VFS 与 SPIFFS 文件系统

淘宝店铺：

PC 端：

http://n-xytrt8gqu585po94mwj5atokcyd4.taobao.com/index.htm

手机端：

https://shop.m.taobao.com/shop/shop_index.htm?sellerId=755668508&shopId=104493595&inShopPageId=423890608&pathInfo=shop/index2

资料下载地址：

链接：https://pan.baidu.com/s/1kCjD8yktZECSGmHomx_veg?pwd=q8er
提取码：q8er

源码下载地址：

https://gitee.com/vi-iot/esp32-board.git

一、VFS 虚拟文件系统

先来看下文件系统的定义，文件系统是操作系统中用于组织、管理和存储持久性数据的一个关键组件。它是方法和数据结构的集合，使操作系统能够有效地在存储设备（如硬盘驱动器、固态硬盘、USB 闪存驱动器等）上存储、检索和管理文件。

文件系统通常由三个主要部分组成：

1）**文件系统的接口**：用户和应用程序与文件系统交互的方式。

2）**对对象操纵和管理的软件集合**：实现文件创建、删除、读取、写入、重命名等操作的系统软件。

3）**对象及属性**：实际存储的数据文件以及与之相关的元数据信息。

当我们使用标准文件操作时，比如我们使用 fwrite(buffer,size,count,file)函数时，我们不用关心到底是写入到磁盘上哪个地址，偏移量是多少等等诸如此类的硬件底层问题，因为这些操作文件系统已经帮我们处理好了，我们只需要关注往哪个文件写入什么内容，从哪个文件读取什么内容即可，文件系统帮我们把这些文件有效的管理组织起来，形成包括文件和目录的层次结构。

在 esp-idf 中虚拟文件系统 (VFS) 组件为驱动程序提供一个统一接口，可以操作类文件对象。这类驱动程序可以是 FAT、SPIFFS 等真实文件系统，也可以是提供文件类接口的设备驱动程序。

VFS 组件支持 C 库函数（如 fopen 和 fprintf 等）与文件系统 (FS) 驱动程序协同工作。在高层级，每个 FS 驱动程序均与某些路径前缀相关联。当一个 C 库函数需要打开文件时，VFS 组件将搜索与该文件所在文件路径相关联的 FS 驱动程序，并将调用传递给该驱动程序。针对该文件的读取、写入等其他操作的调用也将传递给这个驱动程序。

例如，使用 /fat 前缀注册 FAT 文件系统驱动，之后即可调用 fopen("/fat/file.txt", "w")。之后，VFS 将调用 FAT 驱动的 open 函数，并将参数 /file.txt 和合适的打开模式传递给 open 函数；后续对返回的 FILE* 数据流调用 C 库函数也同样会传递给 FAT 驱动。

如需注册 FS 驱动程序，应用程序首先要定义一个 esp_vfs_t 结构体实例，并用指向 FS API 的函数指针填充它。

```
esp_vfs_t myfs = {
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
    .open = &myfs_open,
    .fstat = &myfs_fstat,
    .close = &myfs_close,
    .read = &myfs_read,
};
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

在上述代码中需要用到 read、write 或 read_p、write_p，具体使用哪组函数由 FS 驱动程序 API 的声明方式决定。

示例 1：声明 API 函数时不带额外的上下文指针参数，即 FS 驱动程序为单例模式，此时使用 write

```
ssize_t myfs_write(int fd, const void * data, size_t size);
// In definition of esp_vfs_t:
.flags = ESP_VFS_FLAG_DEFAULT,
.write = &myfs_write,// ... other members initialized
// When registering FS, context pointer (third argument) is NULL:
```

