FreeRTOS学习笔记

开发板：正点原子F103战舰

**目录**

[1基础知识 2](#_Toc160009521)

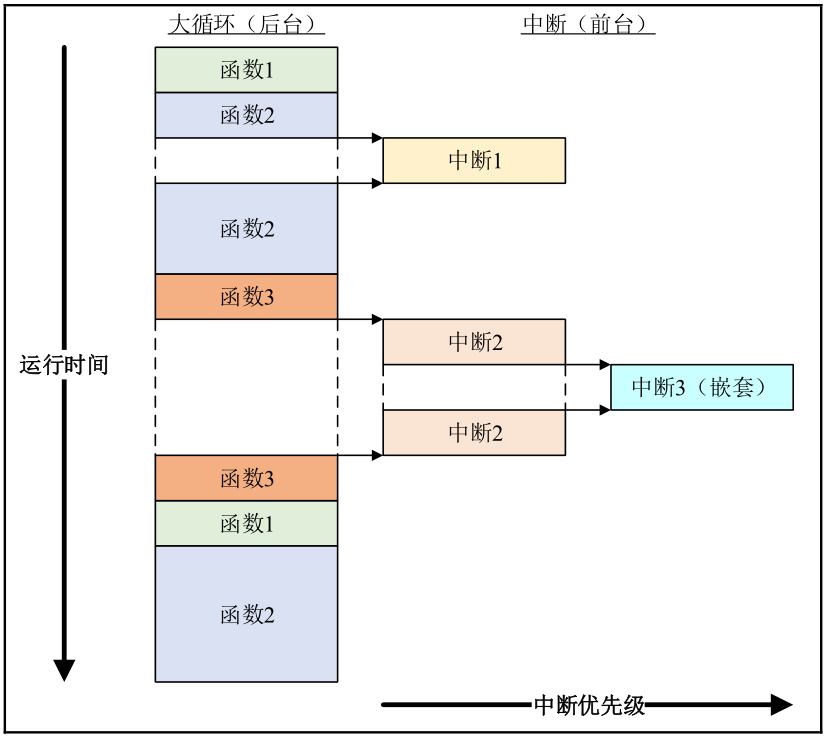
[2. 列表和列表项 4](#_Toc160009522)

[3. 系统启动流程 5](#_Toc160009523)

[4. 任务相关常用API 6](#_Toc160009524)

# 1基础知识

FreeRTOS为免费的实时操作系统，任务和任务管理是RTOS的核心，使用RTOS的目的就是为了使用RTOS的多任务管理能力。FreeRTOS使用的是多任务系统，与裸机的单任务系统相比，实时性得到了提高。

 图示

描述已自动生成

**单任务（前后台）系统 多任务系统**

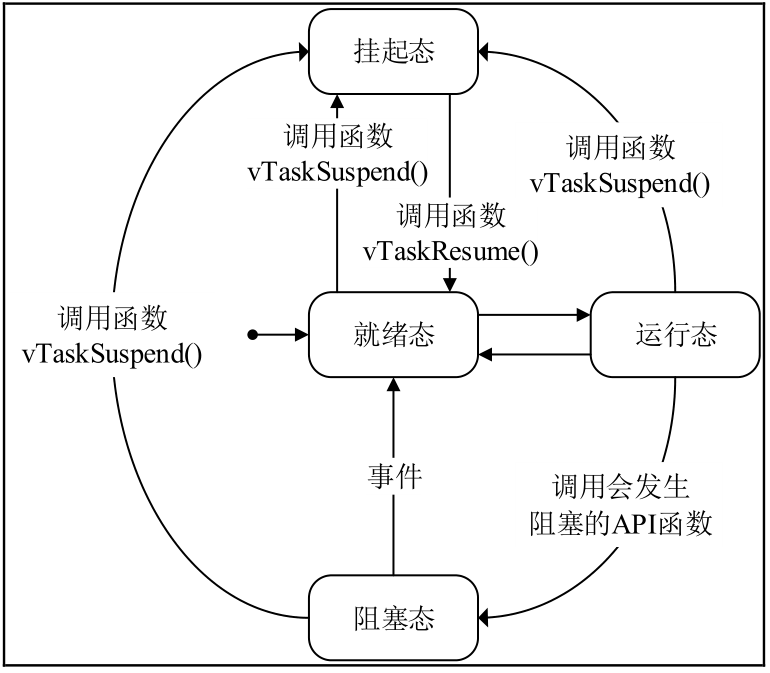
单任务系统一般是在main函数中使用一个大循环，在循环中调用相应的函数以处理相应的事务，这个大循环可以视为程序的前后台，前台为各种中断的中断服务函数，后台为大循环中的函数。因为大循环中的函数没有优先级，只能顺序执行，所以不论事务多紧急，都只能顺序执行各个函数。虽然可使用中断服务函数处理紧急事务，但在大工程项目中，这种单任务系统就会显得力不从心。

