd 函数**，在 platform_create_lcd 函数中**创建该类型的 lcd_t 对象**。
4. 根据 Framebuffer 的个数，选择刷新 LCD 的方式（分别为 Flush 模式和 Swap 模式），并**重载 lcd_t 对象中平台相关的函数，实现显示模式的逻辑代码**。
5. 在 platform_create_lcd 函数中返回初始化完毕的 lcd_t 对象给 AWTK 使用。

创建 lcd_t 对象后，将默认使用 Flush 刷新模式，用户可以通过重载 lcd_t 的以下函数实现不同的刷新模式：

函数名称	说明	备注
flush	每帧画面绘制结束时被调用，将图像从 offline_fb 拷贝到 online_fb	通常在使用特殊的 LCD 类型时重载
swap	每帧画面绘制结束时被调用，交换 offline_fb 与 online_fb 的地址	通常在使用 Swap 模式刷新 LCD 时重载

在 GUI 主循环中，AWTK 总是把界面先绘制到 offline_fb，再根据移植层实现的不同刷新策略，把 offline_fb 的内容送到 LCD 显示。刷新 LCD 模式如图所示：

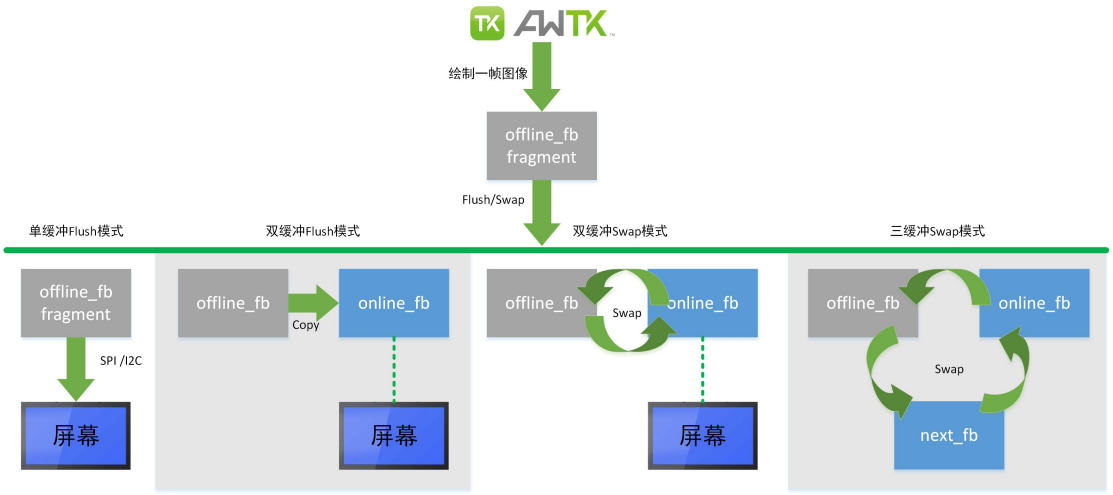

图 4.2　LCD 刷新模式

1. Flush 模式：是指把离线显存（offline_fb）中的数据拷贝在线显存（online_fb）中的操作，LCD 会把 online_fb 中的数据显示出来。Flush 过程中 LCD 显存指针一直保持不变。
2. Swap 模式：是指离线显存（offline_fb）和在线显存（online_fb）的地址交换，LCD 再把新的 online_fb 数据显示出来。Swap 过程中 LCD 显存指针会切换到新的 online_fb 显存上。

这里不同的刷新策略由用户在移植层重载 lcd_t 的 flush/swap 接口去实现，比如缓冲区数量、如何拷贝或交换、如何实现垂直同步等。更多两种显示模式的介绍详见本章中的以下小节。

4.1　评估 Framebuffer 数量

根据不同平台的显存大小，用户可以选择以下四种 Framebuffer 的方案：

1. 单 Framebuffer：平台的显存比较小，只能开辟一块屏幕大小的显存，LCD 显示和 AWTK 绘图都使用块显存。
2. 双 Framebuffer：平台的显存较大，可以开辟两块屏幕大小的内存，一块用于 LCD 显示（online_fb），另一块用于 AWTK 绘图使用（offline_fb）。
3. 三 Framebuffer：平台的显存很大，可以开辟三块屏幕大小的内存，一块用于 LCD 显示（online_fb），一块用于 AWTK 绘图使用（offline_fb），另外一块用于等到交换显存（next_fb）。
4. 片段式 Framebuffer：平台的显存非常小，无法开辟一块完整屏幕大小的显存，只能开

辟一小块内存作为显存的情况。

注: 一块屏幕大小的显存所占内存为：LCD 的宽度 * LCD 的高度 * LCD 的每像素占用的字节数。

4.2 创建 lcd_t 对象

AWTK 的 lcd_t 类型有一个子类 lcd_mem_t，该子类用于和 Framebuffer 绑定，并且该子类提供了嵌入式下常用的 LCD 颜色格式的实现，同时每种颜色格式的 lcd_mem_t 类型都有单/双/三 Framebuffer 的构造函数，让用户适配 LCD 变得非常简单。

这些适配代码实现详见 awtk/ src/ lcd 目录下前缀为“lcd_mem_”的文件，其中 lcd_mem_rgb565.c 提供了 RGB565 格式的 lcd_mem_t 类型的实现，其他文件名以此类推，用户只需要根据系统的 LCD 颜色格式找到对应的 lcd_mem_t 类型使用即可。

注: AWTK 的所有颜色格式布局为小端内存排序，例如 AWTK 的 lcd_t 类型颜色格式为 RGBA，fb 中的内存布局由低位往高位的顺序是 R，G，B，A

4.2.1 16 位色 lcd_t 类型

AWTK 提供了 BGR565 和 RGB565 两种 16 位色的 lcd_mem_t 类型，创建方法如下代码：

```
/* 该函数创建一个颜色格式为 rgb565 的 lcd_t 对象并将其返回给 AWTK 使用 */
lcd_t* platform_create_lcd(wh_t w, wh_t h) {
  /* w 和 h 为 LCD 的分辨率 */
  /* online_fb 为 LCD 正在显示的 fb；offline_fb 为 AWTK 用于绘制的 fb */
  /* 此处把两个 fb 地址传给 lcd_t 对象实现关联 */
  return lcd_mem_rgb565_create_double_fb(w, h, online_fb, offline_fb);
}
```

以下为 AWTK 提供的 16 位色 lcd_mem_t 类型的所有创建函数，请用户根据需求使用：

```
/* 颜色格式：bgr565 */
lcd_t* lcd_mem_bgr565_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_bgr565_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb);
lcd_t* lcd_mem_bgr565_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb, uint8_t* next_fb);

/* 颜色格式：rgb565 */
lcd_t* lcd_mem_rgb565_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_rgb565_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb);
lcd_t* lcd_mem_rgb565_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb,  uint8_t* next_fb);
```

4.2.2 32 位色 lcd_t 类型

AWTK 提供了 BGRA8888 和 RGBA8888 两种 32 位色的 lcd_mem_t 类型，创建方法和上面的 16 位色的 lcd_mem_t 类似。以下为所有创建函数，请用户根据需要选择使用：

```
/* 颜色格式：bgra8888 */
lcd_t* lcd_mem_bgra8888_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_bgra8888_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_
↪t* offline_fb);
lcd_t* lcd_mem_bgra8888_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_
↪t* offline_fb, uint8_t* next_fb);

/* 颜色格式：rgba8888 */
lcd_t* lcd_mem_rgba8888_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_rgba8888_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_
↪t* offline_fb);
lcd_t* lcd_mem_rgba8888_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_
↪t* offline_fb, uint8_t* next_fb);
```

4.2.3 24 位色 lcd_t 类型

AWTK 提供了 BGR888 和 RGB888 两个 24 位色的 lcd_mem_t 类型，创建方法和上面的 16 位色的 lcd_mem_t 类似。由于 24 位色内存不对齐，拷贝速度较慢，影响绘制效率，因此如果硬件允许，建议使用 32 位色的 LCD。以下为所有创建函数，请用户根据需要选择使用：

```
/* 颜色格式为：bgr888 */
lcd_t* lcd_mem_bgr888_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_bgr888_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb);
lcd_t* lcd_mem_bgr888_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb, uint8_t* next_fb);

/* 颜色格式为：rgb888 */
lcd_t* lcd_mem_rgb888_create_single_fb(wh_t w, wh_t h, uint8_t* fbuff);
lcd_t* lcd_mem_rgb888_create_double_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb);
lcd_t* lcd_mem_rgb888_create_three_fb(wh_t w, wh_t h, uint8_t* online_fb, uint8_t*␣
↪offline_fb, uint8_t* next_fb);
```

4.2.4 特殊 lcd_t 类型

AWTK 目前仅提供了常见的 16/24/32 位颜色的 LCD 创建函数，而在某些硬件平台上，还有特殊的 LCD 类型，比如单色屏、5551 的屏等。AWTK 提供了一个名为 lcd_mem_special 的 lcd_mem_t 类型，用于解决该情况。创建方法和上面的 16 位色的 lcd_mem_t 类似，还需要用户提供额外的几个回调函数，由用户实现 Flush 操作，将离线的 offline_fb 内容拷贝到特殊格式的 LCD 在线显存上。

注：离线 offline_fb 仅支持 565、888 和 8888 这几种标准格式。

```
/* awtk/lcd/lcd_mem_special.h 文件 */
/**
 * @method lcd_mem_special_create
 *
 * 创建lcd对象。
 *
 * @param {wh_t} w 宽度。
 * @param {wh_t} h 高度。
 * @param {bitmap_format_t} format 离线lcd的格式。一般用 BITMAP_FMT_BGR565 或
↪BITMAP_FMT_RGBA8888。
 * @param {lcd_flush_t} flush 回调函数，用于刷新GUI数据到实际的LCD。
 * @param {lcd_resize_t} on_resize 用于调整LCD的大小。一般用NULL即可。
 * @param {lcd_destroy_t} on_destroy lcd销毁时的回调函数。
 * @param {void*} ctx 回调函数的上下文。
 *
 * @return {lcd_t*} 返回lcd对象。
 */
lcd_t* lcd_mem_special_create(wh_t w, wh_t h, bitmap_format_t fmt, lcd_flush_t on_
↪flush, lcd_resize_t on_resize, lcd_destroy_t on_destroy, void* ctx);
```

4.2.5 AWTK 内部位图颜色格式

确定了 lcd_t 类型的颜色格式后，还需要明确 AWTK 内部解码图片时，缓存的位图颜色格式。如果 AWTK 解码图片的颜色格式和 LCD 的颜色格式一致，那么像素点在显示时就不需要做颜色转换，可以提高渲染图片的效率。

AWTK 默认将所有图片解码为 RBGA8888 格式的位图，并提供了以下宏定义来设置解码位图的颜色格式：

宏定义	不透明图片解码为	透明图片解码为
缺省宏定义	RBGA8888	RBGA8888
WITH_BITMAP_BGR565	BGR565	无影响
WITH_BITMAP_RGB565	RGB565	无影响
WITH_BITMAP_BGR888	BGR888	无影响
WITH_BITMAP_RGB888	RGB888	无影响
WITH_BITMAP_BGRA	无影响	BGRA8888

需要注意的是，AWTK 在绘图的时候会自动检测是否需要颜色转化，就算**不定义这些宏或者随便定义宏也可以正常显示，只是效率变低而已。**

4.3 使用 Flush 模式的双 Framebuffer 移植案例

此处假设平台的 LCD 颜色格式为 BGR565，平台显存可以分配出双 Framebuffer 。

1、LCD 初始化

AWTK 程序启动时，会在 tk_init 函数中调用 platform_create_lcd 函数初始化 LCD，该函数与平台相关，一般在移植层上面实现，AWTK 运行流程详见本文 1.3 章节的内容。GUI 主循环 main_loop 的基本功能通过 main_loop_raw.inc 来实现，用户只需实现 platform_create_lcd 函数初始化 LCD 即可，代码如下：

```
extern uint8_t* online_fb;   /* 系统 LCD 使用的显存 */
extern uint8_t* offline_fb;  /* 给 AWTK 绘图的显存 */
lcd_t* platform_create_lcd(wh_t w, wh_t h) {
  /* 根据 LCD 类型调用对应 lcd_mem_t 构造函数创建 lcd_t 对象 */
  return lcd_mem_bgr565_create_double_fb(w, h, online_fb, offline_fb);
}
#include "main_loop/main_loop_raw.inc"
```

2、LCD 的刷新流程

本案例中 lcd_mem_bgr565_create_double_fb 函数创建的 lcd_t 对象默认使用 Flush 模式刷新图像，无需用户实现额外代码，即 GUI 通过拷贝内存的方式刷新 LCD，绘图流程如下：

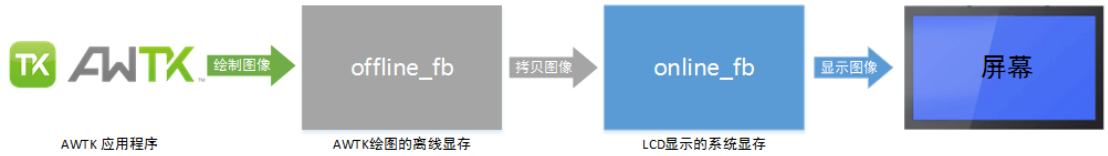

图 4.3 Flush 模式的双 Framebuffer 刷新流程

(1) AWTK 调用 lcd_begin_frame 函数获取脏矩形，并获取 offline_fb 地址，准备绘制。

(2) AWTK 调用 lcd_t 对象的相关接口将界面变化的部分绘制到 offline_fb 上。

(3) AWTK 调用 lcd_end_frame 函数完成绘制，该函数中会调用 lcd_mem_flush 函数将 offline_fb 中脏矩形部分的图像数据拷贝到 online_fb 上完成 LCD 的刷新。

注: 上述的流程或者函数被封装在 lcd_mem.inc 中，所以用户不需要重载任何函数。

本方案使用简单，无需写额外的刷新逻辑代码，但无法实现垂直同步功能，因此在显示时可能会出画面现撕裂现象。

4.4 使用 Swap 模式的双 Framebuffer 移植案例

此处假设平台的 LCD 颜色格式为 BGR565，平台显存可以分配出双 Framebuffer 。

显存地址（online_fb）交换操作一般要调用具体平台的 API，所以用户除了需要创建 lcd_mem_t 对象，还需要重载实现对象的 swap 函数，代码如下：

```
extern uint8_t* online_fb;    /* 系统 LCD 使用的显存 */
extern uint8_t* offline_fb;   /* 给 AWTK 绘图的显存 */

static ret_t lcd_mem_swap(lcd_t* lcd) {
  lcd_mem_t* mem = (lcd_mem_t*)lcd;
```

```
  uint8_t* tmp_fb = mem->offline_fb;

  system_wait_vbi();  /* 调用系统API加入垂直同步等待来保证画面没有撕裂 */

  /*
   * 调用系统API把 offline_fb 设置为系统 LCD 使用的显存地址，
   * 随后交换 offline_fb 和 online_fb 地址。
   */
  system_reg.fbaddr = mem->offline_fb;
  lcd_mem_set_offline_fb(mem, mem->online_fb);
  lcd_mem_set_online_fb(mem, tmp_fb);
  return RET_OK;
}

lcd_t* platform_create_lcd(wh_t w, wh_t h) {
  /* 根据 LCD 类型调用对应 lcd_mem_t 构造函数创建 lcd_t 对象 */
  lcd_t* lcd = lcd_mem_bgr565_create_double_fb(w, h, online_fb, offline_fb);
  lcd->swap = lcd_mem_swap;          /* 重载 swap 函数 */
  lcd->support_dirty_rect = FALSE;  /* 关闭脏矩形机制(新版本AWTK可以不关闭) */
  return lcd;
}
#include "main_loop/main_loop_raw.inc"
```

Swap 模式和 Flush 模式唯一的不同就是 Swap 模式重载了 lcd_t 对象的 swap 函数，让原本 LCD 的刷新流程的最后一步 lcd_mem_flush 函数改为调用 lcd_mem_swap 函数完成显存地址交换，并且重新设置系统 LCD 使用的显存地址，其刷新流程如下图所示：

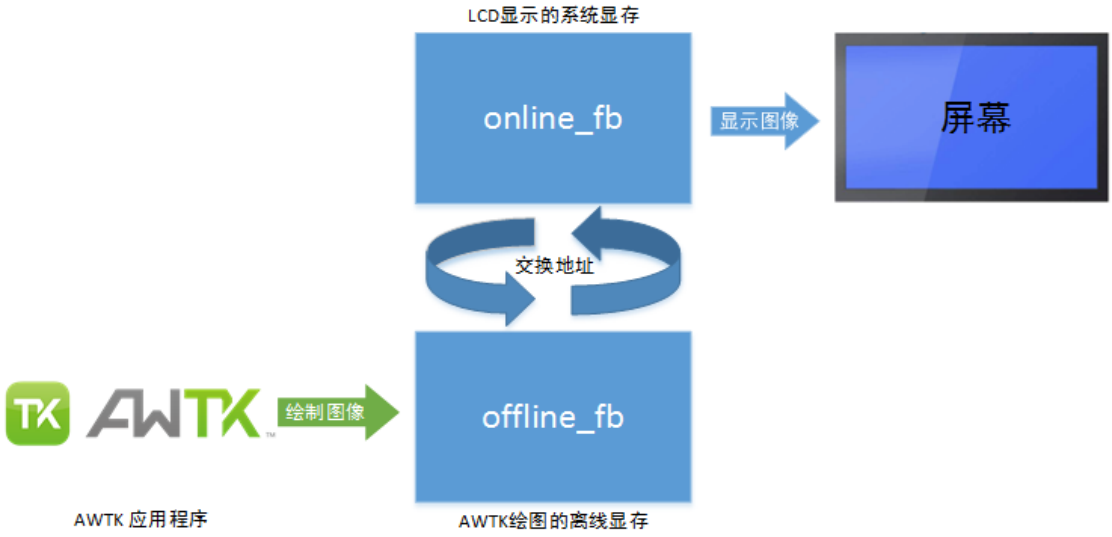

图 4.4　Swap 模式的双 Framebuffer 刷新流程

注: 在使用 Swap 模式的时候，一般会加入垂直同步等待来保证画面没有撕裂。

4.5 使用 Swap 模式的三 Framebuffer 移植案例

此处假设平台的 LCD 颜色格式为 BGR565，平台显存可以分配出三 Framebuffer，并且加入垂直同步机制。

1、LCD 的刷新流程如下：

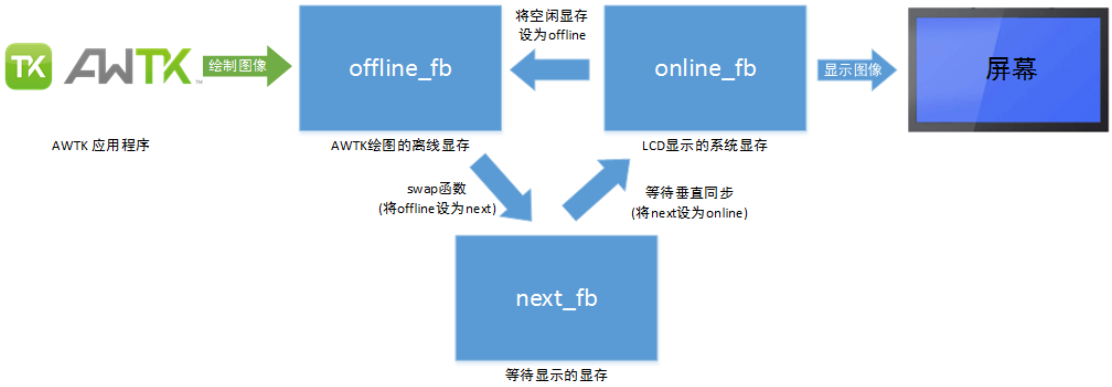

图 4.5 Sawp 模式的三 Framebuffer 刷新流程

(1) AWTK 调用 lcd_begin_frame 函数获取脏矩形区域和准备绘制。

(2) AWTK 将界面变化部分绘制到 offline_fb 上面。

(3) AWTK 调用 lcd_end_frame 函数，完成绘制，其中会调用重载的 swap 函数。

(4) 移植层在 swap 函数中等待完成垂直同步的消息后，把 offline_fb 地址交给 next_fb 保存，并且把全局变量中的 online_fb 地址作为 offline_fb 地址。