```
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

示例 2：声明 API 函数时需要一个额外的上下文指针作为参数，即可支持多个 FS 驱动程序实例，此时使用 write_p

```
ssize_t myfs_write(myfs_t* fs, int fd, const void * data, size_t size);
// In definition of esp_vfs_t:
.flags = ESP_VFS_FLAG_CONTEXT_PTR,
.write_p = &myfs_write,// ... other members initialized
// When registering FS, pass the FS context pointer into the third argument
// (hypothetical myfs_mount function is used for illustrative purposes)
myfs_t* myfs_inst1 = myfs_mount(partition1->offset, partition1->size);
ESP_ERROR_CHECK(esp_vfs_register("/data1", &myfs, myfs_inst1));
// Can register another instance:
myfs_t* myfs_inst2 = myfs_mount(partition2->offset, partition2->size);
ESP_ERROR_CHECK(esp_vfs_register("/data2", &myfs, myfs_inst2));
```

已注册的 FS 驱动程序均有一个路径前缀与之关联，此路径前缀即为分区的挂载点。

如果挂载点中嵌套了其他挂载点，则在打开文件时使用具有最长匹配路径前缀的挂载点。例如，假设以下文件系统已在 VFS 中注册：

在 /data 下注册 FS 驱动程序 1

在 /data/static 下注册 FS 驱动程序 2

那么：

打开 /data/log.txt 会调用驱动程序 FS1；

打开 /data/static/index.html 需调用驱动程序 FS2；

即便 FS 驱动程序 2 中没有/index.html，也不会在 FS 驱动程序 1 中查找 /static/index.html。

挂载点名称必须以路径分隔符 (/) 开头，且分隔符后至少包含一个字符。但在以下情况中，VFS 同样支持空的挂载点名称：1. 应用程序需要提供一个”最后方案“下使用的文件系统；2. 应用程序需要同时覆盖 VFS 功能。如果没有与路径匹配的前缀，就会使用到这种文件系统。

VFS 不会对路径中的点 (.) 进行特殊处理，也不会将 .. 视为对父目录的引用。在上述示例中，使用 /data/static/../log.txt 路径不会调用 FS 驱动程序 1 打开 /log.txt。特定的 FS 驱动程序（如 FATFS）可能以不同的方式处理文件名中的点。

执行打开文件操作时，FS 驱动程序仅得到文件的相对路径（挂载点前缀已经被去除）：

以 /data 为路径前缀注册 myfs 驱动；

应用程序调用 fopen("/data/config.json", ...)；

VFS 调用 myfs_open("/config.json", ...)；

myfs 驱动打开 /config.json 文件。

VFS 对文件路径长度没有限制，但文件系统路径前缀受 ESP_VFS_PATH_MAX 限制，即路径前缀上限为 ESP_VFS_PATH_MAX。各个文件系统驱动则可能会对自己的文件名长度设置一些限制。

以上内容是我从官方资料中筛选出来的，大家可能看完了也不是很懂，但没关系，看不懂有两种方法，一是再几遍，二就是直接简单点，用代码演示一下。在代码演示之前，还需要讲解一个文件系统，那就是 esp-idf 里面的 spiffs 文件系统。

二、SPIFFS 文件系统

SPIFFS 是一个用于 SPI NOR flash 设备的嵌入式文件系统，支持磨损均衡、文件系统一致性检查等功能。在介绍 ESP32 分区表的时候，我们知道了分区类型 Type 有两种，一种是 app，另一种是 data，如果我们想要用到 spiffs 进行存储时，我们需要新建一个分区，然后指定 Type=data，子类型 Subtype=spiffs，表示此分区用于 spiffs 文件系统存储。

```
# Note: if you have increased the bootloader
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 1M,
storage,  data, spiffs,  ,        0xF0000,
```