多任务系统从宏观上看可以同时执行多个任务，其实CPU在同一时刻只能处理一个任务，但是多任务系统的任务调度器会根据相应算法进行分配CPU的使用权，优先级相同的任务可以轮流运一段极短的时间，即时间片调度，所以宏观角度就是同时执行多个任务。执行任务时，高优先级的任务可以抢占低优先级的任务，即抢占式调度。

通过配置FreeRTOSConfig.h中的宏configMAX\_PRIORITIES，可将每个任务分配一个0~(configMAX\_PRIORITIES-1)的任务优先级，优先级数值越大，优先级越高configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION定义为1时，STM32则使用硬件方法计算下一个要运行的任务，STM32是32位CPU，所以configMAX\_PRIORITIES不能超过32，且为了节省硬件资源，应合理地将宏 configMAX\_PRIORITIES定义为满足应用需求的最小值。

②任务状态

FreeRTOS 中任务存在四种任务状态，分别为运行态、就绪态、阻塞态和挂起态。正在被运行的任务就处于运行态；已经能够被执行的任务就处于就绪态；一个任务因延时一段时间或等待外部事件发生时就处于阻塞态，处于阻塞态的任务通常都有一个阻塞的超时时间，在任务阻塞达到或超过这个超时时间后，即使任务等待的外部事件还没有发生，任务的阻塞态也会被解除；与阻塞态一样，处于挂起态的任务也无法被运行，通过函数 vTaskSuspend()和vTaskResums()进入和退出挂起态。



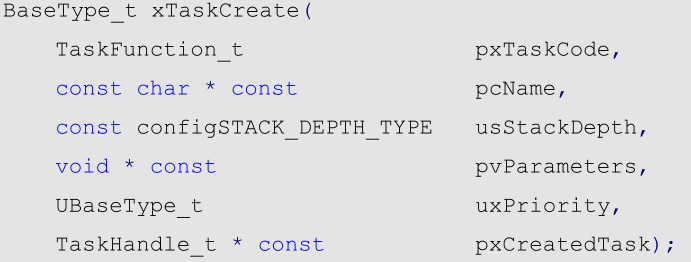
任务状态转换图

FreeRTOS中的每一个已创建任务都包含一个任务控制块，任务控制块是一个结构体变量，用于存储任务的属性。结构体包含很多成员变量，可通过FreeRTOSConfig.h 配置文件中的配置项宏定义进行裁剪。当使用静态方式创建任务时，需要用户自行分配一块内存，作为任务的栈空间，使用动态方式创建任务时，系统则会自动从系统堆中分配一块内存，作为任务的栈空间。创建任务时就需要用到任务控制块，下面介绍常用的任务相关的API函数。

动态方式创建任务：xTaskCreate() 静态方式创建任务：xTaskCreateStatic() 删除任务：vTaskDelete()

使用动态方式创建任务时需要将宏configSUPPORT\_DYNAMIC\_ALLOCATION配置为1，此函数创建的任务会立刻进入就绪态，函数原型如下。xTaskCreateStatic()函数使用静态方式创建任务，任务的任务控制块以及任务的栈空间所需的内存，