(5) 移植层另外一个时刻，垂直同步回调函数触发后，会先把 online_fb 地址保存在全局变量中，把 online_fb 地址设置为 next_fb 的地址，并且对外通知完成垂直同步的消息。

2、伪代码实现

```
static uint8_t* volatile s_next_fb = NULL; /* next_fb地址 */
static uint8_t* volatile s_next_offline_fb = NULL; /* 上一帧online_
↪fb，也是下一帧offline_fb */
extern uint8_t* s_framebuffers[3];         /* 三个Framebuffer的地址 */

/* 重载 swap 函数 */
ret_t lcd_mem_swap(lcd_t* lcd) {
  lcd_mem_t* mem = (lcd_mem_t*)lcd;
  /* 等到s_next_fb空闲同步已完成 */
  while (s_next_fb != NULL) {};
  /* 将绘制好的 offline_fb 设置为 next_fb */
  s_next_fb = lcd_mem_get_offline_fb(mem);
  /* 把上一帧的 online_fb 作为下一帧的 offline_fb */
  lcd_mem_set_offline_fb(mem, s_next_offline_fb);
  return RET_OK;
}

/* LCD 扫描完成后会触发垂直同步回调函数 */
void LTDC_IRQHandler(void) {
```

```
    if (s_next_fb != NULL) {
      /* s_next_offline_fb 变量保存上一帧的 online_fb 地址 */
      s_next_offline_fb = system_reg.fbaddr;
      /* 调用系统API把 s_online_fb 的地址设置为当前 LCD 的显示显存 */
      system_reg.fbaddr = s_next_fb;
      s_next_fb = NULL;
    }
}

/* 创建 lcd_t 对象 */
lcd_t* platform_create_lcd(wh_t w, wh_t h) {
  /* s_next_offline_fb 变量保存下一帧的 offline_fb 地址  */
  s_next_offline_fb = s_framebuffers[2];
  /* 调用系统API把 s_online_fb 的地址设置为当前 LCD 的显示显存 */
  system_reg.fbaddr = s_framebuffers[0];

  /* 根据 LCD 类型调用对应构造函数创建 lcd_t 对象， s_framebuffers[1] 为 offline_fb
↪地址。 */
  lcd_t* lcd = lcd_mem_bgr565_create_three_fb(w, h, s_framebuffers[0], s_
↪framebuffers[1],
                                              s_framebuffers[2]);
  /* 关闭脏矩形机制和重载 swap 函数 */
  lcd->swap = lcd_mem_swap;
  /* 调用系统API配置屏幕中断*/
  system_enable_ltdc_irq();
  return lcd;
}

#include "main_loop/main_loop_raw.inc"
```

三 Framebuffer 实现的 LCD 相比于双 Framebuffer，多了一个等待显示的缓冲 next_fb，这样可以缓解双 Framebuffer 每帧画面要等待垂直同步信号，效率降低的问题，并且通过交换显存地址刷新 LCD，可以有效解决图像撕裂的问题。

4.6 使用 Swap + Flush 移植案例

Swap 和 Flush 各有各的优势（具体详见下一章节），使用 Swap + Flush 来刷新 LCD 可以结合这两种模式的优点，解决垂直同步问题的同时也可以启用脏矩形机制，并且还能支持 LCD 旋转（默认情况下只有 Flush 模式支持 LCD 旋转）。但两种模式的结合会消耗更多的内存，也会多一次显存拷贝操作，请酌情使用。

注: 此处采用 awtk-linux-fb[8] 项目中的双 fb 刷新模式作为案例来讲解如何同时使用两种刷新模式。注意这里说的双 fb 指的是一个离线 fb 和两个在线 fb，可以在目标板上执行 fbset 命令设置虚拟高 =2 倍屏幕高来设置在线 fb 数量。如果显存足够大，甚至可以设置成三 fb。

1、awtk-linux-fb 中 fb 的刷新流程如下：

[8] https://github.com/zlgopen/awtk-linux-fb

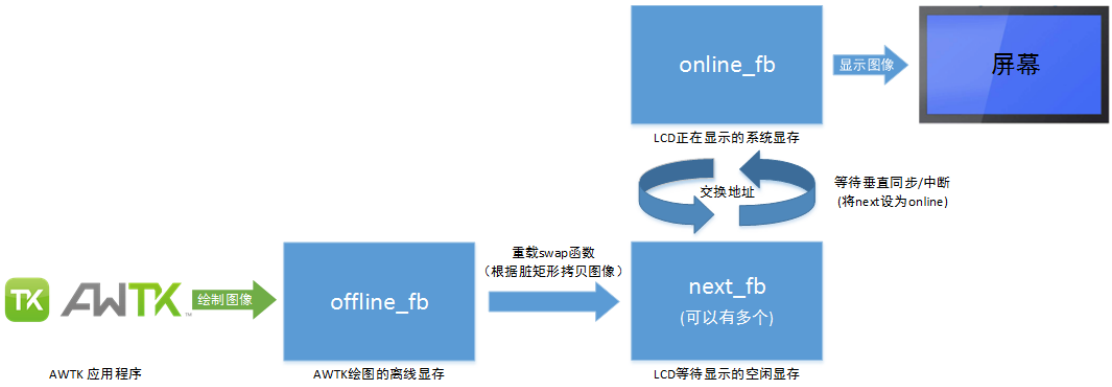

图 4.6　Swap+Flush 的刷新流程

(1) AWTK 调用 lcd_begin_frame 函数获取脏矩形区域和准备绘制。

(2) AWTK 将界面变化部分绘制到 offline_fb 上面，其中 offline_fb 是 malloc 出来的一块与在线 fb 同大小的常规内存。

(3) AWTK 调用 lcd_end_frame 函数完成绘制，中间会调用 swap/flash 函数，这两个函数均可被重载（案例中 swap/flash 被重载为 lcd_mem_linux_wirte_buff 函数）。

(4) lcd_mem_linux_wirte_buff 函数中会获取脏矩形列表，并根据这些脏矩形把 offline_fb 中的数据拷贝到空闲的 next_fb 中，拷贝完成后将这个空闲的 next_fb 放入繁忙队列。

(5) 线程 fbswap_thread 会等待 next_fb 准备好以及屏幕垂直同步中断，中断到来时交换 online_fb 和 next_fb。即把上一个的 online_fb 的地址放入空闲队列变成 next_fb，然后把绘制好的 next_fb 将其设置为当前 online_fb 显示到 LCD 上。

2、关键代码分析：

```
/* awtk-linux-fb/awtk-port/lcd_linux/lcd_linux_fb.c */
/* 以下将省略大量无关代码以及线程安全锁代码，只为演示整个 fb 操作流程 */

/* 把offline fb的内容拷贝到指定的在线fb(fbid)，其他RTOS平台移植时可参考本函数 */
static ret_t lcd_linux_flush(lcd_t* base, int fbid) {
  buff = fb->fbmem0 + size * fbid; /* 获得要进行拷贝的在线 fb 显存地址 */

  /* 分别创建 offline_fb 和 online_fb 对象，用于后续的脏矩形拷贝操作，固定的套路 */
  lcd_linux_init_drawing_fb(lcd, &offline_fb);
  lcd_linux_init_online_fb(lcd, &online_fb, buff, fb_width(fb), fb_height(fb), fb_
↪line_length(fb));

  /* 把在线 fb 显存地址放到脏矩形管理器中管理，␣
↪并更新脏矩形管理器中的所有地址的脏矩形信息 */
  lcd_fb_dirty_rects_add_fb_info(&(lcd->fb_dirty_rects_list), buff);
  lcd_fb_dirty_rects_update_all_fb_dirty_rects(&(lcd->fb_dirty_rects_list), base->
↪dirty_rects);

  /* 从脏矩形管理器获取这个在线 fb 显存的脏矩形， 并根据脏矩形来拷贝对应的数据 */
  dirty_rects = lcd_fb_dirty_rects_get_dirty_rects_by_fb(&(lcd->fb_dirty_rects_
↪list), buff);
  if (dirty_rects != NULL && dirty_rects->nr > 0) {
```

```
    for (int i = 0; i < dirty_rects->nr; i++) {
      const rect_t* dr = (const rect_t*)dirty_rects->rects + i;
      /* 根据应用是否旋转 LCD 进行直接拷贝或旋转拷贝 */
      if (o == LCD_ORIENTATION_0) {
        image_copy(&online_fb, &offline_fb, dr, dr->x, dr->y);
      } else {
        image_rotate(&online_fb, &offline_fb, dr, o);
      }
    }
  }

  /* 重置脏矩形管理器中这个在线 fb 显存的脏矩形 */
  lcd_fb_dirty_rects_reset_dirty_rects_by_fb(&(lcd->fb_dirty_rects_list), buff);
  return RET_OK;
}

/* AWTK 往 offline fb 绘制完一帧画面时，将进入该函数，该函数在 GUI 线程运行 */
static ret_t lcd_mem_linux_wirte_buff(lcd_t* lcd) {
  /* 等待并获得下一个空闲的在线 fb（next_fb） */
  tk_semaphore_wait(s_sem_spare, -1);
  fb_taged_t* spare_fb = get_spare_fb();

  /* 把 offline fb 的内容拷贝到这个空闲的在线 fb（next_fb） */
  ret = lcd_linux_flush(lcd, spare_fb->fbid);

  /* 把 next_fb 的标记修改为数据已准备好，发信号通知 fbswap_thread 交换显存 */
  spare_fb->tags = FB_TAG_READY;
  tk_semaphore_post(s_sem_ready);
}

/* 两个在线 fb 交换线程 */
static void* fbswap_thread(void* ctx) {
  /* 等待 next_fb 数据准备好（填充完成）， 把 next_fb 设置为当前在线 online_fb */
  tk_semaphore_wait(s_sem_ready, -1);

  fb_taged_t* ready_fb = get_ready_fb();
  int ready_fbid = ready_fb->fbid;
  vi.yoffset = ready_fbid * fb_height(fb);

  /* 发出切换显存地址请求 FBIOPAN_DISPLAY，并等交换完成(垂直同步到来时交换完成) */
  ioctl(fb->fd, FBIOPAN_DISPLAY, &vi);
  ioctl(fb->fd, FBIO_WAITFORVSYNC, &dummy);

  /* 把之前的的 online_fb 变为下一个空闲状态（next_fb），并发信号通知 GUI 线程 */
  fb_taged_t* last_busy_fb = get_busy_fb();
  last_busy_fb->tags = FB_TAG_SPARE;

  tk_semaphore_post(s_sem_spare);
  ready_fb->tags = FB_TAG_BUSY;
```

```
}

/* lcd 适配层初始化 */
static lcd_t* lcd_linux_create_swappable(fb_info_t* fb) {
  uint8_t* offline_fb = (uint8_t*)(malloc(fb_size()));
  lcd = lcd_mem_XXXXX_create_single_fb(w, h, offline_fb);

  /* 重载 swap 函数和 flush 函数，LCD 不旋转时将进入 swap 函数，LCD 旋转时将进入
↪flush 函数 */
  lcd->swap = lcd_mem_linux_wirte_buff;
  lcd->flush = lcd_mem_linux_wirte_buff;
}
```

上述内容可以总结为以下两点：

- 在 lcd_mem_linux_wirte_buff 函数中，根据脏矩形的区域把 offline_fb 的数据拷贝（同时旋转）到空闲的 next_fb。
- 垂直同步信号到来时，把 next_fb 和 online_fb 进行交换，实现屏幕显示帧更新，并解决刷新画面撕裂问题。

示例代码详见：awtk-linux-fb/awtk-port/lcd_linux/lcd_linux_fb.c。

4.7 理解的 Flush 和 Swap 的本质

在 AWTK 整个绘图流程中，有两个非常重要的 Framebuffer 指针。一个是在 lcd_t 对象中记录的当前的离线显存指针 offline_fb，AWTK 会在该指针指向的缓冲区中做所有的绘图操作；另外一个是 LCD 控制器记录的当前在线显存指针 online_fb（通常是一个硬件寄存器），控制器会不断的扫描该指针指向的缓冲区，把内容映射到显示屏上。Flush 和 Swap 要做的核心工作就是如何有效的使用和控制这两个指针，使得 AWTK 绘制的画面可以及时的完美的显示到屏幕上。

```
mem->offline_fb      /* offline_fb */
system_reg.fbaddr    /* online_fb  */
```

Flush 通常指的是拷贝操作，把离线显存（offline_fb）的数据拷贝到 LCD 显示显存（online_fb）中，但如果拷贝过程中刚好 LCD 也在刷新 online_fb，相当于一边在写 online_fb，另外一边同时在读，速度不可能完全同步，很容易出画面撕裂感。

Swap 通常指的是交换指针操作，LCD 控制器每扫描完一帧画面后会触发一次中断（垂直同步信号），这时马上的把离线显存（offline_fb）和在线显存（online_fb）的两个指针做交换，由于地址的设置几乎是瞬间就完成，所以 LCD 画面不会出现撕裂感。但由于中断处理过程与 GUI 主循环是异步执行的，两个指针应在合适的时机做交换，即必须保证将要切换到 online_fb 的是一帧绘制完整的缓冲区。实践中可以通过多缓冲和信号量同步的机制解决。

在某些应用场合，甚至可以把 Flush 和 Swap 机制结合起来使用，比如 awtk-linux-fb，通过 Flush 实现带 cache 的可旋转的 offline_fb，通过多个 online_fb 缓冲 Swap 实现垂直同步交换。AWTK 并没有限制用户如何使用这两种模式，而取决于用户如何实现 lcd_t 的 flush 或 swap 接口，如何有效的使用和控制 online_fb、offline_fb 这两个指针，移植时应根据使用场景需求灵活取舍。

4.7.1 Flush 的优点和缺点总结

Flush 是把离线显存（offline_fb）的数据拷贝到其他显存中，无法实现垂直同步，而拷贝势必会消耗性能。同时 Flush 最坏的情况是既要全屏画，又要全屏拷贝，这样会导致双倍的消耗时间，但是 Flush 也不是一无是处的，下面是 Flush 的优点：

1. 拷贝就意味着离线显存（offline_fb）的地址是不会变，AWTK 可以一直使用这块显存，那么这块内存就可以是 cache 类型的，有 CPU 的 cache 机制支持，可以提高内存的读写速度，即可以提高 AWTK 绘图的效率。
2. 由于通过拷贝内存刷新图像，所以可以在拷贝的时候给显存数据做旋转，达到屏幕旋转的效果，其中 AWTK 的 tk_set_lcd_orientation 函数（LCD 旋转机制）就是使用 Flush 模式来实现的。
3. 由于通过拷贝内存刷新图像，所以可以在拷贝的过程中，做颜色转化，比如本文 3.2.3 章节中说的要在特殊颜色格式的 LCD 上（lcd_mem_special 类型）显示，也是使用到 Flush 模式。

4.7.2 Swap 的优点和缺点总结

Swap 是把离线显存（offline_fb）的地址和其他的显存地址进行交换，而交换地址是不需要耗费时间的，所以 Swap 机制最大的好处就是省下拷贝数据的时间，实现垂直同步。但是这也会带来另外一个问题，就是离线显存（offline_fb）的属性一般是 no cache，这样会让 AWTK 在绘图时，尤其是颜色混合计算时，频繁的读取离线显存（offline_fb）的数据，导致性能下降。

同时因为 Swap 的显存地址交换，无法实现画面旋转功能，无法启用 AWTK 的脏矩形机制，即 Swap 模式下每一帧都需要全屏绘图。

注：在 AWTK 1.7 及其以上版本中，三 Framebuffer 的 Swap 模式支持脏矩形刷新机制，但必须使用 lcd_mem_set_offline_fb 去修改 offline_fb 指针。

4.8 片段式 Framebuffer 移植案例

在低端的嵌入式平台上，可能没有足够的空间创建一屏的 Framebuffer，而直接使用寄存器写入颜色数据容易出现屏幕闪烁和画面撕裂的现象，比较好的方法是创建一小块 Framebuffer，把需要绘制的区域分成多个小块，一次绘制一小块，由于 AWTK 有脏矩阵机制，在大多数情况下只需要刷新一小块屏幕，依然可以保持比较快的速度，且解决寄存器写入的闪烁问题。

此处假设平台的 LCD 颜色格式为 BGR565，平台显存无法分配出单个完整的 Framebuffer。

1、LCD 初始化

只需在 platform_create_lcd 函数中调用 lcd_mem_fragment_create 函数创建 lcd_t 对象即可，代码如下：

```
static lcd_t *platform_create_lcd(wh_t w, wh_t h) {
  /* 创建基于片段式 Framebuffer 实现的 lcd_t 对象 */
  return lcd_mem_fragment_create(w, h);
}
```

```
#include "main_loop/main_loop_raw.inc"   /* 添加 main_loop_raw.inc 中的代码 */
```

2、LCD 的工作原理

片段式 Framebuffer 的工作原理其实就是创建一小块内存作为片段式 Framebuffer，其大小为 FRAGMENT_FRAME_BUFFER_SIZE * 单位像素大小，在绘制图像时，先通过 AWTK 的脏矩形机制找到重绘区域，然后根据重绘区域的大小判断是否需要进行切片：

- 若无需切片，则直接将重绘区域的图像拷贝到片段式显存中，再通过 Flush 函数绘制到屏幕的对应区域；
- 若需要切片，则在切片后按照顺序将每一片重绘区域先拷贝到片段式显存中，再通过 Flush 函数绘制到屏幕的对应区域。