目前 spiffs 文件系统使用有如下限制：

- 目前，SPIFFS 尚不支持目录，但可以生成扁平结构。如果 SPIFFS 挂载在 /spiffs 下，在 /spiffs/tmp/myfile.txt 路径下创建一个文件则会在 SPIFFS 中生成一个名为 /tmp/myfile.txt 的文件，而不是在 /spiffs/tmp 下生成名为 myfile.txt 的文件；
- SPIFFS 并非实时栈，每次写操作耗时不等；
- 目前，SPIFFS 尚不支持检测或处理已损坏的块。
- SPIFFS 只能稳定地使用约 75% 的指定分区容量。
- 当文件系统空间不足时，垃圾收集器会尝试多次扫描文件系统来寻找可用空间。根据所需空间的不同，写操作会被调用多次，每次函数调用将花费几秒。同一操作可能会花费不同时长的问题缘于 SPIFFS 的设计，且已在官方的 SPIFFS github 仓库或是 https://github.com/espressif/esp-idf/issues/1737 中被多次报告。这个问题可以通过 SPIFFS 配置部分缓解。
- 当垃圾收集器尝试多次（默认为 10 次）扫描整个文件系统以回收空间时，在每次扫描期间，如果有可用的数据块，则垃圾收集器会释放一个数据块。因此，如果为垃圾收集器设置的最大运行次数为 n（可通过 SPIFFS_GC_MAX_RUNS 选项配置，该选项位于 SPIFFS 配置中），那么 n 倍数据块大小的空间将可用于写入数据。如果尝试写入超过 n 倍数据块大小的数据，写入操作可能会失败并返回错误。
- 如果 ESP32 在文件系统操作期间断电，可能会导致 SPIFFS 损坏。但是仍可通过 esp_spiffs_check 函数恢复文件系统。详情请参阅官方 SPIFFS FAQ。

三、程序例程

由第一节可知，VFS 是虚拟文件系统，它是一个抽象层概念，在注册时需要把它的操作具体关联到某种实际的存储介质，另外，如果我们要想使用 SPIFFS 文件系统进行存储，也需要配合 VFS 一起使用。在本例程中 VFS 与 SPIFFS 文件系统结合使用，在 esp-idf 中已经将 VFS 与 SPIFFS 的关联操作封装好，我们使用提供的接口就可以轻松的把 SPIFF 文件系统挂载，请看如下代码。具体代码在 esp32-board/spiffs 中

```
#include <stdio.h>
#include <string.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include "esp_err.h"
#include "esp_log.h"
#include "esp_spiffs.h"
```

```
static const char *TAG = "spiffs";
void app_main(void)
{
    ESP_LOGI(TAG, "Initializing SPIFFS");

    esp_vfs_spiffs_conf_t conf = {
      .base_path = "/spiffs",    //可以认为挂着点，后续使用 C 库函数
fopen("/spiffs/...")
      .partition_label = NULL,  //指定 spiffs 分区，如果为 NULL，则默认为分区
表中第一个 spiffs 类型的分区
      .max_files = 5,           //最大可同时打开的文件数
      .format_if_mount_failed = true
    };
    //初始化和挂载 spiffs 分区
    esp_err_t ret = esp_vfs_spiffs_register(&conf);
    //失败处理
    if (ret != ESP_OK) {
        if (ret == ESP_FAIL) {
            ESP_LOGE(TAG, "Failed to mount or format filesystem");
        } else if (ret == ESP_ERR_NOT_FOUND) {
            ESP_LOGE(TAG, "Failed to find SPIFFS partition");
        } else {
            ESP_LOGE(TAG, "Failed to initialize SPIFFS (%s)",
esp_err_to_name(ret));
        }
        return;
    }
    //执行 SPIFFS 文件系统检查
    ESP_LOGI(TAG, "Performing SPIFFS_check().");
    ret = esp_spiffs_check(conf.partition_label);//操作 spiffs 文件系统器件
断电，可能会导致 SPIFFS 损坏，可通过 esp_spiffs_check 恢复
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "SPIFFS_check() failed (%s)", esp_err_to_name(ret));
        return;
    } else {
        ESP_LOGI(TAG, "SPIFFS_check() successful");
    }
    //获取 SPIFFS 可用区域大小
    size_t total = 0, used = 0;
    ret = esp_spiffs_info(conf.partition_label, &total, &used);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to get SPIFFS partition information (%s).
Formatting...", esp_err_to_name(ret));
```

```
        esp_spiffs_format(conf.partition_label);
        return;
    } else {
        ESP_LOGI(TAG, "Partition size: total: %d, used: %d", total, used);
    }
    //可用空间异常，执行 SPIFFS 检查
    if (used > total) {
        ESP_LOGW(TAG, "Number of used bytes cannot be larger than total.
Performing SPIFFS_check().");
        ret = esp_spiffs_check(conf.partition_label);
        if (ret != ESP_OK) {
            ESP_LOGE(TAG, "SPIFFS_check() failed (%s)",
esp_err_to_name(ret));
            return;
        } else {
            ESP_LOGI(TAG, "SPIFFS_check() successful");
        }
    }
    //结合 VFS，可以使用标准 C 库函数进行文件读写
    ESP_LOGI(TAG, "Opening file");
    FILE* f = fopen("/spiffs/hello.txt", "w");
    if (f == NULL) {
        ESP_LOGE(TAG, "Failed to open file for writing");
        return;
    }
    fprintf(f, "Hello World!\n");
    fclose(f);
    ESP_LOGI(TAG, "File written");
    //检查/spiffs/foo.txt 这个文件是否存在，如果存在删除它
    struct stat st;
    if (stat("/spiffs/foo.txt", &st) == 0) {
        // 删除/spiffs/foo.txt 文件
        unlink("/spiffs/foo.txt");
    }
    //重命名文件
    ESP_LOGI(TAG, "Renaming file");
    if (rename("/spiffs/hello.txt", "/spiffs/foo.txt") != 0) {
        ESP_LOGE(TAG, "Rename failed");
        return;
    }
    //打开 foo 文件读取一行
    ESP_LOGI(TAG, "Reading file");
    f = fopen("/spiffs/foo.txt", "r");
    if (f == NULL) {
```

```
        ESP_LOGE(TAG, "Failed to open file for reading");
        return;
    }
    char line[64];
    fgets(line, sizeof(line), f);
    fclose(f);
    // strip newline
    char* pos = strchr(line, '\n');
    if (pos) {
        *pos = '\0';
    }
    ESP_LOGI(TAG, "Read from file: '%s'", line);
    //测试完成，卸载
    esp_vfs_spiffs_unregister(conf.partition_label);
    ESP_LOGI(TAG, "SPIFFS unmounted");
}
```