绘制流程图如下：

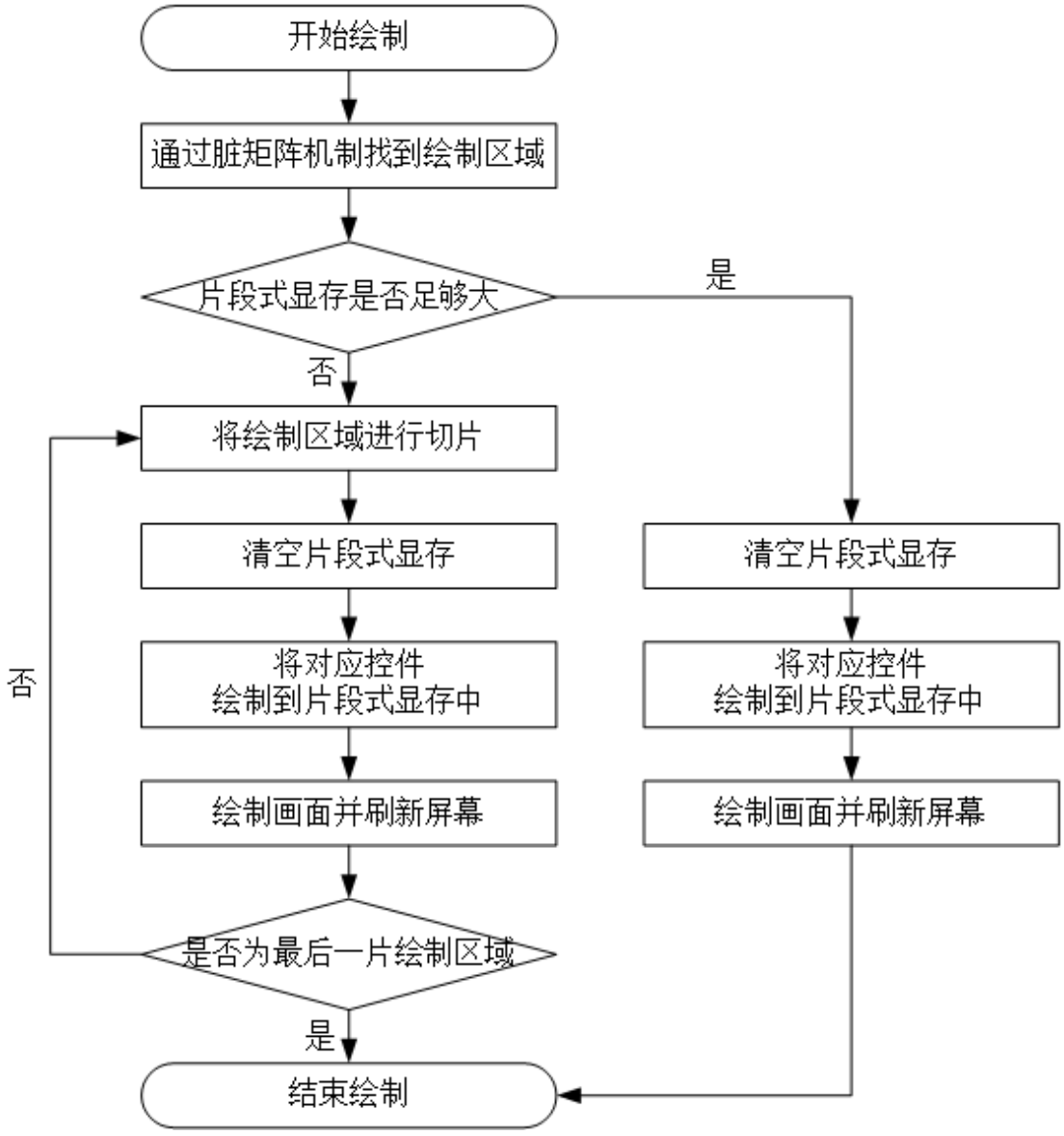

图 4.7　片段式显存绘制流程图

3、LCD 的实现

AWTK 提供了基于片段式 Framebuffer 的 LCD 缺省实现，只需要定义宏 FRAGMENT_FRAME_BUFFER_SIZE 并提供 set_window_func 和 write_data_func 两个函数/宏即可。它们的用法如下：

- set_window_func 函数用于设置 LCD 上要写入颜色数据的区域，该区域对 AWTK 来讲实际上是就脏矩形区域，以矩形左上角为起始坐标，向右延伸宽度，向下延伸高度。设置区域相较于每次写入颜色时设置坐标，可以极大提高工作效率。
- write_data_func 函数用于向 LCD 上写入颜色数据，从设置区域的起始坐标开始，从左往右、从上往下每次写入一个像素点。

首先，通过定义宏 FRAGMENT_FRAME_BUFFER_SIZE 设置片段式显存的大小，代码如下：

```
#define FRAGMENT_FRAME_BUFFER_SIZE 8 * 1024
```

注: 备注：宏 FRAGMENT_FRAME_BUFFER_SIZE 的单位是像素个数，其大小必须大于等于 LCD 的宽度，这是由于片段式 Framebuffer 的实现原理导致的。

接下来定义宏 set_window_func 和宏 write_data_func，代码如下：

```
#define set_window_func LCD_Set_Window       /* 使用 LCD_Set_Window␣
↪函数设置绘制区域 */
#define write_data_func LCD_WriteData_Color  /* 使用 LCD_WriteData_Color␣
↪函数写入颜色数据 */
```

除了以上方式之外，在特殊情况下（比如 SPI 接口的 LCD 或特殊的 LCD 格式类型），也可以选择实现 lcd_draw_bitmap_impl 函数/宏来绘制图像，它负责把变化的部分更新到物理设备，参考片段式 Framebuffer 中的 Flush 函数（lcd_mem_fragment_flush）

```
/* awtk/src/lcd/lcd_mem_fragment.inc */
...
static ret_t lcd_mem_fragment_flush(lcd_t* lcd) {
  lcd_mem_fragment_t* mem = (lcd_mem_fragment_t*)lcd;

  int32_t x = mem->x;
  int32_t y = mem->y;
  uint32_t w = mem->fb.w;
  uint32_t h = mem->fb.h;
  pixel_t* p = mem->buff;

  /**
   * 如果定义了宏 lcd_draw_bitmap_
↪impl，则使用该宏来刷新屏幕（将片段式显存中的图像数据拷贝到屏幕上），
   * 否则使用宏 set_window_func 和宏 write_data_func 来刷新屏幕。
  */
#ifdef lcd_draw_bitmap_impl
  lcd_draw_bitmap_impl(x, y, w, h, p);
#else
  uint32_t nr = w * h;
  set_window_func(x, y, x + w - 1, y + h - 1);
  while (nr-- > 0) {
```

```
    write_data_func(*p++);
  }
#endif

  return RET_OK;
}
...
```

4、片段式 Framebuffer 的优缺点

片段式 Framebuffer 主要用在内存不足以提供完整 Framebuffer 的低端平台上，其优点相当明显。

- 大幅度减少了 Framebuffer 的内存开销，支持脏矩阵机制，提高了低端平台上的渲染性能；
- 能有效解决基于寄存器实现的 LCD 在屏幕较大时出现的闪烁和画面撕裂问题。

由于内存和 CPU 性能问题，片段式 Framebuffer 的缺点如下：

- 不支持窗口动画；
- 不支持离线画布（canvas_offline_t）；
- 不支持控件截图，即无法调用 widget_take_snapshot_rect 接口；
- 脏矩形区域较大时，重绘区域切片较多，绘制速度比较慢。

片段式 Framebuffer 的具体实现，可以参考 AWTK 针对 STM32f103ze 的移植[9]。

4.9 LCD 显示异常时的排查方法

在移植 AWTK 后，如果 LCD 显示异常，比如出现颜色异常、花屏、画面错位等情况，通常可以按照以下顺序排查问题：

1. 先不运行 AWTK，直接向显存中写入数据，检查硬件 LCD 是否能正常显示，颜色是否正常；
2. 确保硬件 LCD 显示正常后，检查移植层中的 AWTK lcd_t 对象的颜色格式是否与硬件 LCD 一致；
3. 确保颜色格式一致后，检查 lcd_t 对象使用的 Framebuffer 地址是否正确；
4. 以上都确认无误后，可以调试移植层 lcd_t 对象的重载函数，比如重载的 flush 函数或 swap 函数，查看绘制数据是否正常。

[9] https://github.com/zlgopen/awtk-stm32f103ze-raw

5. 输入设备适配

5.1 输入设备适配的基本步骤

在 AWTK 中，主循环（main_loop）主要负责维持事件分发和界面绘制这个不断循环的过程，而输入设备移植的本质就是捕获系统的触摸、鼠标、键盘等输入事件，再分发给 GUI 处理，各类输入设备移植的步骤基本相同，可总结为以下三步：

1. 通过设备驱动获取相应的设备事件；
2. 将设备事件转化为 AWTK 支持的事件；
3. 向 GUI 主循环（main_loop）分发转化后的事件。

5.1.1 捕获系统输入事件

不同平台上捕获系统输入设备事件的方法都不一样，具体需要根据实际平台上的驱动实现，此处不做赘述。

5.1.2 AWTK 支持的输入事件

目前 AWTK 支持的输入设备事件详见下表，只要能将获取到的设备事件转化为下表中的事件，那么移植就没有太大问题，各个事件转化时需要设置的参数详见下文的示例教程：

设备事件类型	事件名称	常用设备
wheel_event_t	鼠标滚轮事件	鼠标
pointer_event_t	指针事件	鼠标、触摸屏
key_event_t	按键事件	键盘、特殊按键设备（比如 3/5 按键）
multi_gesture_event_t	多点触摸手势事件	支持多点触摸的触摸屏

注: 以上设备事件包含的具体信息以及它们支持的事件类型详见：awtk/src/base/events.h。

5.1.3 分发输入设备事件

捕获到系统设备事件并将其转化为 AWTK 支持的事件后，可以调用 AWTK 提供的以下接口将事件分发到 GUI 的主循环中处理：

函数名称	说明	备注
main_loop_queue_event	分发事件请求	可分发 AWTK 中的任意事件，需提供事件请求结构体
main_loop_post_key_event	分发按键事件	需提供键值与按键状态
main_loop_post_pointer_event	分发指针事件	需提供指针坐标与指针状态
main_loop_post_multi_gesture_event	分发多点触摸事件	需提供多点触摸事件结构体

注: 如果在非 GUI 线程中调用以上接口分发事件，那么必须参考本文第五章的内容适配互斥锁。

5.1.4 分发输入设备事件的方式

常见的输入设备事件分发方式有两种，分别是在 GUI 线程中分发以及在非 GUI 线程中分发，AWTK 为这两种方式分别提供了简单的实现，以方便用户移植。

1、在 GUI 线程中分发事件

对于裸系统平台，AWTK 提供了 main_loop_raw.inc，该文件实现了裸系统 main_loop 的基本功能，在移植时用户只需实现负责输入设备事件的分发 platform_disaptch_input 函数即可，该函数在 GUI 主循环中被调用，会阻塞主线程，具体的流程详见本文 1.3 章节，示例代码如下：

```
/* 分发输入设备事件 */
ret_t platform_disaptch_input(main_loop_t *l) {
  /* 捕获系统的触摸、鼠标、键盘等输入事件，并转化为 AWTK 支持的事件 */
  event_queue_req_t req;           /* AWTK 的事件请求 */
  memset(&req, 0x00, sizeof(req));
  req.xxx = system_get_input_event();
  ...
  /* 调用本文上一小节中介绍的接口将事件分发到 GUI 主循环 */
  main_loop_queue_event(l, &req);
  return RET_OK;
}

#include "main_loop/main_loop_raw.inc"
```

2、在非 GUI 线程中分发事件

在非 GUI 线程中获取并分发设备事件，不会阻塞主线程，可以在一定程度上提高 GUI 刷新效率。示例代码如下：

```
static void* input_run(void* ctx) {
  main_loop_t* loop = (main_loop_t*)ctx;
  while(1) {
    /* 捕获系统的触摸、鼠标、键盘等输入事件，并转化为 AWTK 支持的事件 */
    event_queue_req_t req;           /* AWTK 的事件请求 */
    memset(&req, 0x00, sizeof(req));
    req.xxx = system_get_input_event();
    ...
    /* 调用本文上一小节的接口将事件分发到 GUI 主循环 */
    main_loop_queue_event(loop, &req);
  }
  return NULL;
}

ret_t platform_disaptch_input(main_loop_t *l) {
  if (run_once) { /* 保证仅执行一次，创建事件分发线程 */
      system_create_thread(input_run);
  }
}
```

```
#include "main_loop/main_loop_raw.inc"
```

需要注意的是，本章节中的示例代码主要描述输入设备移植的步骤逻辑，具体的实现代码需要根据平台实际的设备驱动来编写，读者不必过于纠结示例代码中的实现细节。

5.2 鼠标移植教程

事件	event_type_t 标识	事件类型及参数
鼠标左键按下	EVT_POINTER_DOWN	pointer_event_t
鼠标左键弹起	EVT_POINTER_UP	pointer_event_t
鼠标右键弹起	EVT_CONTEXT_MENU	pointer_event_t
鼠标移动	EVT_POINTER_MOVE	pointer_event_t
鼠标滚轮	EVT_WHEEL	wheel_event_t
鼠标中键按下	EVT_KEY_DOWN	key_event_t，TK_KEY_WHEEL
鼠标中键弹起	EVT_KEY_UP	key_event_t，TK_KEY_WHEEL

移植鼠标设备时，用户只要获取鼠标设备信息将其转化为 AWTK 的事件，并调用 main_loop_queue_event 接口将事件分发到 main_loop 中即可，代码如下：

```
/* 获取鼠标设备信息，将其转化为 AWTK 指针事件，并分发到 main_loop 中 */
void input_dispatch_mouse(/*...*/) {
  event_queue_req_t req;   /* AWTK 的事件请求 */
  memset(&req, 0x00, sizeof(req));

  /* 获取鼠标设备信息（请根据实际驱动实现） */
  evt = system_get_mouse_event();

  /* 将鼠标信息转化为 AWTK 的指针事件，此处以指针抬起为例 */
  req.event.type = EVT_POINTER_UP;    /* 设置事件类型 */
  req.pointer_event.pressed = evt.pressed; /* 设置是否按压 */
  req.pointer_event.x = evt.x;        /* 鼠标指针的 x 坐标 */
  req.pointer_event.y = evt.y;        /* 鼠标指针的 y 坐标 */
  ...

  /* 设置事件结构大小为 pointer_event_t，其他事件类似 */
  req.event.size = sizeof(req.pointer_event);

  /* 向主循环分发一个事件 */
  main_loop_queue_event(main_loop(), &req);
}
```

5.3 触摸移植教程

事件	event_type_t 标识	事件类型及参数
触摸按下	EVT_POINTER_DOWN	pointer_event_t
触摸弹起	EVT_POINTER_UP	pointer_event_t
触摸拖动	EVT_POINTER_MOVE	pointer_event_t

触摸设备的移植与鼠标设备类似，代码如下：

```
/* 获取触摸屏设备信息，将其转化为 AWTK 指针事件，并分发到 main_loop 中 */
void input_dispatch_touch(/*...*/) {
  event_queue_req_t req   /* AWTK 的事件请求 */
  memset(&req, 0x00, sizeof(req));

  /* 获取触摸屏设备信息（请根据实际驱动实现） */
  evt = system_get_touch_event();

  /* 将触摸屏信息转化为 AWTK 的指针事件，此处以手指抬起为例 */
  req.event.type = EVT_POINTER_UP;    /* 设置事件类型 */
  req.pointer_event.pressed = evt.pressed; /* 设置是否按压 */
  req.pointer_event.x = evt.x;        /* 触摸的 x 坐标 */
  req.pointer_event.y = evt.y;        /* 触摸的 y 坐标 */
  ...

  /* 设置事件结构大小为 pointer_event_t，其他事件类似 */
  req.event.size = sizeof(req.pointer_event);

  /* 向主循环分发一个事件 */
  main_loop_queue_event(main_loop(), &req);
}
```

5.4 键盘移植教程

事件	event_type_t 标识	事件类型及参数
按键按下	EVT_KEY_DOWN	key_event_t
按键弹起	EVT_KEY_UP	key_event_t

触摸设备的移植同样与鼠标设备类似，区别在于获取键盘按键扫描码后，要将其转化成 AWTK 的 key_code_t 代码，代码如下：

```
/* 获取键盘设备信息，将其转化为 AWTK 按键事件，并分发到 main_loop 中 */
void input_dispatch_keyboard(/*...*/) {
  event_queue_req_t req   /* AWTK 的事件请求 */
  memset(&req, 0x00, sizeof(req));

  /* 获取键盘设备信息（请根据实际驱动实现） */
  evt = system_get_key_event();
```

```
  key_code = system_key_to_key_code(evt.key);

  /* 将键盘信息转化为 AWTK 的按键事件，此处以某键按下为例 */
  req.event.type = EVT_KEY_DOWN;    /* 设置事件类型 */
  req.key_event.key = key_code;     /* 设置键值TK_KEY_xxx，AWTK的键值请参考 awtk/
↪src/base/keys.h */
  ...

  /* 设置事件结构大小为 key_event_t，其他事件类似 */
  req.event.size = sizeof(req.key_event);

  /* 向主循环分发一个事件 */
  main_loop_queue_event(main_loop(), &req);
}
```

5.5 其他设备移植教程

根据以上章节中鼠标、触摸屏和键盘设备的移植教程，可以发现各类输入设备的移植流程大同小异，其核心步骤都是获取设备事件将其转化为 AWTK 支持的事件并调用 main_loop_queue_event 接口分发到 main_loop 中，区别只在于设备事件的获取与转化。

某些嵌入式平台会提供一些特殊的输入设备，比如 STM32f103ze 平台提供四个特殊按键，分别表示上、下、左、右，常用于 GUI 界面上切换控件焦点，与移植键盘设备类似，实现代码如下:

```
/* 获取特殊按键设备信息，将其转化为 AWTK 按键事件，并分发到 main_loop 中 */
void input_dispatch_4key(/*...*/) {
  bool is_press = FLASE; /* 默认按键为抬起状态 */

  /* 获取键盘设备信息（请根据实际驱动实现） */
  int key = KEY_Scan(0);

  /* 获取键值和按键状态，AWTK 的键值请参考 awtk/src/base/keys.h */
  input_dispatch_get_key_value_and_state(&key, &is_press);

  /* 向主循环分发一个按键事件 */
  main_loop_post_key_event(main_loop(), is_press, key);
}
```

6. 平台相关接口

6.1 文件系统移植

文件系统属于移植 AWTK 时的非必须功能，如果要使用文件模块的功能，比如想把 AWTK 应用的资源文件放到文件系统中管理，则需要对文件系统进行移植。在 AWTK 中，需要开启宏 **WITH_FS_RES** 来支持文件系统功能，更多关于通用文件模块描述请参考本文附录二。

移植文件系统需要实现 awtk/src/tkc/fs.h 中的相关接口，其中包括：文件操作接口（fs_file_t）、文件夹操作接口（fs_dir_t）以及文件系统操作接口（fs_t），因为接口太多就不在这里一一列举，具体详见本文附录一，完成以上适配工作后，只需实现获取文件系统对象的 os_fs 函数即可，示例代码详见下文。

1、实现 fs.h 中的相关接口

先实现文件读写操作接口 fs_file_vtable_t，以及文件夹遍历接口 fs_dir_vtable_t，代码片段如下：

```
static int32_t fs_os_file_read(fs_file_t* file, void* buffer, uint32_t size) {
  FILE* fp = (FILE*)(file->data);
  return (int32_t)fread(buffer, 1, size, fp);
}

static int32_t fs_os_file_write(fs_file_t* file, const void* buffer, uint32_t size)␣
↪{
  FILE* fp = (FILE*)(file->data);
  return fwrite(buffer, 1, size, fp);
}

/* 构建 fs_file_t 对象的虚函数表 */
static const fs_file_vtable_t s_file_vtable = {
  .read = fs_os_file_read,
  .write = fs_os_file_write,
  ...
}

/* 构建 fs_dir_t 对象的虚函数表 */
static const fs_dir_vtable_t s_dir_vtable = {
  .read = fs_os_dir_read,
  ...
};
```

2、实现 os_fs 函数

接着实现文件系统对象 fs_t，文件对象打开成功后要与上述 fs_file_vtable_t 对象挂接一起。最后实现 os_fs 函数返回全局 fs_t 对象即可，代码片段如下：

```c
/* 打开文件 */
static fs_file_t* fs_os_open_file(fs_t* fs, const char* name, const char* mode) {
  /* 此处使用 fopen 接口实现为例，具体请根据相关平台实现 */
  fs_file_t* f = TKMEM_ZALLOC(fs_file_t);
  f->vt = &s_file_vtable;  /* 挂接fs_file_vtable_t对象 */
  f->data = (void*)fopen(name, mode);
  return f;
}

/* 打开目录 */
static fs_dir_t* fs_os_open_dir(fs_t* fs, const char* name) {
  fs_dir_t* d = TKMEM_ZALLOC(fs_dir_t);
  d->vt = &s_dir_vtable;  /* 挂接fs_dir_vtable_t对象 */
  d->data = opendir(name);
  return d;
}

/* 构建 fs_t 对象：将实现的文件系统操作函数赋值给虚表中对应的函数指针 */
static const fs_t s_os_fs = {.open_file = fs_os_open_file,
                             .open_dir = fs_os_open_dir,
                             ...};
/* 获取 fs_t 对象 */
fs_t* os_fs(void) {
  return (fs_t*)&s_os_fs;
}
```

此外，在嵌入式平台中，有时没有 posix 兼容的文件系统 API，需要把一些文件系统实现包装成 awtk/src/tkc/fs.h 中的接口，此时可以直接使用 AWTK 提供的文件系统适配器 awtk-fs-adapter[10]，目前支持的文件系统如下：

- FATFS 主要用于访问 TF card。
- SPIFFS 主要用于访问 Nor Flash。

6.2 多线程相关资源

AWTK 本身只有一个 GUI 线程，无需用到多线程相关资源，因此线程相关功能可根据项目实际需求决定是否进行移植。AWTK 中的多线程相关功能的对应的接口文件详见下表：

函数库	接口文件	说明
thread	src/tkc/thread.h	线程
mutex	src/tkc/mutex.h	互斥锁
cond	src/tkc/cond.h	条件变量
semaphore	src/tkc/semaphore.h	信号量

需要注意的是，在应用程序中，如果需要在非 GUI 线程向主循环（GUI 线程）发送事件则必须参考本文 5.5.2 章节的内容移植**互斥锁**，如果在非 GUI 线程中调用 AWTK 内存管理器（awtk/src/tkc/mem.h）中的相关函数，除了需移植**互斥锁**之外，还需要参考本文 5.2.3

[10] https://github.com/zlgopen/awtk-fs-adapter

章节内容移植信号量。

下文中的示例代码均以 pthread 为例，具体代码实现请参考：awtk/ platforms/ pc/ thread_with_pthread.c。

注: AWTK 提供了常见 RTOS 的多线程资源实现，具体详见 awtk/src/platforms 目录下对应平台的实现代码，用户可直接加入工程使用。

6.2.1 线程

移植 AWTK 的线程需要实现 awtk/src/tkc/thread.h 中的 _tk_thread_t 结构体以及相关接口，结构体代码如下：

```
struct _tk_thread_t {
  /* 平台无关成员（移植时必须定义） */
  void*             args;       /* 线程回调函数上下文 */
  tk_thread_entry_t entry;      /* 线程回调函数 */
  bool_t            running;    /* 线程是否正在执行 */
  const char*       name;       /* 线程的名称 */
  uint32_t          stack_size; /* 线程的栈大小 */
  uint32_t          priority;   /* 线程的优先级 */

  /* 平台相关成员（移植到其他平台时，请根据实际需求定义） */
  pthread_t thread;  /* 此处以 pthread 为例 */
};
```

线程相关接口说明详见下表：

接口	说明	备注
tk_thread_create	创建 thread 对象	保存上下文 ctx 和回调函数 entry
tk_thread_set_name	设置线程名称	
tk_thread_set_stack_size	设置线程栈大小	
tk_thread_set_priority	设置线程优先级	某些平台可能不支持
tk_thread_get_priority_from_platform	获取平台相关的优先级	某些平台可能不支持
tk_thread_start	启动线程	将 running 属性设置为 TRUE
tk_thread_join	等待线程退出	将 running 属性设置为 FALSE
tk_thread_get_args	获取线程的参数	
tk_thread_destroy	销毁 thread 对象	
tk_thread_self	获取当前线程的原生句柄	

6.2.2 互斥锁

移植 AWTK 的互斥锁需要实现 awtk/src/tkc/mutex.h 中的 _tk_mutex_t 结构体以及相关接口，结构体代码如下：

```
struct _tk_mutex_t {
  /* 定义一个互斥锁，请根据实际平台实现 */
  pthread_mutex_t mutex;  /* 此处以 pthread 为例 */
};
```

互斥锁相关接口详见下表：

接口	说明
tk_mutex_create	创建互斥锁对象
tk_mutex_lock	加锁
tk_mutex_try_lock	尝试加锁
tk_mutex_unlock	解锁
tk_mutex_destroy	销毁互斥锁对象

6.2.3 信号量

移植 AWTK 的信号量需要实现 awtk/src/tkc/semaphore.h 中的 _tk_semaphore_t 结构体以及相关接口，结构体代码如下：

```
struct _tk_semaphore_t {
  /* 定义一个信号量，请根据实际平台实现 */
  sem_t* sem;  /* 此处以 sem_t 为例 */
};
```

信号量相关接口详见下表：

接口	说明
tk_semaphore_create	创建信号量对象
tk_semaphore_wait	获取资源
tk_semaphore_post	释放资源
tk_semaphore_destroy	销毁信号量对象

6.2.4 条件变量

移植 AWTK 的条件变量需要实现 awtk/src/tkc/cond.h 中的 _tk_cond_t 结构体以及相关接口，结构体代码如下：

注: 条件变量通常与互斥锁结合使用，一般来说，移植条件变量前需要先移植互斥锁。

```
struct _tk_cond_t {
  /* 定义一个条件变量，请根据实际平台实现 */
  pthread_cond_t cond;  /* 此处以 pthread 为例 */
};
```

条件变量相关接口详见下表：

接口	说明
tk_cond_create	创建条件变量对象
tk_cond_wait	等待
tk_cond_wait_timeout	等待指定时间

续上表

接口	说明
tk_cond_signal	唤醒
tk_cond_destroy	销毁条件变量对象

7. 硬件加速

7.1 G2D 硬件加速适配

7.1.1 AWTK 绘图流程

G2D 硬件加速是指将计算量较大的图像处理工作分配给专门的硬件外设来处理，减轻 CPU 的计算量，以此提高图像绘制的性能。

不同硬件平台的硬件加速外设不一样，其实现方法也有区别，在 AWTK 中，用户需要实现 awtk/src/base/g2d.h 文件中的相关接口，详见下表，然后开启宏 **WITH_G2D** 即可支持硬件加速，更多关于硬件加速模块的说明详见本文附录二。

函数名称	说明	备注
g2d_fill_rect	用颜色填充指定的区域	
g2d_copy_image	把图片指定区域拷贝到 Framebuffer 中	
g2d_rotate_image	把图片指定区域进行旋转并拷贝到 Framebuffer 的相应区域	本函数主要用于辅助实现横屏和竖屏的切换，一般支持 90 度、180 度和 270 度旋转即可
g2d_blend_image	把图片指定区域渲染到 Framebuffer 的指定区域，若两个区域大小不同则进行缩放	如果硬件设备不支持缩放或全局 Alpha 融合算法（绘制半透明图像），请返回 RET_NOT_IMPL 使用软件渲染

为方便读者理解，此处以绘制一张半透明的背景图为例，介绍 AWTK 是如何支持硬件加速的，绘制流程图如下：

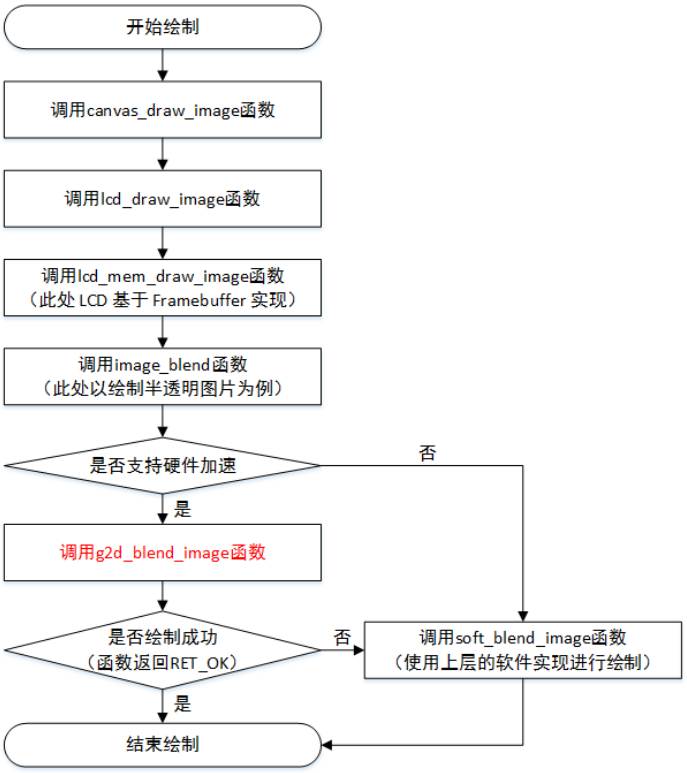

图 7.1　半透明背景图绘制流程

特别备注：

1. 上图中的 canvas_draw_image 函数是 AWTK 提供的绘图功能，其接口详见：awtk/src/base/canvas.h。
2. 上图中的 lcd_draw_image 函数是 AWTK 提供的 LCD 相关功能，其接口详见：awtk/src/base/lcd.h。
3. AWTK 提供了基于 Framebuffer 的 LCD 缺省实现，代码详见：awtk/src/lcd/lcd_mem.inc。
4. 基于 Framebuffer 实现的 LCD 通常会调用 awtk/src/blend/image_g2d.h 中的接口进行绘制。

7.1.2　STM32 平台硬件加速案例

目前，AWTK 内置了 STM32 系列平台 G2D 硬件加速的实现，代码详见：awtk/src/blend/stm32_g2d.c，只需定义下面的宏即可启用该功能：

```
/* 如果支持 STM32 硬件加速，请定义本宏（AWTK 会定义真正起作用的宏 WITH_G2D） */
#define WITH_STM32_G2D 1
```

STM32 平台通过外设 DMA2D（Direct Memory Access 2D）支持硬件加速，此处以实现 g2d.h 文件中的 g2d_fill_rect 函数为例，介绍如何通过 DMA2D 实现硬件加速，其他接口的实现方法类似，详情可参考：awtk/src/blend/stm32_g2d.c。

```
/* 用颜色填充指定的区域 */
ret_t g2d_fill_rect(bitmap_t* fb, const rect_t* dst, color_t c) {
  ...
  /* 计算像素占用字节、颜色格式以及填充的颜色值 */
```

```
  get_output_info(fb->format, &o_pixsize, &o_format, &color);

  /* 为修改数据锁定位图缓冲区 */
  uint8_t* fb_data = bitmap_lock_buffer_for_write(fb);

  /* 计算行偏移以及填充区域的起始地址 */
  uint16_t o_offline = fb->w - dst->w;
  uint32_t o_addr = ((uint32_t)fb_data + o_pixsize * (fb->w * dst->y + dst->x));

/************************************************************
      以下为 DMA2D 的相关用法，具体请根据实际平台实现
************************************************************/
  /* 使能 DMA2D，请根据实际情况实现 */
  __HAL_RCC_DMA2D_CLK_ENABLE();

  /* 先停止 DMA2D，并设置硬件加速模式为：从寄存器到内存 */
  DMA2D->CR &= ~(DMA2D_CR_START);
  DMA2D->CR = DMA2D_R2M;

  /* 配置 DMA2D，例如此处需进行以下配置，请根据实际情况实现 */
  DMA2D->OPFCCR = o_format;   /* 设置颜色格式 */
  DMA2D->OOR = o_offline;     /* 设置行偏移 */
  DMA2D->OMAR = o_addr;       /* 设置目标缓冲区地址 */
  DMA2D->OCOLR = color;       /* 设置填充颜色 */
  DMA2D->NLR = h | (w << 16); /* 设置行数 */

  /* 启动 DMA2D 进行填充，并等待它工作结束 */
  DMA2D->CR |= DMA2D_CR_START;
  DMA2D_WAIT
/************************************************************/

  /* 绘制完成，解锁位图缓冲区 */
  bitmap_unlock_buffer(fb);

  return RET_OK;
}
```

7.2 硬件图像解码适配

7.2.1 AWTK 解码图片的流程

AWTK 默认采用 stb 库进行软件图像解码，全靠 CPU 计算。硬件解码是指将图像解码的工作分配给专门的硬件来处理，减轻 CPU 的计算量，从而提高图像绘制的性能。

在 AWTK 中，用户需要自己实现硬件解码的接口，然后调用 image_loader_register 函数注册该接口，后续 AWTK 程序在进行解码时会自动调用该接口解码图片。

```
/**
 * @method image_loader_register
```

```
 * 注册图片加载器。
 *
 * @annotation ["static"]
 * @param {image_loader_t*} loader loader对象。
 *
 * @return {ret_t} 返回RET_OK表示成功，否则表示失败。
 */
ret_t image_loader_register(image_loader_t* loader);
```

此处以获取 JPG 图片为例展示 AWTK 是如何获取图片的：

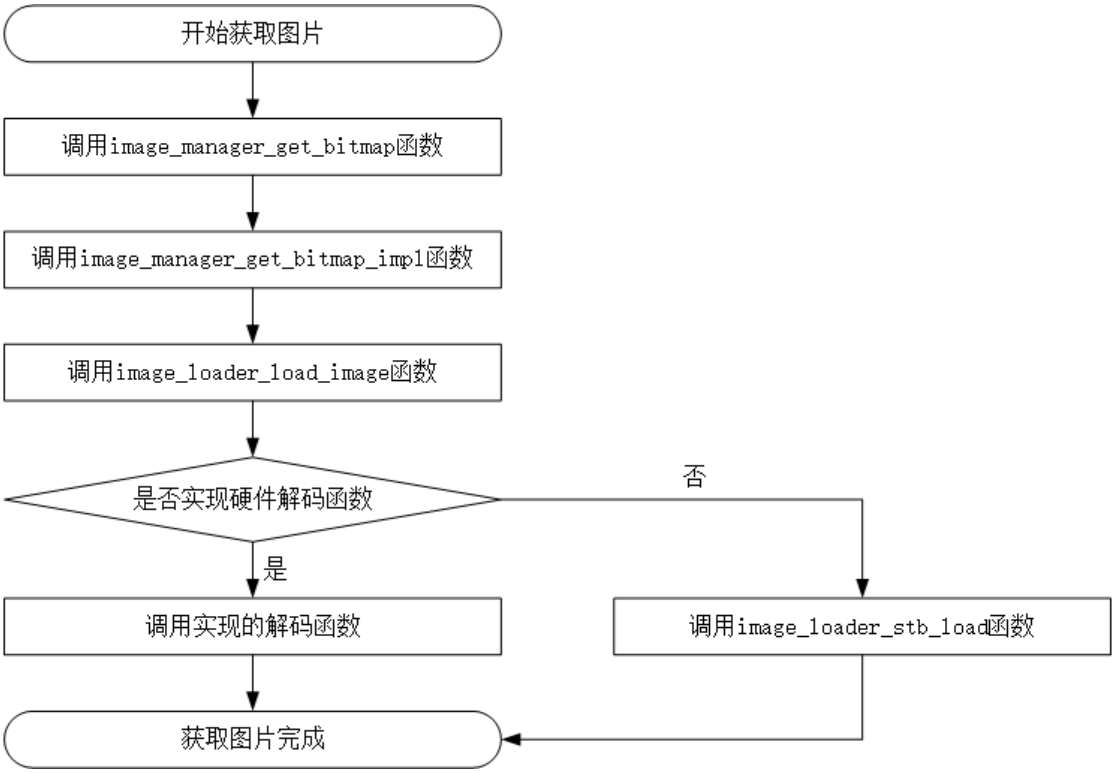

图 7.2　获取图片流程

注: 注：

1. 上图中的 image_manager_get_bitimp 函数是 AWTK 提供的获取图片功能，接口详见：awtk/src/base/image_manager.c。
2. 上图中的 image_loader_load_image 函数是 AWTK 提供的加载图片接口，接口详见：awtk/src/base/image_loader.h。

7.2.2　STM32 平台硬件图像解码案例

本节以 AWTK 针对 STM32F767igtx 为例，大致讲解如何将硬件解码接口集成到 AWTK 中，移植层源码可前往 GitHub[11] 下载。

步骤一：用户需根据实际使用的硬件平台实现 JPG 图像硬件解码的接口，然后定义一个 image_loader_t 的对象，重载对象中的 load 函数，代码如下：

[11] https://github.com/zlgopen/awtk-stm32f767igtx-raw

```
/* awtk-stm32f767igtx-raw/awtk-port/stm32_jpg_image_loader.c */
/* JPG硬件解码函数 */
static ret_t image_loader_stm32_jpg_load(image_loader_t *l,
                                         const asset_info_t *asset,
                                         bitmap_t *image) {
/* 实现JPG硬件解码，具体实现由硬件决定 */
}

static const image_loader_t stm32_jpg_image_loader = {
    .load = image_loader_stm32_jpg_load};

image_loader_t *image_loader_stm32_jpg() {
  return (image_loader_t *)&stm32_jpg_image_loader;
}
```

步骤二：在 LCD 初始化时顺便注册上一步骤中创建的 image_loader_t 对象，代码如下：

```
/* awtk-stm32f767igtx-raw/awtk-port/main_loop_stm32_raw.c */
/* 在LCD初始化时调用注册函数 */
lcd_t* platform_create_lcd(wh_t w, wh_t h) {
  image_loader_register(image_loader_stm32_jpg());
  return stm32f767_create_lcd(w, h);
}
```

7.3 图像专用内存

某些平台的 G2D 硬件加速需要特定的内存，不能直接使用 malloc 开辟出来的内存。例如：在 M1107 嵌入式 Linux 下的 G2D 硬件加速使用的内存需要真实的物理地址，不能是 malloc 出来的虚拟内存地址。AWTK 为了解决这个问题，提供了一种专用图像内存的解决方案。

7.3.1 工作原理

AWTK 内部加载图片（如 png、jpg 等）时，会把图片解码后的位图内容存放到 graphic_buffer_t 所管理的内存中，后续要显示图片时 AWTK 则会通过 graphic_buffer_t 获取对应位图数据并拷贝到目标 Framebuffer 中。