首先我们需要包含 esp_spiffs.h 头文件，这个头文件声明了与 spiffs 和 VFS 关联的操作

esp_vfs_spiffs_conf_t 结构体定义了一些内容配置，其中比较关键的是.base_path = "/spiffs", //这个可以认为是挂着点，也就是后续可以使用 C 库函数 fopen("/spiffs/...")打开文件的前缀。其余的参数大家看注释便知道是什么意思。

然后调用 esp_vfs_spiffs_register 函数把配置设置进去，这个函数会初始化 SPIFFS 文件系统，然后把底层对 SPIFFS 文件系统的读写操作注册到 VFS，以及将 SPIFFS 文件系统挂载到指定的挂载点”/spiffs”。

esp_spiffs_check 函数会对 SPIFFS 分区进行检查，修复损坏的文件，清理未引用的页面等，官方推荐如果 SPIFFS_info 返回 used 大于 total，或者获取到任何以下错误代码时：SPIFFS_ERR_NOT_FINALIZED 、 SPIFFS_ERR_NOT_INDEX 、 SPIFFS_ERR_IS_INDEX 、 SPIFFS_ERR_IS_FREE 、 SPIFFS_ERR_INDEX_SPAN_MISMATCH 、 SPIFFS_ERR_DATA_SPAN_MISMATCH、SPIFFS_ERR_INDEX_REF_FREE、SPIFFS_ERR_INDEX_REF_LU、SPIFFS_ERR_INDEX_REF_INVALID 、 SPIFFS_ERR_INDEX_FREE 、 SPIFFS_ERR_INDEX_LU 、 SPIFFS_ERR_INDEX_INVALID，都应运行检查。

当挂载成功后，我们就可以使用 fopen、fwrite、rename 等这些 C 语言标准文件操作来对 spiffs 进行操作了，这部分不再叙述。

在对操作完成后，使用 esp_vfs_spiffs_unregister 卸载掉 spiffs 文件系统，这个函数的参数如果是 NULL，则会对分区表中第一个 spiffs 分区进行操作，会检测这个分区是否已经挂载，如果挂载了就会卸载掉这个分区，如果没有则返回错误。