默认 graphic_buffer_t 对象所管理的内存是使用系统 malloc 分配出来的，如果要让位图数据存放到专用内存，则需要替换掉默认 graphic_buffer_t 的行为，改为通过专用 API 分配出来的内存。因此，需要重载 graphic_buffer_t 类成员函数实现多态，让 AWTK 在分配位图内存时使用我们自己内存分配函数。

基本步骤：

1. 创建一个 graphic_buffer_physical.c 文件，并且实现 graphic_buffer_create_for_bitmap、graphic_buffer_create_with_data 函数，重载 graphic_buffer_t 的多态函数指针。
2. 移除 `awtk/src/graphic_buffer/graphic_buffer_default.c` 文件的编译。
3. 把 graphic_buffer_physical.c 文件放入项目中编译。

下面将详细解释位图缓存结构，及给出一个实际移植例子。

7.3.2 位图缓存 graphic_buffer_t

如果希望 AWTK 使用专用内存的话，就必须了解 AWTK 的位图缓存（graphic_buffer_t 类型）。该类型是用于分配和存放 AWTK 位图数据的，所以开辟专用内存的话，需要重新实现该类型的成员函数，并且要让 AWTK 创建和获取位图缓存时对接到这个新的 graphic_buffer_t 对象。

```
/* awtk/base/graphic_buffer.h */
struct _graphic_buffer_t;
typedef struct _graphic_buffer_t graphic_buffer_t;

typedef uint8_t* (*graphic_buffer_lock_for_read_t)(graphic_buffer_t* buffer);
typedef uint8_t* (*graphic_buffer_lock_for_write_t)(graphic_buffer_t* buffer);
typedef uint32_t (*graphic_buffer_get_physical_width_t)(graphic_buffer_t* buffer);
typedef uint32_t (*graphic_buffer_get_physical_height_t)(graphic_buffer_t* buffer);
typedef uint32_t (*graphic_buffer_get_physical_line_length_t)(graphic_buffer_t*
↪buffer);
typedef ret_t (*graphic_buffer_unlock_t)(graphic_buffer_t* buffer);
typedef ret_t (*graphic_buffer_attach_t)(graphic_buffer_t* buffer, void* data,
↪uint32_t w, uint32_t h);
typedef bool_t (*graphic_buffer_is_valid_for_t)(graphic_buffer_t* buffer, bitmap_t*
↪bitmap);
typedef ret_t (*graphic_buffer_destroy_t)(graphic_buffer_t* buffer);

/* 省略无关代码.... */
typedef struct _graphic_buffer_vtable_t {
  graphic_buffer_lock_for_read_t lock_for_read;
  graphic_buffer_lock_for_write_t lock_for_write;
  graphic_buffer_unlock_t unlock;
  graphic_buffer_attach_t attach;
  graphic_buffer_is_valid_for_t is_valid_for;
  graphic_buffer_destroy_t destroy;
  graphic_buffer_get_physical_width_t get_width;
  graphic_buffer_get_physical_height_t get_height;
  graphic_buffer_get_physical_line_length_t get_line_length;
} graphic_buffer_vtable_t;

/**
 * @class graphic_buffer_t
 * graphic_buffer。
 */
struct _graphic_buffer_t {
  const graphic_buffer_vtable_t* vt;
};
```

从 graphic_buffer_t 类型来看，该类型只有一个虚表（graphic_buffer_vtable_t 对象），就是为了给用户多态使用的，我们可以通过下表了解虚表对象中的各个函数指针具体的作用是什么：

函数指针名字	作用
lock_for_read	用于读取数据的时候加锁使用的，在某些情况下位图数据需要加锁处理。
lock_for_write	用于写入数据的时候加锁使用的，在某些情况下位图数据需要加锁处理。
unlock	用于解锁上面的 lock_for_read 和 lock_for_write。
attach	给 graphic_buffer_t 对象直接赋予外部的内存地址。
is_valid_for	判断该 graphic_buffer_t 对象是否有效。
destroy	释放 graphic_buffer_t 对象
get_width	获取 graphic_buffer_t 对象物理宽
get_height	获取 graphic_buffer_t 对象物理高
get_line_length	获取 graphic_buffer_t 对象物理行长

上面讲的都是 graphic_buffer_t 对象创建之后的成员函数指针，而 graphic_buffer_t 对象是通过下面两个函数来创建的：

函数名字	作用
graphic_buffer_create_for_bitmap	创建一个 graphic_buffer_t 对象，给该对象分配一块内存地址。
graphic_buffer_create_with_data	创建一个 graphic_buffer_t 对象，并且绑定外部内存地址。

AWTK 的位图对象（bitmap_t 对象）默认是会使用 graphic_buffer_create_for_bitmap 函数来创建出来的。

而在有固定内存的情况（例如：FrameBuffer），又想使用 AWTK 的位图对象的时候，就使用 graphic_buffer_create_with_data 把外部固定内存地址绑定到 AWTK 的位图对象中。

7.3.3 awtk-linux-fb 的移植案例

本节以 awtk-linux-fb 移植层为例，大致讲解如何将实现位图使用专用内存的解决方案。

步骤一：在 `awtk-linux-fb/awtk-port` 目录下，创建一个 `graphic_buffer_physical.c` 文件，并且在该文件下，实现了 graphic_buffer 类型的多态。

```
/* graphic_buffer_physical.c */

/**
 * @class graphic_buffer_physical_t
 * graphic_buffer default
 */
typedef struct _graphic_buffer_physical_t {
  graphic_buffer_t graphic_buffer;

  uint8_t* data;
  uint32_t w;
  uint32_t h;
  uint32_t line_length;
  bitmap_format_t format;
} graphic_buffer_physical_t;

/* 此处简单实现 graphic_buffer_physical_t 对象中的虚表函数 */
static bool_t graphic_buffer_physical_is_valid_for(graphic_buffer_t* buffer,␣
↪bitmap_t* bitmap) {
```

```
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return_value_if_fail(b != NULL && bitmap != NULL, FALSE);
  if (bitmap->orientation == LCD_ORIENTATION_0 || bitmap->orientation == LCD_
↪ORIENTATION_180) {
    return b->w == bitmap->w && b->h == bitmap->h;
  } else {
    return b->w == bitmap->h && b->h == bitmap->w;
  }
}

static uint8_t* graphic_buffer_physical_lock_for_read(graphic_buffer_t* buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return b->data;
}

static uint8_t* graphic_buffer_physical_lock_for_write(graphic_buffer_t* buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return b->data;
}

static ret_t graphic_buffer_physical_unlock(graphic_buffer_t* buffer) {
  return RET_OK;
}

static ret_t graphic_buffer_physical_attach(graphic_buffer_t* buffer, void* data,␣
↪uint32_t w, uint32_t h) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  b->w = w;
  b->h = h;
  b->data = data;
  return RET_OK;
}

static ret_t graphic_buffer_physical_destroy(graphic_buffer_t* buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  /* 判断 data_head 来释放内存的 */
  if(b->data_head != NULL) {
    mem_physical_free(b->data_head);
  }
  TKMEM_FREE(b);
  return RET_OK;
}

static uint32_t graphic_buffer_physical_get_physical_width(graphic_buffer_t*␣
↪buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return b->w;
}
```

```
static uint32_t graphic_buffer_physical_get_physical_height(graphic_buffer_t*
↪buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return b->h;
}

static uint32_t graphic_buffer_physical_get_physical_line_length(graphic_buffer_t*
↪buffer) {
  graphic_buffer_physical_t* b = (graphic_buffer_physical_t*)(buffer);
  return b->line_length;
}

static const graphic_buffer_vtable_t s_graphic_buffer_physical_vtable = {
    .lock_for_read = graphic_buffer_physical_lock_for_read,
    .lock_for_write = graphic_buffer_physical_lock_for_write,
    .unlock = graphic_buffer_physical_unlock,
    .attach = graphic_buffer_physical_attach,
    .is_valid_for = graphic_buffer_physical_is_valid_for,
    .get_width = graphic_buffer_physical_get_physical_width,
    .get_height = graphic_buffer_physical_get_physical_height,
    .get_line_length = graphic_buffer_physical_get_physical_line_length,
    .destroy = graphic_buffer_physical_destroy};

graphic_buffer_t* graphic_buffer_create_with_data(const uint8_t* data, uint32_t w,
↪uint32_t h, bitmap_format_t format) {
  /* 创建 graphic_buffer 对象 */
  graphic_buffer_physical_t* buffer = TKMEM_ZALLOC(graphic_buffer_physical_t);
  /* 把外部的内存地址配置给 graphic_buffer 对象 */
  buffer->data = (uint8_t*)data;
  /* 配置 graphic_buffer 对象的物理宽高等数据 */
  buffer->w = w;
  buffer->h = h;
  buffer->line_length = bitmap_get_bpp_of_format(format) * w;
  buffer->graphic_buffer.vt = &s_graphic_buffer_physical_vtable;

  return (graphic_buffer_t*)(buffer);
}

ret_t graphic_buffer_create_for_bitmap(bitmap_t* bitmap) {
  uint8_t* data = NULL;
  /* 创建 graphic_buffer 对象 */
  graphic_buffer_physical_t* buffer = TKMEM_ZALLOC(graphic_buffer_physical_t);
  /* 配置 graphic_buffer 对象的物理宽高等数据 */
  buffer->w = bitmap->w;
  buffer->h = bitmap->h;
  buffer->line_length = bitmap_get_line_length(bitmap);
  buffer->graphic_buffer.vt = &s_graphic_buffer_physical_vtable;
  /* 根据具体平台来分配位图专用内存，这里是分配物理内存地址，这里特别要给 data_head
↪赋值，用于释放的 */
```

```
  buffer->data = buffer->data_head = mem_physical_alloc(bitmap->h * buffer->line_
↪length);

  bitmap->buffer = buffer;
  return buffer != NULL ? RET_OK : RET_OOM;
}
```

步骤二：修改 awtk-linux-fb/awtk-port/SConscript 脚本，让上面创建的 graphic_buffer_physical.c 文件加入到项目中编译，在 SOURCES 对象中增加 graphic_buffer_physical.c 文件。

```
# awtk-linux-fb/awtk-port/SConscript

# 省略其他无关代码
SOURCES = [
  'input_thread/mouse_thread.c',
  'input_thread/input_thread.c',
  'input_thread/input_dispatcher.c',
  'lcd_linux/lcd_linux_fb.c',
  'lcd_linux/lcd_linux_drm.c',
  'lcd_linux/lcd_linux_egl.c',
  'lcd_linux/lcd_mem_others.c' ,
  'main_loop_linux.c',

  # 增加编译 graphic_buffer_physical.c 文件
  'graphic_buffer_physical.c',
  ]
```

步骤三：修改 awtk-linux-fb/awtk_config.py 脚本，禁用 AWTK 默认的 graphic_buffer 类型，找到 GRAPHIC_BUFFER='default'，修改为 GRAPHIC_BUFFER='custom'。

```
# awtk-linux-fb/awtk_config.py

#GRAPHIC_BUFFER='default'
GRAPHIC_BUFFER='custom'
```

第三部分

裁剪篇

8. 裁剪篇导读

本文第 3 章到第 7 章的内容介绍了如何移植 AWTK，从本章节开始，主要介绍如何裁剪 AWTK，其中包括 AWTK 的资源占用情况，以及在中资源平台和低资源平台上的裁剪案例。阅读本篇内容能帮助读者了解 AWTK 的各个功能模块及其资源占用情况，以及如何对它们进行裁剪。

注: 本文中展示的数据基于 AWTK 1.6.1 测试得到，其他版本可能存在差异。

9. AWTK 资源占用

程序的 ROM 和 RAM 资源占用一直都是嵌入式开发关心的问题，ROM 上通常存放代码、常量和应用资源数据，RAM 内存通常存放 Framebuffer、AWTK 框架功能以及应用程序的动态内存。应用相关的资源占用与应用的规模强烈相关，而 Framebuffer 则与平台的缓冲数量、屏幕分辨率以及颜色深度等相关，用户可以自行估算出来。本章仅讨论 AWTK 框架功能的资源占用（AWTK 核心和非核心模块代码 ROM 以及 RAM 的占用情况）。

图 9.1　资源占用概述图

9.1　资源占用

AWTK 中有核心代码和非核心代码，核心代码就是不能裁剪，必须要添加到项目中的代码，而非核心代码又分为控件和可裁剪的功能模块，所以 AWTK 资源占用可以简单分为三部分：无法裁剪的核心功能模块，可裁剪的功能模块以及控件。本章节讨论是这三部分的资源占用情况。

测试说明：

1. 本章节的内存占用测试是在 32 位的 MCU 上进行的，在 64 位的 MCU 上测试结果可能会不一样。
2. 本章节的 ROM 占用测试是在无调试信息以及编译选项为 -Os 的情况下进行的，数据通过 arm-none-eabi-size.exe 测试得到。

9.1.1　核心功能模块的资源占用

将 AWTK 的非核心代码全部裁剪掉，核心代码的 ROM 占用约为 100KB，RAM 占用约为 3KB，AWTK 核心模块介绍及其内存占用请参考本文附录三。

需要注意的是：

1. 由于 AWTK 存在图片缓冲机制，显示全新图片时，如果内存足够，图片管理器的内存都会增加。
2. 不同的 lcd_t 类型内存占用都不一样，lcd_t 对象一般仅占几百个字节，真正占用内存的是 Framebuffer。
3. 要注册 UI 控件还需要加上额外的 3KB 控件管理器内存。

注：在内存资源紧缺的平台上，用户可以在适当时机（比如切换窗口时），手动调用 image_manager_unload_unused 或者 image_manager_unload_all 函数来及时清除图片缓存，避免图片缓冲机制

与其他业务逻辑抢资源。

9.1.2 可裁剪的功能模块的资源占用

功能模块	宏	ROM 占用	RAM 占用
图片解码模块	WITH_STB_IMAGE	约 35.9KB	PNG：解码瞬时峰值内存占用约为两份位图数据。JPG：18KB + 解码瞬时峰值内存占用约为三份位图数据。GIF：解码后大约为所有帧的位图数据总和
字体解码模块	WITH_STB_FONT	约 15.1KB	stb 字库对象为：3.2KB + 用到的字模缓存数据
标准的 UNICODE 换行算法模块	WITH_UNICODE_BREAK	约 47.3KB	大于 1.39KB
Google 拼音输入法功能	WITHOUT_INPUT_METHOD、WITH_NULL_IM	约 24.5KB + 1.1MB（输入法资源）	大于 700KB
AGGE 矢量画布模块	WITH_NANOVG_AGGE	约 10KB	大于 30KB（18.8KB + 缓存数据）
窗口动画功能	WITHOUT_WINDOW_ANIMATORS	约 3.3KB	520B + 两份显存大小的瞬时峰值
布局算法模块	WITHOUT_LAYOUT	大约 1.9KB	128B
控件动画功能	WITHOUT_WIDGET_ANIMATORS	大约 3KB	120B
剪切板模块	WITHOUT_CLIPBOARD	大约 232B	16B
对话框高亮策略模块	WITHOUT_DIALOG_HIGHLIGHTER	大约 1.2KB	60B
加载文件系统资源模块	WITH_FS_RES	大约 5.5KB	0B
扩展控件	WITHOUT_EXT_WIDGETS	大约 90KB	0B
基本控件	AWTK_NOGUI	大约 12.3KB	0B

需要注意的是：

1. WITH_ 开头的宏用于开启某个功能模块，WITHOUT_ 开头的宏用于裁剪某个功能模块，通常 AWTK 默认开启的功能，需要用 WITHOUT_ 开头的宏来裁剪，默认不开启的功能，需要用 WITH_ 开头的宏来开启。
2. 图片解码器最大的问题是解码时，峰值内存较高，尤其是在解码 JPG 时，如果 JPG 类型为 YUV，峰值会根据 YUV 的采样比来决定，上表中默认采样比为 1，如果采样比大于 1，峰值内存只会更大。
3. JPG 和 PNG 解码完成后，会释放解码时开辟的多余的峰值内存，将一份解码后的位图的数据存放到 AWTK 的图片缓冲中。
4. GIF 格式的解码仅在开启图片解码模块以及通用文件系统模块时支持，并且解码后的 GIF 所有帧都会被拼接到一张 bitmap 中，所以 GIF 的内存占用为所有帧的位图数据总和。

5. Truetype 字体解码器本身不占什么内存，但由于缓存机制，字模的内存占用会随着显示字符的增多而变大，每个字模内存约等于字号 * 字号所占字节。
6. Nanovg 和 Agge 是 AWTK 自带的第三方的矢量画布类库。
7. 在矢量画布中，缓存数据分别在 nanovg 和 AGGE 中，nanovg 创建的时候默认包含了 128 个 point（约 32B），16 个 path（约 40B）和 256 个 vertex（约 16B），但如果运行过程中同一帧内的缓存数据超过默认的缓存数量，就会重新分配缓冲数据，这会增加矢量画布的内存占用量。
8. AGGE 创建时没有默认缓存数据，它绘制前会把 nanovg 的缓存数据转为坐标点（约 12B），如果当前的坐标点个数超过之前缓存的坐标点个数，就会重新分配缓存坐标点数组，同时如果让 AGGE 画一个全新的图片，就会增加一个贴图缓存（约 32B），所以 AGGE 运行时会增加矢量画布的内存占用量。

9.1.3 控件的资源占用

关于控件的资源占用：

1. 除了控件自身的最低运行内存占用以外，修改控件相关属性以及设置 inline_style（在 UI 文件中增加的风格属性就是 inline_style），都可能导致控件的内存变大，同时每创建一个控件就会多占用一份内存，每个控件的内存都是独立的。
2. 控件占用的 ROM 计算比较简单，每种控件的 ROM 占用都是固定的，而且还有一些控件是公用同一个代码文件的，所以多个控件占用同一份 ROM。

下面列出一些常用的控件资源占用列表，在附录四中会列出所有 AWTK 的控件资源占用列表。

控件名称	ROM 占用	最少 RAM 占用
dialog	大约 1.2KB	424B
window	567B	412B
image	大约 1KB	168B
button	大约 1.7B	160B
label	大约 3.3KB	148B
progress_bar	大约 2.5KB	160B
check_button	大约 1.6KB	140B
radio_button	大约 1.6KB	140B
system_bar	983B	444B
system_bar_bottom	983B	444B
view	509B	140B
edit	大约 19.5KB	868B
combo_box	大约 5.5KB	大约 1.2KB
combo_box_item	大约 1.3KB	144B
scroll_view	大约 6.6KB	272B
list_view	大约 3KB	160B
list_item	大约 1KB	152B
scroll_bar	大约 5.8KB	164B
scroll_bar_d	大约 5.8KB	920B
scroll_bar_m	大约 5.8KB	164B

续上表

控件名称	ROM 占用	最少 RAM 占用
combo_box_ex	601B	大约 1.4KB

9.2 评估项目资源占用方法

9.2.1 评估项目 AWTK 相关 ROM 的大小

只保留 AWTK 的核心代码和基础控件时 ROM 约为 130KB，根据实际需求再加上使用的控件和功能模块的 ROM 大小。如果没有定义 **WITHOUT_EXT_WIDGETS** 和 **AWTK_NOGUI** 宏，那么控件占用的 ROM 分别为 90KB 和 12.3KB，不论用到多少个控件，都占用相同的 ROM。

9.2.2 评估项目 AWTK 相关 RAM 内存大小

假设 LCD 是 800 * 480 的 16 位色屏幕，采用双 Framebuffer 机制，计算内存的步骤如下：

1. 双 Framebuffer 的显存为：800 * 480 * 2 * 2 = 1.5MB。
2. 估算 AWTK 使用的总 Heap 大小：

   (1) 计算 Heap 中的小对象内存池大小，详见附录三。

   (2) 统计要使用的可裁剪的功能模块，计算所用到的模块的内存总和。

   (3) 统计要使用的控件，以及同一画面中有多少个控件，计算所有显示的控件的内存占用总和。

   (4) 如果开启了图片解码模块，则需要计算解码峰值，以及同一个画面显示的所有贴图缓存的内存总和。

   (5) 如果开启了字库解码模块，则需要计算同一个画面显示的所有字符缓存的内存总和。

   (6) 如果开启窗口动画模块，则需要分配额外两份 Framebuffer 大小的内存，用于保存前后两个窗口的截图。

   (7) 程序运行过程中可能会出现内存碎片，汇总好以上内存占用量之后，再乘以系数 1.2 到 2，就可以得出来项目大概需要多少内存。

9.2.3 评估项目其他资源占用

不同领域的项目应用程序规模差异较大，用户需要自行估算其他业务逻辑的 ROM 和 RAM 占用，以及用到的图片字体等资源数据的 ROM 占用情况。

在项目资源中，通常占 ROM 最多的通常是图片资源和字体资源，其中字体资源可以通过 AWTK Designer[12] 的资源管理器进行裁剪。

[12] https://awtk.zlg.cn/

9.3 平台资源和功能模块裁剪建议

9.3.1 ROM 和功能模块

该表中仅代表 AWTK 本身占用的 ROM，不包括应用程序自己的的代码以及资源数据。

平台	ROM 大小	建议裁剪 (禁用) 的 AWTK 功能
高资源平台	(2MB，+∞]	无
中资源平台	(256KB，2MB]	拼音输入法功能（包括输入法资源，约占 1.1MB ROM）
低资源平台	(128KB，256KB]	除位图字体功能，位图贴图功能以及部分基本控件之外的所有功能

9.3.2 内存和功能模块

该表中仅代表 AWTK 本身占用的运行内存，不包括显存和应用程序所需的其他业务内存。内存比较小的情况下尽量使用相同字号和相同字库的字符。

平台	RAM 大小	建议裁剪 (禁用) 的 AWTK 功能
高资源平台	(4MB，+∞]	无
中资源平台	(2MB，4MB]	拼音输入法功能（包括输入法资源）
中资源平台	(1MB，2MB]	播放窗口动画模块，对话框高亮策略模块，图片解码功能
低资源平台	(512KB，1MB]	布局算法模块，Truetype 字库解码模块，AGGE 矢量画布模块
低资源平台	(128KB，512KB]	标准的 UNICODE 换行算法，剪切板算法，加载文件系统资源模块
低资源平台	(10KB，128KB]	除位图字体功能，位图贴图功能以及部分基本控件之外的所有功能

10. 裁剪功能案例

AWTK 全功能开启时，如果编译选项为 -Os （不包含调试信息），此时 ROM 占用大概为 2MB。需要注意的是，因为项目资源（字体、图片、UI 文件等）占用的 ROM 是根据实际情况而定的，所以本章节讨论的裁剪只针对代码文件，不包含资源文件。

在 ROM 较少的板子上，通常要根据实际需求裁剪掉 AWTK 的部分功能，降低 ROM 占用，但这势必会造成功能缺失，并且某些功能被裁减后也会降低性能或者绘图效果，功能模块介绍可查看本文附录二。

10.1 中资源平台的裁剪案例

在中资源平台上，除了输入法需要裁剪以外，其他功能基本都可以使用。输入法本身的代码并不多，但是输入法资源（比如输入法字典、联想字库等）却很大，例如 AWTK 中的谷歌输入法的资源约占用 1MB 多。

本章将 AWTK 针对 STM32F492igtx 平台的 移植层[13] 作为中资源平台上的裁剪案例，该平台的资源如下：

- 片内 ROM：1MB；
- 片内 RAM：512KB；
- 片外 RAM：32MB。

10.1.1 awtk_config.h 的宏

下面是 STM32F492igtx 移植层中的宏配置，此处定义了 **WITH_IME_NULL** 和 **WITHOUT_SUGGEST_WORDS**，它们的作用分别是不启用输入法但是支持使用软键盘，以及不使用联想词组。

```
/* awtk_port/awtk_config.h */
/* 嵌入式系统有自己的main函数时，请定义本宏 */
#define USE_GUI_MAIN 1

/* 如果支持png/jpeg图片，请定义本宏 */
#define WITH_STB_IMAGE 1

/* 如果支持Truetype字体，请定义本宏 */
#define WITH_STB_FONT 1

/* 如果定义本宏，使用标准的UNICODE换行算法，除非资源极为有限，请定义本宏 */
#define WITH_UNICODE_BREAK 1

/* 如果定义本宏，将图片解码成BGRA8888格式，否则解码成RGBA8888的格式 */
#define WITH_BITMAP_BGRA 1

/* 如果定义本宏，将不透明的PNG图片解码成BGR565格式，建议定义 */
#define WITH_BITMAP_BGR565 1
```

[13] https://github.com/zlgopen/awtk-stm32f429igtx-raw

```
/* 如果ROM空间较小，不足以放大字体文件时，请定义本宏 */
#define WITH_MINI_FONT 1

/* 如果启用STM32 G2D硬件加速，请定义本宏 */
#define WITH_STM32_G2D 1

/* 如果启用VGCANVAS，而且没有OpenGL硬件加速，请定义本宏 */
#define WITH_NANOVG_AGGE 1

/* 如果启用VGCANVAS，请定义本宏 */
#define WITH_VGCANVAS 1

/* 如果启用输入法，但不想启用联想功能，请定义本宏 */
#define WITHOUT_SUGGEST_WORDS 1

/* 如果不启用输入法，但支持软键盘，请定义本宏 */
#define WITH_IME_NULL 1
```

10.1.2 需添加到工程中的代码

在裁剪时候需要注意一些问题，如果开启了某些功能后，需要把相关的代码添加在项目中。

1. STM32F492igtx 移植层中定义了宏 **WITH_VGCANVAS** 和宏 **WITH_NANOVG_AGGE**，说明支持矢量画布功能，需要添加的文件详见本文附录二中的矢量画布模块。
2. 如果定义了宏 **WITH_NULL_IM**，说明不需要软键盘了，可以把 awtk/src/ext_widgets/keyboard 目录中的代码去掉，并且把 awtk/src/ext_widgets/ext_widgets.c 文件中 keyboard 相关的代码注册代码注释掉。

注: 根据实际裁剪情况，需添加到工程中编译的 AWTK 代码详见下文 8.3 章节。

10.2 低资源平台的裁剪案例

AWTK 不太建议在低资源平台上运行，ROM 过小会限制 AWTK 的功能和控件，而 RAM 过小会导致 AWTK 的效率降低，并且还会导致部分功能无法使用。

本章将 AWTK 针对 STM32F103ze 平台的 移植层[14] 作为低资源平台上的裁剪案例，该平台的资源如下：

- 片内 ROM：1MB；
- 片内 RAM：64KB；

需要注意的是，虽然 ROM 有 1MB，但是 RAM 只有 64KB（这 64KB 还需要包括显存），并且本案例中会把的项目的 ROM 裁剪到 150KB 左右（不包含项目资源）。

[14] https://github.com/zlgopen/awtk-stm32f103ze-raw

10.2.1 awtk_config.h 的宏

由于 103 的项目中 RAM 只有 64KB，所以这里采用了片段式显存的方案，详情可参考 awtk/docs/lcd.md，下面分配了 8 * 1024 * 2 字节作为片段式显存，然后给 AWTK 分配了 4 * 6000 字节作为运行内存。

```
/* 嵌入式系统有自己的main函数时，请定义本宏 */
 #define USE_GUI_MAIN 1

/* 如果需要支持预先解码的位图字体，请定义本宏。一般只在RAM极小时，才启用本宏 */
#define WITH_BITMAP_FONT 1

/* 如果不需输入法，请定义本宏 */
#define WITH_NULL_IM 1

/* 如果出现wcsxxx之类的函数没有定义，请定义该宏 */
#define WITH_WCSXXX 1

/* AWTK 只保留基本功能和控件 */
#define AWTK_LITE 1

/* 如果想裁剪 AWTK 的基本控件，请定义本宏 */
// #define AWTK_NOGUI 1

/* 如果内存不足以提供完整的 FrameBuffer，请定义本宏(单位：像素个数) */
 #define FRAGMENT_FRAME_BUFFER_SIZE 8 * 1024
```

按照上述宏定义，在编译选项为 -O0 时，ROM 占用约为 230KB，如果编译选项改为 -O2 或者 -O3，ROM 会低于 200KB。

10.2.2 裁剪多余的部分

AWTK_LITE 宏主要实现了下列功能的裁剪：

定义的宏	作用
WITH_NULL_IM	裁剪了输入法和软键盘
WITHOUT_LAYOUT	裁剪了布局算法
WITHOUT_CLIPBOARD	裁剪了剪切板模块
WITHOUT_EXT_WIDGETS	裁剪了扩展控件
WITHOUT_INPUT_METHOD	裁剪输入法全局对象
WITHOUT_WINDOW_ANIMATORS	裁剪了窗口动画
WITHOUT_WIDGET_ANIMATORS	裁剪了控件动画
WITHOUT_DIALOG_HIGHLIGHTER	裁剪对话框高亮模块

接下来，移除多余的部分可以进一步降低 ROM 占用，在最新的 AWTK 中可以不需要做下面的操作，因为宏 **AWTK_LITE** 宏已经做了。

首先，可以裁剪掉 fscript 模块，该模块是一个简单的脚本引擎，通常给 awtk-mvvm[15]

[15] http://github/zlgopen/awtk-mvvm

使用，普通的 AWTK 程序并不需要，裁剪步骤如下：

1. 在 STM32F103ze 移植层中，移除 fscript.c 文件。
2. 把 awtk_global.c 文件中关于 fscript 的代码删除。
3. 修改 widget.c 文件中的 widget_exec_code 函数，代码如下：

```
/* awtk/src/base/widget.c */
static ret_t widget_exec_code(void* ctx, event_t* evt) {
  return RET_OK;
}
```

重新编译后，ROM 大小会降到 210KB 左右。

接下来，如果项目没有用到圆角矩形效果，还可以裁剪掉该功能，修改 canvas.c 文件，代码如下：

```
/* awtk/src/base/canvas.c */

/* 1.注释掉该文件 */
// #include "ffr_draw_rounded_rect.inc"

/* 2. 修改以下相关函数，去掉实现代码 */
ret_t canvas_fill_rounded_rect(/*...*/) { /* 此处表示省略参数，下文类似 */
  return RET_FAIL;
}

ret_t canvas_stroke_rounded_rect(/*...*/) {
  return RET_FAIL;
}

ret_t canvas_fill_rounded_rect_ex(/*...*/) {
  return RET_FAIL;
}

ret_t canvas_stroke_rounded_rect_ex(/*...*/) {
  return RET_FAIL;
}
```

重新编译后，ROM 大小会降到 200KB 左右，并且占用内存也减少了 3KB。

最后，我们继续裁剪离线画布功能，这个功能常用于窗口动画以及控件截图，在内存紧缺时用处不大，裁剪该功能首先需要注释掉 canvas.c 文件中的以下代码：

```
/* awtk/src/base/canvas.c */
// #include "canvas_offline.inc"
```

然后，修改 window_manager_default.c 文件中两个函数，代码如下：

```
/* awtk/src/window_manager/window_manager_default.c  */
ret_t window_manager_default_snap_curr_window(/*...*/) {
  return RET_NOT_IMPL;
}
```

```
ret_t window_manager_default_snap_prev_window(/*...*/) {
  return RET_NOT_IMPL;
}
```

重新编译后，ROM 大小已降为 190KB，接着把编译选项改成 -O2，会降至 150KB，如果定义宏 **AWTK_NOGUI**，还可以进一步降到 130KB，但这会裁剪掉基本控件，导致 UI 文件无法正常显示。

10.3 要添加到工程的文件

AWTK 在嵌入式下的核心代码结构与需要添加到工程中的代码文件详见本文 1.4 章节中的表格。

第四部分

附录篇

11. 附录一：文件系统移植相关接口

文件操作相关接口（fs_file_t）说明详见下表：

接口	说明	备注
fs_file_read	读取文件	
fs_file_write	写入文件	
fs_file_printf	将格式化字符串写入文件	某些平台可能不支持
fs_file_seek	定位读写指针到指定的位置	
fs_file_truncate	清除文件内容	
fs_file_eof	判断文件是否结束	
fs_file_tell	获取文件当前读写位置	
fs_file_size	获取文件大小	
fs_file_sync	同步文件到磁盘	
fs_file_stat	获取文件信息	如果使用 file_browser 控件，则必须实现
fs_file_close	关闭文件	

文件夹操作相关接口（fs_dir_t）说明详见下表：

接口	说明	备注
fs_dir_rewind	重置文件夹读取位置到开始	某些平台可能不支持
fs_dir_read	读取文件夹对象	
fs_dir_close	关闭文件夹对象	

文件系统操作相关接口（fs_t）说明详见下表：

接口	说明	备注
fs_open_file	打开文件	mode 取值请参考 fopen 函数
fs_remove_file	删除文件	
fs_file_exist	判断文件是否存在	
fs_file_rename	文件重命名	
fs_open_dir	打开目录	
fs_create_dir	创建目录	
fs_change_dir	修改当前目录	
fs_remove_dir	删除目录	
fs_dir_exist	判断目录是否存在	
fs_dir_rename	目录重命名	
fs_get_file_size	获取文件大小	
fs_get_disk_info	获取文件系统信息	
fs_stat	获取文件信息	使用 file_browser 控件时必须实现
fs_get_cwd	获取当前所在目录	某些平台可能不支持
fs_get_exe	获取可执行文件所在目录	某些平台可能不支持
fs_get_user_storage_path	获取 home 目录或者应用程序可以写入数据的目录	某些平台可能不支持
fs_get_temp_path	获取临时目录	某些平台可能不支持

12. 附录二：AWTK 可裁剪的功能模块

12.1 窗口动画模块

AWTK 提供窗口动画功能，在窗口打开或关闭时，可以引入一个过渡动画，让用户感觉这个过程是流畅的。其基本原理很简单：在打开或关闭窗口时，把前后两个窗口预先绘制到两张内存图片上，按照指定规则显示两张图片，形成动画效果。

AWTK 默认启动窗口动画模块，其占用资源详见本文第九章，对于资源紧缺的低端平台，如果不使用窗口动画，编译时定义下面的宏即可：

```
#define WITHOUT_WINDOW_ANIMATORS 1
```

注: 以上宏定义以及本文中介绍的 AWTK 功能模块相关的宏定义如果没有特别说明，一般情况下，在 PC 上修改 awtk_config.py 文件，在嵌入式系统上修改 awtk_config.h 文件。

此外，启用窗口动画时，默认情况下，需要对前后两个窗口进行截图，以获得更好的性能，但是这需要两个 Framebuffer 大小的内存，如果内存不够，可以选择关闭截图缓存，直接进行绘制。编译时定义下面的宏即可：

```
/**
 * 通常在以下情况，关闭窗口动画截图缓存：
 *  1. 窗口界面简单；
 *  2. 内存资源紧缺，但 CPU 速度较快。
 *
 * 限制条件：
 *  1.不支持缩放窗口动画。
 *  2.不支持对话框高亮策略。
 *
 * 备注：如果绘制速度慢，而且内存紧缺，建议关闭窗口动画。
 */
#define WITHOUT_WINDOW_ANIMATOR_CACHE 1
```

注: AWTK 1.6.1 不支持上述宏定义，如需关闭窗口动画缓存，请使用 1.6.1 版本以上的 AWTK。

12.2 控件动画模块

AWTK 提供控件动画功能，常用于入场动画、离场动画、装饰用户界面和吸引用户注意力等，其主要特色和使用方法详见：awtk/docs/widget_animator.md。

AWTK 默认启动控件动画模块，其占用资源详见本文第九章，对于资源紧缺的低端平台，如果不使用控件动画，编译时定义下面的宏即可：

```
#define WITHOUT_WIDGET_ANIMATORS 1
```

注: 控件动画中的缩放与旋转动画需要矢量画布 vgcanvas 的支持，其介绍详见下文矢量图画布模块。

12.3 图片解码模块

AWTK 采用第三方的 STB 类库实现 jpg、png 和 gif 格式图片资源的软解码，其占用资源详见本文第九章。

需要注意的是 STB 类库在解码 jpg 图片时，所需内存约为三个位图大小 + 18456B，而解码 png 图片时，所需内存约为两个位图大小，以上皆为解码时的内存峰值，解码完成后会这些内存会被释放，若 STB 在解码时申请不到这么大的内存，将解码失败无法显示图片，但不影响程序正常运行。

如果希望启动图片解码模块，即支持 jpg/png/gif 图片，编译时定义下面的宏即可：

```
#define WITH_STB_IMAGE 1
```

如果希望将图片解码成 BGRA8888 格式的位图，请定义下面的宏，否则将解码成 RGBA8888 格式的位图：

```
#define WITH_BITMAP_BGRA 1
```

如果希望将不透明的 PNG 图片解码成 BGR565 格式的位图，请定义下面的宏：

```
#define WITH_BITMAP_BGR565 1
```

如果希望将不透明的 PNG 图片解码成 RGB565 格式的位图，请定义下面的宏：

```
#define WITH_BITMAP_RGB565 1
```

注: 解码出来的位图格式与 LCD 的格式保存一致，可以大幅度提高性能。

此外，AWTK 的图片解码模块为提高效率，存在缓存机制，通常在第一次显示图片时将其解码成位图并缓存到图片管理器（image_manager）中，下一次再显示这张图片时就直接到缓存拿就行了，当内存不足时 AWTK 会自动清除长时间不用的图片缓存，用户也可以调用 image_manager_unload_unused 接口清除指定时间内没有使用的图片缓存。

注:

1. AWTK 中用于图片解码的 STB 类库详见：awtk/3rd/stb/stb_image.h。
2. AWTK 中的图片解码器详见：awtk/src/image_loader_stb.h。

12.4 字体解码模块

AWTK 支持矢量（TrueType）字体解码，可以将矢量字体解码成位图显示到界面上，目前提供 STB 类库和 FreeType 类库两种实现的字体解码器。FreeType 类库虽然功能强大，但与 STB 类库相比，项目代码量大，占用较多 ROM 资源，因此 AWTK 通常使用 STB 类库解码矢量字体，编译时定义下面的宏即可，STB 字体解码器占用资源详见本文第九章。

```
#define WITH_STB_FONT 1
```

如果仍然希望使用 FreeType 类库解析矢量字体，则可以定义下面的宏：

```
#define WITH_FT_FONT 1
```

注: 如果以上两个宏同时定义，则优先使用 STB 类库。

此外，AWTK 的字体解码模块为提高效率，存在缓存机制，通常在第一次显示字体时将其解码成字模并缓存到 glyph_cache_t 中，下一次再显示这个字体时就直接到缓存拿就行了，当缓存的字模个数达到上限或内存不足时 AWTK 会自动清除最长时间不用的字模缓存，用户也可以调用 font_manager_shrink_cache 接口清除字模缓存。

AWTK 在嵌入式平台上默认字模缓存的最大个数为 256 个，用户可以通过定义宏 TK_GLYPH_CACHE_NR 来设置字模缓存的容量：

```
#ifndef TK_GLYPH_CACHE_NR
#ifdef WITH_SDL
#define TK_GLYPH_CACHE_NR 4096
#else
#define TK_GLYPH_CACHE_NR 256   /* 设置最多缓存 256 个字模 */
#endif /*WITH_SDL*/
```

注:

1. STB 类库实现的字体解码器详见：awtk/src/font_loader/font_loader_stb.h。
2. TrueType 类库实现的字体解码器详见：awtk/src/font_loader/font_loader_ft.h。

12.5 输入法模块

AWTK 提供输入法模块功能，它的实现不是特别复杂，但涉及的组件较多，理解起来比较困难，这里简单介绍一下 AWTK 中输入法模块的内部实现框架，详见下图：

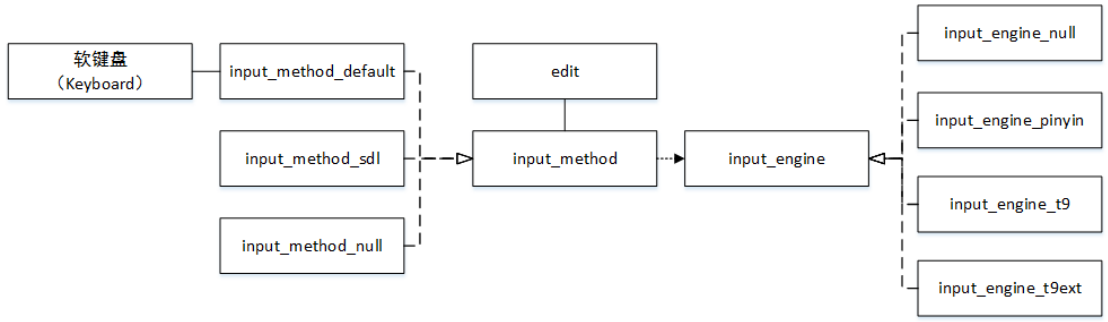

图 12.1 输入法模块实现框架

- input_method_default 提供 AWTK 输入法模块的缺省实现，负责软键盘的开关和输入法引擎的创建，通常用于嵌入式平台。
- input_method_sdl 包装了 SDL 的原生输入法，使用平台自身的输入引擎，通常用于桌面 Linux、MacOS、Windows、Android 和 iOS 等平台。
- input_method_null 提供了空的输入法实现，在不启用输入法时使用。

AWTK 默认启用输入法模块，其占用资源详见本文第九章，对于资源紧缺的低端平台，如果不使用输入法功能，编译时定义下面的宏即可：

```
#define WITH_NULL_IM 1
```

注: 启用 AWTK 输入法模块时，需添加 awtk/src/input_methods/input_method_creator.c 文件。

输入法引擎主要负责将用户的按键转换成一组候选字，这些候选字将在软键盘上的候选字控件上显示出来。输入法引擎有很多，不同的语言也有不同的输入法引擎。目前 AWTK 只支持英文和中文输入，提供多种输入法引擎，具体详见：awtk/src/input_engines/README.md 文档。

注: 如果不启用输入法，可以不用加任何输入法引擎。

在嵌入式平台上，常用的输入法引擎如下：

输入法引擎	适用场景	宏定义	需要添加的源文件
null 输入法引擎	用于启用软键盘，但无需输入法引擎的情况，比如只需软键盘输入英文	WITH_IME_NULL	input_engine_null.c
Goole 拼音输入引擎	用于需要中文拼音输入的情况，要求 ROM 不小于 4M	WITH_IME_PINYIN	input_engine_pinyin.c
带触摸屏的 T9 输入法引擎	T9 输入法又称九宫格输入法，常用于 Android 和 iOS 等平台	WITH_IME_T9	input_engine_t9.c
外接键盘 T9 输入法引擎	需要使用 T9 输入法，但又没有触摸屏时，使用外部硬键盘，请使用本引擎	WITH_IME_T9EXT	input_engine_t9ext.c

注:

1. 以上输入法引擎不能同时使用，其所需源文件均放在 awtk/src/input_engines 目录。
2. 如果希望使用 Goole 拼音输入引擎加入中文输入法，还需要加入 gpinyin 类库，即 awtk/3rd/gpinyin/src 目录下的源码文件，加入中文输入法的步骤请参考：awtk/docs/chinese_ime.md。
3. 如果使用 Goole 拼音输入引擎，并希望自定义拼音输入法字典和联想字库，请参考：awtk/ docs/how_to_update_gpinyin_data.md。
4. T9 输入法引擎的使用方法详见：awtk/docs/t9_notes.md。

如果启用输入法模块，但不想启用联想功能，请定义本下面的宏：

```
#define WITHOUT_SUGGEST_WORDS 1
```

注: 联想功能是指输入某个汉字或词组后，输入法会根据该汉字或词组提供其常用的组词，例如输入汉字“我”后，输入法会根据”我们”、“我国”、“我省”等常用组词，提供”们”、“国”、”省”等汉字。

此外，AWTK 的输入法引擎默认候选字个数为 255 个，即候选字缓冲区大小为 255 字节，用户可以通过定义下面的宏来设置候选字缓冲区的大小：

```
#define TK_IM_DEFAULT_MAX_CANDIDATE_CHARS 255

#ifndef TK_IM_MAX_CANDIDATE_CHARS
#define TK_IM_MAX_CANDIDATE_CHARS TK_IM_DEFAULT_MAX_CANDIDATE_CHARS
#endif
```

12.6 矢量图画布模块

AWTK 支持矢量图画布，主要用于绘制矢量图形（比如图片的旋转与缩放，控件圆角等，但是填充圆角矩形和边框为 1 的圆角矩形不受影响），其抽象基类接口详见：awtk/src/base/vgcanvas.h，目前 AWTK 提供了基于 nanovg 的实现，由于原生 nanovg 只支持 OpenGL 硬件渲染，因此我们对它进行了一些改进，使用 agg/agge 作为 nanovg 图形渲染后端，实现软件渲染；：

注: nanovg 是第三方开源的矢量图形库（Vector graphics library），GitHub 地址：https://github.com/memononen/nanovg。

经过改进后，AWTK 的矢量图画布模块共有以下四类实现：

矢量图画布的实现	说明	适用场景	后端实现
vgcanvas_nanovg_soft	纯软件实现	适合大多数嵌入式平台	其后端有 agg 和 agge 两种实现
vgcanvas_nanovg_gl	OpenGL 实现	适合有 GPU 且支持 OpenGL 的平台	其后端有 OpenGL3、OpenGLES2 和 OpenGLES3 三种实现
vgcanvas_cairo	cairo 实现	cairo 也是软件矢量库，速度快功能全，但代码体积大，请酌情使用	
vgcanvas_null	空实现	适合资源极其有限的低端平台，不支持矢量图画布模块时使用本实现	

注: 以上矢量图画布的实现源码均放在 awtk/src/vgcanvas 目录。

AWTK 中控制矢量图画布模块的宏较多，此处统一列出，具体用法详见注释。

需要注意的是，在嵌入式平台上，通常采用 agge 软件渲染模式支持矢量画布，此时只需定义宏 WITH_NANOVG_AGGE 即可，AWTK 会自行定义宏 WITH_VGCANVAS、WITH_NANOVG 和 WITH_NANOVG_SOFT。

注: 采用 agge 软件渲染模式支持矢量画布会占用约 30 KB 内存和 20 KB 的 ROM，具体详见本文第九章

```
/**
 * 如果启用矢量图画布，请定义本宏
 */
#define WITH_VGCANVAS 1

/**
 * 如果希望使用基于 nanovg 实现的矢量图画布，请定义本宏
 */
#define WITH_NANOVG 1

/**
 * 在有 GPU 时，如果希望使用硬件 OpenGL 渲染矢量画布，请定义本宏
```

```
 * > 备注：定义本宏时，AWTK 会自动定义宏 WITH_VGCANVAS 和 WITH_NANOVG。
 */
#define WITH_NANOVG_GPU 1

/**
 * 在没有 GPU 时，如果希望使用软件渲染矢量画布，请定义本宏
 * > 备注：定义本宏时，AWTK 会自动定义宏 WITH_VGCANVAS 和 WITH_NANOVG。
 */
#define WITH_NANOVG_SOFT 1

/**
 * 如果希望使用软件渲染矢量画布，且启用 agge 作为 nanovg 的后端，请定义本宏
 * > 备注：agge 相较于 agg，代码量小，速度快，但图像质量稍差。
 * > 备注：定义本宏时，AWTK 会自动定义宏 WITH_VGCANVAS、WITH_NANOVG 和 WITH_NANOVG_
↪SOFT。
 */
#define WITH_NANOVG_AGGE 1

/**
 * 如果希望使用软件渲染矢量画布，且启用 agg 作为 nanovg 的后端，请定义本宏
 * > 备注：agg 相较于 agge，代码量大，速度慢，但图像质量较好，且 agg 是以 GPL␣
↪协议开源，
 如果在商业软件中使用，需要与原作者协商：http://www.antigrain.com。
 * > 备注：定义本宏时，AWTK 会自动定义宏 WITH_VGCANVAS、WITH_NANOVG 和 WITH_NANOVG_
↪SOFT。
 */
#define WITH_NANOVG_AGG 1
```

此外，在嵌入式平台上定义宏 WITH_NANOVG_AGGE，支持矢量图画布时，需要加到工程中的文件详见下表：

目录	说明
awtk/src/svg	支持矢量图画布时全部加入
awtk/src/vgcanvas	使用软件渲染支持矢量画布请加入 vgcanvas_nanovg_soft.c，不支持矢量画布则加入 vgcanvas_null.c
awtk/3rd/agge	启用 agge 作为 nanovg 的后端实现软件渲染时加入
awtk/ 3rd/ nanovg/ base	支持矢量图画布时全部加入

12.7 对话框高亮模块

AWTK 支持对话框高亮策略，即打开对话框时，将背景窗口变暗或变模糊，以突出当前对话框的重要性，其占用资源详见本文第九章。AWTK 把对话框高亮策略抽象成接口（awtk/src/base/dialog_highlighter），用户可以自己实现特殊效果的高亮策略，也可以使用缺省的高亮策略。

注: AWTK 对话框高亮策略的使用方法和自定义高亮策略的步骤请参考: awtk/docs/dialog_highlight.md。

缺省的对话框高亮策略是让背景窗口变暗，其原理与窗口动画类似，打开对话框时，把

前后两个窗口预先绘制到两张内存图片上（会在背景上画一层半透明的蒙版来突出对话框），按照指定规则显示两张图片，形成动画效果。它可以是静态的，即透明度（alpha）固定不变，也可以是动态的，即透明度（alpha）在打开对话框的过程中是不断变化的。

动态的高亮策略在对话框打开的动画过程中，背景窗口逐渐变暗，有更好的视觉效果，但也需要更多的计算开销。对于计算性能不高的低端平台建议使用静态的高亮策略，即将 start_alpha 和 end_alpha 属性设为相同值。

对于低端平台，如果不使用对话框高亮策略，请定义下面的宏：

```
#define WITHOUT_DIALOG_HIGHLIGHTER 1
```

12.8 剪切板模块

AWTK 提供了剪切板功能，抽象接口详见：awtk/src/clip_board.h，目前有以下两种实现：

- clip_board_sdl 包装了 SDL 的原生剪切板，使用平台自身的剪切板功能，通常用于桌面 Linux、MacOS、Windows、Android 和 iOS 等平台。
- clip_board_default 提供 AWTK 剪切板的缺省实现，通常用于嵌入式平台。

注: 以上剪切板的实现文件均放在 awtk/src/clip_board 目录。

AWTK 默认启用剪切板，其占用资源详见本文第九章，对于低端平台，如果不使用剪切板功能，请定义下面的宏：

```
#define WITHOUT_CLIPBOARD 1
```

12.9 布局模块

如果界面上的控件是预先知道的，而且窗口大小固定不变，那么此时创建界面非常简单，直接设置控件的位置和大小即可。但在以下情况中，使用布局参数是更好的选择：

- 窗口的大小是可以动态调整的；
- 需要适应不同分辨率的 LCD；
- 界面上的控件是动态生成的，需要在程序运行中创建界面；

AWTK 提供了控件布局管理器（layouter），其中包含对控件自身进行布局的 self_layouter 以及对子控件进行布局的 children_layouter，它们的介绍和详细用法请参考：awtk/docs/layout.md。

注: 除了 AWTK 默认支持的布局参数（比如百分比、位置居左/中/右、浮动布局、网格布局等）之外，AWTK 还提供了灵活的扩展机制，self_layouter 和 children_layouter 都是抽象接口，用户可以自行实现新的布局方式。

AWTK 默认启用布局模块，其占用资源详见本文第九章，如果不使用布局管理器的功能，请定义下面的宏：

```
#define WITHOUT_LAYOUT 1
```

12.10 通用文件系统模块

AWTK 提供了通用文件系统的抽象接口（awtk/src/tkc/fs.h），由于该功能与平台系统相关，目前 AWTK 仅内置了 Linux、MacOS、Windows、Android 和 iOS 平台上的实现，代码详见：awtk/src/platforms/pc/fs_os.c，如果需要在其他嵌入式平台上使用文件系统的功能请参考本文第六章进行移植。

在 AWTK 中，文件系统模块的功能主要用于支持文件系统时，从文件系统中加载项目资源（比如 UI 界面、图片、字库等），资源目录结构和加载方式详见 AWTK 开发实践。

如果支持从文件系统加载资源，请定义下面的宏，AWTK 将使用项目 res/assets/default/raw 目录下的资源，其中 default 是主题名称：

```
#define WITH_FS_RES 1
```

AWTK 也支持自定义资源根目录，在某些嵌入式平台上，由于 ROM 有限，项目资源又比较多，通常会将代码（程序/固件）放在 ROM 中，而资源放在文件系统中，比如外接 SD 卡，此时可以定义下面的宏指定资源所在的路径：

```
#define APP_RES_ROOT "0://AwtkApplication/res/"
```

12.11 标准的 UNICODE 换行模块

AWTK 提供了标准的 UNICODE 换行算法，用于处理字符串的换行情况（比如遇到“\r\n”、“\r”、”\n”等符号换行、英文单词不允许换行等情况），其需要约 50 KB 的 ROM 和 1.5 KB 的 RAM。

在嵌入式平台上，除非资源极为有限，否则建议定义下面的宏，启用 UNICODE 换行模块：

```
#define WITH_UNICODE_BREAK 1
```

12.12 半透窗口机制模块

AWTK 支持半透窗口机制，即支持窗口背景为透明色的刷新机制。在某些平台下，支持多个 LCD 的硬件图层（Framebuffer），这些图层可以在硬件绘制时做融合显示到 LCD 上，比软件合成要快得多，既然是要做图层融合，那么通常都会存在窗口背景为透明的情况，此时可以定义下面的宏，启用 AWTK 半透窗口机制：

注: AWTK 的半透窗口机制只适用于多个硬件图层融合的情况，比如一个硬件图层刷新 GUI，另一个硬件图层播放视频，二者融合在一起显示整个界面。

```
#define WITH_LCD_CLEAR_ALPHA 1
```

需要注意的是，启用半透窗口机制要求 AWTK 的 LCD 类型必须为 32 位色（RGBA 或 BGRA），并且会增加窗口绘制时的计算量（做透明色混合运算），降低 AWTK 的性能，用户请根据实际情况决定是否启用。

12.13 AWTK 内存管理模块

AWTK 提供了内存管理器，其接口详见：awtk/ src/ tkc/ mem.h，它基于内存分配器（mem_alloactor_t）实现，目前 AWTK 提供的内存分配器详见下图：

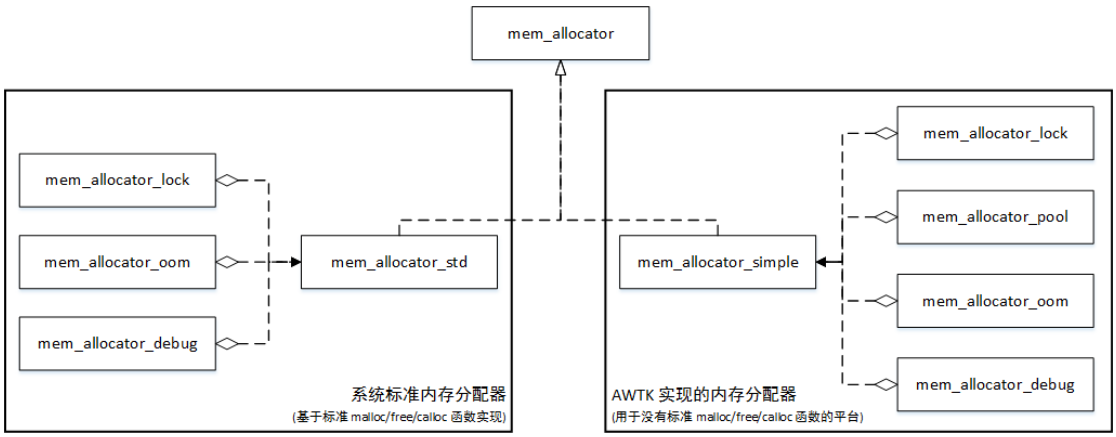

图 12.2 AWTK 内存分配器

注: AWTK 初始化时，会调用 tk_mem_init 函数初始化内存管理器（给内存管理器分配一大块内存），具体流程详见本文第一章。

以上内存分配器有两个基础实现，它们的介绍和适用场景如下：

1、mem_alloactor_std 是基于标准的 malloc/free/calloc 等函数实现的内存分配器。在系统提供了 malloc/free/realloc 函数时，请定义下面的宏，使用本分配器：

```
#define HAS_STD_MALLOC 1
```

2、mem_alloactor_simple 是 AWTK 本身实现的一个简单的内存分配器，通常在系统没有提供标准的 malloc/free/calloc 等函数时使用，即没有定义宏 HAS_STD_MALLOC。

其中，使用 mem_alloactor_simple 作为内存分配器时，AWTK 基于装饰器设计模式对其进行了以下四层包装：

3、mem_alloactor_lock 对现有的 allocator 进行包装，提供互斥功能。由于标准内存分配器使用的 malloc/ free/ realloc 函数本身有互斥功能，所以本装饰器主要用于 mem_allocator_simple。

4、mem_alloactor_pool 对现有的 allocator 进行包装，预先分配一部分内存作为内存池，将其分割成多个小内存块，用于分配小块内存，可以避免内存碎片，内存池占用的资源和分割情况详见本文第九章，需要注意的是，在没有定义宏 HAS_STD_MALLOC 且 tk_mem_init 函数设置的内存大小大于 32KB 时才会分配内存池。

注: 经测试，通常在程序运行中 80% 的情况都是小块内存，在内存池中分配，可以有效的防止内存碎片的产生，并且用户可以跟踪 mem_allocator_pool 的使用情况，根据实际情况调整预先分配的块数（详见

mem_allocator_pool_init 函数）。

5、mem_alloactor_debug 对现有的 allocator 进行包装，记录分配的内存，用于帮助分析内存的使用和泄露，在定义下面的宏后启动该功能，在程序中可以调用 tk_mem_dump 函数打印内存使用情况。

```
#define ENABLE_MEM_LEAK_CHECK 1
```

注：由于 mem_allocator_debug 本身会占用一部分内存，一般在只 PC 上使用。

6、mem_alloactor_oom 对现有的 allocator 进行包装，如果分配内存失败，调用预先设置的回调函数释放内存，然后再重试，AWTK 默认启用该功能，内存耗尽处理流程详见：awtk/docs/out_of_memory.md。

注：由于 AWTK 内置的内存耗尽处理使用了信号量，因此在多线程场景下调用调用内存管理器的接口，需要参考本文第六章内容移植互斥锁和信号量。

此时，调用 TKMEM_ALLOC 函数分配内存时的流程详见下图：

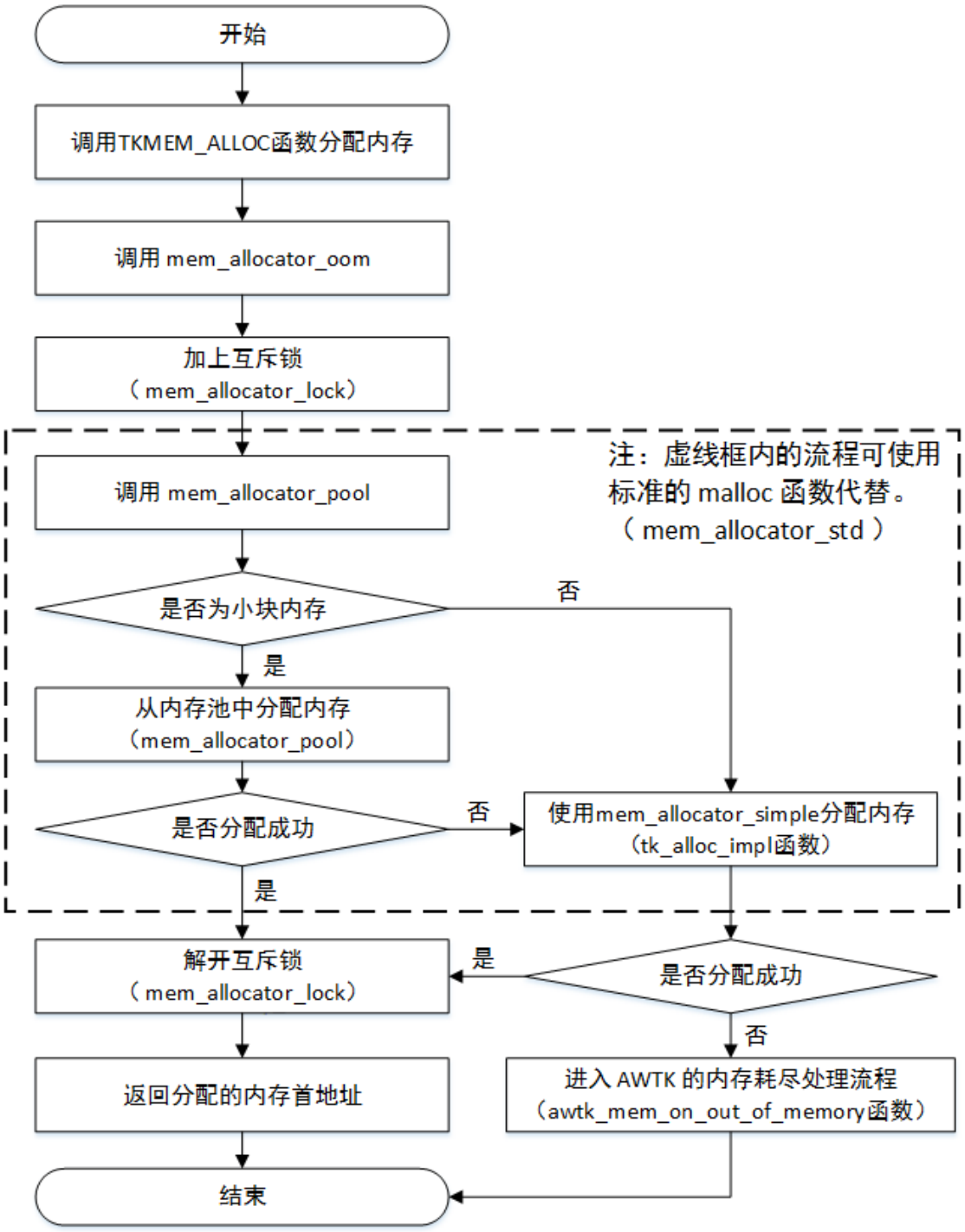

图 12.3　内存管理器流程图

注：

1. 该流程图主要描述使用 mem_alloactor_simple 时的流程，如果使用的是 mem_alloactor_std，仅在 malloc 时有差别。
2. 如果使用标准的内存分配器 mem_allocator_std，那么上图虚线框内的流程即调用标准的 malloc 函数。
3. AWTK 的内存耗尽处理流程请详见：awtk/docs/out_of_memory.md。

此外，为提高绘制性能，AWTK 默认使用内置 tk_memcpy16 函数和 tk_memcpy32 函数拷贝图像，如果平台有提供优化版本的 memcpy 函数，可以定义下面的宏，直接使用 memcpy

函数拷贝图像：

```
#define HAS_FAST_MEMCPY 1
```

在嵌入式系统中，可能有多块不连续的内存，AWTK 也支持多块不连续的内存块，请定义下面的宏指定内存块的数目，并使用 tk_mem_init_ex 函数代替 tk_mem_init 函数初始化内存，示例用法请参考：awtk/docs/how_to_support_multi_mem_block.md。

```
#define TK_MAX_MEM_BLOCK_NR 4
```

12.14 G2D 硬件加速模块

AWTK 支持 G2D 硬件加速，主要用于提高图像绘制的性能，其接口详见：awtk/src/base/g2d.h，只需要该文件中的相关接口，并定义下面的宏，即可启用 G2D 硬件加速：

```
#define WITH_G2D 1
```

目前，AWTK 内置了 STM32 系列平台 G2D 硬件加速的实现，代码详见：awtk/src/blend/stm32_g2d.c，只需定义下面的宏即可启用该功能：

```
#define WITH_STM32_G2D 1
```

此外，在 AWTK 针对 AWorks (RT1052) 平台的 移植层[16] 中，提供了 NXP PXP 硬件加速的实现文件，代码详见：awtk-aworks-rt1052/awtk-port/rt1052_g2d.c，只需定义下面的宏即可启用该功能：

```
#define WITH_PXP_G2D 1
```

注：定义宏 WITH_STM32_G2D 或宏 WITH_PXP_G2D 后，AWTK 会自动定义宏 WITH_G2D 启用硬件加速，用户无需重复定义。

12.15 控件模块

AWTK 提供了丰富的内置控件，用户可以通过组合这些控件快速开发出复杂的 GUI 界面，这些控件分为两类：默认控件和扩展控件，它们的介绍详见下文。

12.15.1 默认控件模块

AWTK 的默认控件都放在 awtk/src/widgets 目录下，其中包括按钮（button）、编辑器（edit）、静态文本（label）等常用控件。AWTK 初始化时，会调用 tk_widgets_init 函数注册这些默认控件（将这些控件的构造函数添加到控件工厂管理器中）。

需要注意的是，启用 AWTK 的极简模式，即定义宏 AWTK_LITE 后，会将部分默认控件裁剪掉，被裁剪的控件较多，此处不逐一列出，具体详见 awtk/src/widgets/widgets.c 文件中的 tk_widgets_init 函数。

```
/**
 * 如果启用 AWTK 的极简模式，请定义本宏
 *
 * > 备注：启用该模式后，有以下限制：
```

[16] https://github.com/zlgopen/awtk-aworks-rt1052

```
 *  1.不支持输入法。
 *  2.不支持控件布局。
 *  3.不支持剪切板。
 *  4.不支持扩展控件。
 *  5.不支持圆角矩形。
 *  6.不支持窗口动画。
 *  7.不支持控件动画。
 *  8.不支持对话框高亮策略。
 *  9.不支持标准的 UNICODE 换行。
 *  10.不支持 fscript 脚本引擎。
 *  11.不支持图片缩放、旋转、平铺和九宫格显示功能。
 *  12.不支持离线画布。
 */
#define AWTK_LITE 1
```

12.15.2 扩展控件模块

AWTK 的扩展控件都放在 awtk/src/ext_widgets 目录下，其中包括多行编辑器（mledit）、表盘（gauge）、环形进度条（progress_circle）等复杂控件。AWTK 初始化时，会调用 tk_ext_widgets_init 函数注册这些扩展控件，具体流程详见本文第一章。

由于扩展控件通常比较复杂，所以使用时请注意以下三点：

1. 部分扩展控件不适合在低端平台使用。
2. 部分扩展控件需要矢量画布 vgcanvas 的支持。
3. 部分扩展控件需要与控件动画效果配合使用。

在低端平台上，除了启用默认控件模块中提到的极简模式时，会裁剪掉扩展控件，AWTK 还另外提供了下面的宏来控制是否支持扩展控件，如果不使用扩展控件，请定义下面的宏：

```
#define WITHOUT_EXT_WIDGETS 1
```

此外，针对扩展控件在低端平台的使用情况，我们做了以下测试：

- 测试平台：STM32f103ze。
- RAM：64 KB。
- ROM：512 KB

由于 STM32f103ze 平台的内存不足以提供一屏的 LCD 的 Framebuffer，此处采用**片段式 Framebuffer 方案**来测试（该方案不支持矢量画布 vgcanvas）并且开启控件动画效果，由于如果控件刷新面积很大，会导致刷新次数变多和刷新效率减低，影响实际的效果，所以该测试不考虑实际效果，只确定扩展控件是否能正常工作，测试结果详见下表。

注：由于在低端平台上使用扩展控件需要注意比较多的问题，建议先了解该扩展控件的代码原理才使用，否则容易出现奇怪的问题，所以低端平台慎用扩展控件。

扩展控件名称	是否正常工作	注意事项
canvas_widget	是	需要注意平台是否支持 vgcanvas，否则调用 vgcanvas 对应的接口会失效。

续上表

扩展控件名称	是否正常工作	注意事项
color_picker	是	需要注意平台的内存是否足够，color_component 控件需要开辟一块和该控件宽高一样的 32 位色的图片内存。
combo_box_ex	是	
features	是	
file_browser	否	需要文件系统支持。
gif_image	否	因为 imagegen 工具暂时不支持生成 gif 格式的资源。
gauge	否	因为需要 vgcanvas 支持。
image_animation	是	
image_value	是	
keyboard	是	需要注意内存是否足够，因为软键盘会创建大量的 button 控件。
mledit	是	
mutable_image	是	但不支持旋转和缩放，因为它们需要 vgcanvas 支持。
progress_circle	否	因为需要 vgcanvas 支持。
rich_text	是	但是需要 widget 动画效果配合。
scroll_label	是	但是需要 widget 动画效果配合。
scroll_view	是	但是需要 widget 动画效果配合。
slide_menu	是	但是需要 widget 动画效果配合。
slide_view	是	但是需要 widget 动画效果配合，slide_indicator 控件只使用 stroke_rect 和 fill_rect 效果，不能使用画圆形的效果，因为画圆需要 vgcanvas 支持。
svg_image	否	因为需要 vgcanvas 支持。
switch	是	但是没有圆角的效果，因为圆角效果需要 vgcanvas 支持。
text_selector	是	但是需要 widget 动画效果配合。
time_clock	否	因为需要 vgcanvas 支持。

注：在低端平台中，通常会禁用控件动画和扩展控件，所以需要手动把取消宏 WITHOUT_WIDGET_ANIMATORS 和宏 WITHOUT_EXT_WIDGETS 宏定义才能正常使用。

13. 附录三：AWTK 核心模块内存占用表

功能模块	内存占用	备注
小对象内存池	11KB 到 110KB	运行内存小于 100KB 时，内存池占用约 11 KB；运行内存小于 1000KB 时，内存池占用约 50 KB；其余情况内存池占用约为 110 KB。
系统信息	75B	68B 为 system_info 对象，7B 为默认字体的名称字符串
定时器管理器	32B	32B 为定时器管理器对象，每个定时器占用 104B，定时器结束后释放
idle 管理器	20B	20B 为 idle 管理器对象，每个 idle 占用 72B，idle 结束后释放
默认主题对象	20B	20B 为主题对象，主题数据属于项目资源，另外计算
资源管理器	192B	72B 为资源管理器对象，其中会开辟大小为 30 的动态数组（120B），如果资源个数超过 30 个，内存将增加。
本地化信息管理器	36B	20B 为 locale_info 对象，16B 为事件分发器。
字库管理器	100B	28B 为 font_manager 对象，8B 为长度为 2 的字库动态数组，64B 为注册的释放字库资源的消息事件。每增加一个矢量字库，会增加约 3.2KB 的字库对象，如果是位图字库，就只会增加 64B。
图片管理器	24B	图片被缓存时，缓存动态数组内存变大，同时会创建 bitmap_cache_t 对象（约 84B），以及开辟图片名称字符串（不超过 32B）。
窗口管理器	810B	688B 为缺省窗口管理器对象，7B 为管理器状态，12B 为风格对象，7B 为风格状态，32B 为注册本地信息修改后的消息事件，16B 为全局的事件分发器，16B 为 window_manager 对象的事件分发器，32B 为注册窗口管理器销毁后的消息事件。
控件工厂管理器	约 3.3KB	36B 为控件工厂对象，45 * 36B 为基本控件注册对象，37 * 36B 为扩展控件注册对象，368B 为存放控件注册对象的动态数组。
lcd_mem_t 对象	312B	272B 为 lcd_mem_t 对象，两个 20B 分别为 online 和 offline 的 graphic_buffer_t 对象，显存占用不算在内。
native_window 对象	338B	160 为 native_window 对象，44B 为 custom_props 属性大小，120B 为长度为 5 的属性动态数组，14B 为增加新的 custom_props 属性名字（字符串）。
消息队列对象	968B	消息队列对象限制了一帧内最多 20 个消息，超过会抛弃后面进来的消息。
事件源管理器	212B	52B 为 event_source_manager 对象，20B 为长度为 5 的存放事件源对象的动态数组，20B 为长度为 5 的临时存放需要分发的事件对象源动态数组，2*60B 分别为封装定时管理器和 idle 管理器的事件源对象开辟的。

14. 附录四：控件的内存和 ROM 占用详细表格

控件名称	ROM 占用	最少内存占用	需要开启的功能模块
dialog	大约 1.2KB	424B	绘制高亮需要开启对话框高亮模块
window	567B	412B	
dialog_title	249B	148B	
dialog_client	270B	148B	
image	大约 1KB	168B	旋转和缩放需要开启矢量画布模块
button	大约 1.7B	160B	
label	大约 3.3KB	148B	换行需要开启标准的 UNICODE 换行模块
progress_bar	大约 2.5KB	160B	
slider	大约 4.5KB	224B	
check_button	大约 1.6KB	140B	
radio_button	大约 1.6KB	140B	
pages	大约 2.1KB	264B	若使用 vpage，则需要开启控件动画模块
button_group	225B	136B	
popup	大约 1.4KB	424B	
color_tile	大约 2KB	168B	
clip_view	558B	136B	
group_box	222B	136B	
system_bar	983B	444B	
system_bar_bottom	983B	444B	
calibration_win	大约 1KB	480B	
view	509B	140B	
overlay	732B	412B	
edit	大约 19.5KB	868B	若支持软键盘输入，则需要开启对应的输入法模块
tab_control	224B	136B	
tab_button	大约 2.5KB	240B	
tab_button_group	大约 1.2KB	220B	
spin_box	528B	大约 1.4KB	
dragger	大约 1.4KB	172B	
combo_box	大约 5.5KB	大约 1.2KB	
combo_box_item	大约 1.3KB	144B	
grid	208B	136B	
grid_item	208B	136B	
row	208B	136B	
column	211B	136B	
app_bar	212B	136B	
digit_clock	928B	368B	
rich_text	大约 7.3KB	228B	控件动画模块
rich_text_view	863B	144B	
color_picker	大约 3.7KB	180B	
color_component	大约 2.4KB	168B	
scroll_view	大约 6.6KB	272B	控件动画模块
list_view	大约 3KB	160B	控件动画模块

续上表

控件名称	ROM 占用	最少内存占用	需要开启的功能模块
list_view_h	大约 1KB	148B	控件动画模块
list_item	大约 1KB	152B	
scroll_bar	大约 5.8KB	164B	控件动画模块
scroll_bar_d	大约 5.8KB	920B	控件动画模块
scroll_bar_m	大约 5.8KB	164B	控件动画模块
slide_view	大约 7.3KB	344B	控件动画模块、矢量画布模块
slide_indicator	大约 8.3KB	268B	控件动画模块、矢量画布模块
slide_indicator_arc	大约 8.3KB	268B	控件动画模块、矢量画布模块
keyboard	大约 2.1KB	528B	输入法模块
lang_indicator	739B	180B	输入法模块
candidates	大约 2.8KB	244B	
time_clock	大约 4.3KB	376B	矢量画布模块
gauge	794B	144B	
gauge_pointer	大约 2.4KB	180B	矢量画布模块
text_selector	大约 6.9KB	232B	控件动画模块
switch	大约 3.1KB	196B	控件动画模块、圆角需要矢量画布模块
image_animation	大约 3.5KB	184B	
progress_circle	大约 3KB	176B	矢量画布模块
svg	大约 19KB	168B	矢量画布模块
gif	951B	176B	stb 图片解码模块
canvas	211B	136B	
image_value	大约 2.3KB	184B	
slide_menu	大约 5KB	196B	控件动画模块
mutable_image	大约 1.3KB	316B	旋转需要开启矢量画布模块
mledit	大约 6.9KB	大约 7.4KB	若支持软键盘输入，则需要开启对应的输入法模块
line_number	大约 1KB	160B	
hscroll_label	大约 3.5KB	360B	
combo_box_ex	601B	大约 1.4KB	
draggable	大约 1.9KB	280B	

需要注意的是：

1. 上表中的内存是创建一个控件的内存。
2. 有部分控件会因为属性配置不同，内存也会有所变化，这里只列出一般默认情况下的内存大小，也可以简单理解为最少的内存占用。
3. AWTK 支持动态增加自定义属性和增加 inline_style 属性以及增加字符串属性等，都可能导致控件的内存变大，比如说给某个控件设置 text 属性或 name 属性（这两个属性都是字符串类型的），控件的内存占用也会增加。
4. 如果风格中使用圆角边框或者缩放，则需要开启矢量画布模块。
5. 如果控件中用到 file:// 绝对路径加载资源，则需要支持文件系统模块。
6. 如果使用了控件动画中的旋转/缩放动画，则需要开启矢量画布模块。

注: 上述功能模块的详细说明以及开启/关闭的方法可以查阅本文附录二：AWTK 可裁剪的功能模块。

诚信共赢，持续学习，客户为先，专业专注，只做第一

广州致远电子股份有限公司

更多详情请访问 www.zlg.cn

欢迎拨打全国服务热线 400-888-4005