需要由用户分配提供，首先需要将宏configSUPPORT\_STATIC\_ALLOCATION 配置为1，任务创建完成后直接进入就绪态。

 图形用户界面, 文本, 应用程序

描述已自动生成

参数介绍如下

|  |  |
| --- | --- |
| **参数** | **描述** |
| pxTaskCode | 指向任务函数的指针 |
| pcName | 任务名，最大长度为 configMAX\_TASK\_NAME\_LEN |
| uxPriority | 任务优先级，最大值为(configMAX\_PRIORITIES-1) |
| **动态创建** | |
| usStackDepth | 任务堆栈大小，单位：字（注意，单位不是字节） |
| pvParameters | 传递给任务函数的参数 |
| pxCreatedTask | 任务句柄，任务成功创建后，会返回任务句柄。任务句柄就是任务的任务控制块 |
| **静态创建** | |
| ulStackDepth | 任务堆栈大小，单位：字（注意，单位不是字节） |
| pvParameters | 传递给任务函数的参数 |
| puxStackBuffer | 任务栈指针，内存由用户分配提供 |
| pxTaskBuffer | 任务控制块指针，内存由用户分配提供 |

示例：（通常使用动态方式创建任务）

1）静态创建任务

#define START\_TASK\_PRIO 1 /\* 任务优先级 \*/

#define START\_STK\_SIZE 128 /\* 任务堆栈大小 \*/

StackType\_t StartTaskStack[START\_STK\_SIZE]; /\* 任务堆栈 \*/

StaticTask\_t StartTaskTCB; /\* 任务控制块 \*/

TaskHandle\_t StartTask\_Handler; /\* 任务句柄 \*/

void start\_task(void \*pvParameters); /\* 任务函数 \*/

StartTask\_Handler = xTaskCreateStatic((TaskFunction\_t )start\_task, /\* 任务函数 \*/

(const char\* )"start\_task", /\* 任务名称 \*/

(uint32\_t )START\_STK\_SIZE, /\* 任务堆栈大小 \*/

(void\* )NULL, /\* 传递给任务函数的参数 \*/

(UBaseType\_t )START\_TASK\_PRIO, /\* 任务优先级 \*/

(StackType\_t\* )StartTaskStack, /\* 任务堆栈 \*/

(StaticTask\_t\* )&StartTaskTCB); /\* 任务控制块 \*/

2）动态创建任务

#define START\_TASK\_PRIO 1 /\* 任务优先级 \*/

#define START\_STK\_SIZE 128 /\* 任务堆栈大小 \*/

TaskHandle\_t StartTask\_Handler; /\* 任务句柄 \*/

void start\_task(void \*pvParameters); /\* 任务函数 \*/

xTaskCreate((TaskFunction\_t )start\_task, /\* 任务函数 \*/

(const char\* )"start\_task", /\* 任务名称 \*/

(uint16\_t )START\_STK\_SIZE, /\* 任务堆栈大小 \*/

(void\* )NULL, /\* 传入给任务函数的参数 \*/

(UBaseType\_t )START\_TASK\_PRIO, /\* 任务优先级 \*/

(TaskHandle\_t\* )&StartTask\_Handler); /\* 任务句柄 \*/

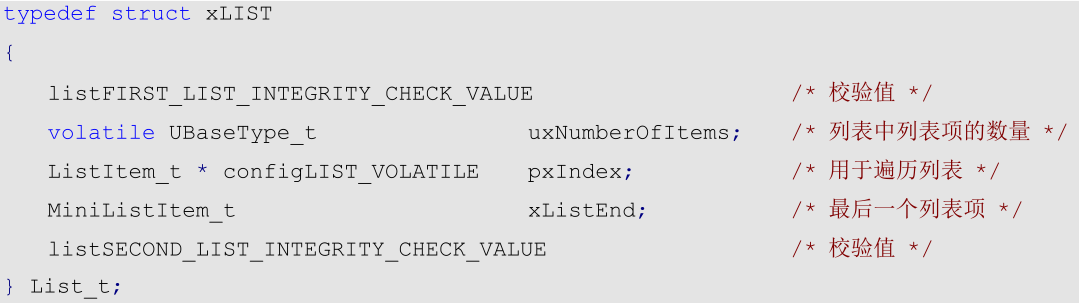
注意，静态创建任务时，还需要用户提供用于提供空闲任务和软件定时器服务任务（如果启用了软件定时器）内存的两个回调函数，这两个回调函数分别为函数 vApplicationGetIdleTaskMemory()和函数vApplicationGetTimerTaskMemory()。具体代码参考正点原子例程。

函数 vTaskDelete()用于删除任务，被删除的任务将被从就绪态任务列表、阻塞态任务列表、挂起态任务列表和事件列表中移除，首先要将宏INCLUDE\_vTaskDelete配置为 1。此函数只有一个参数xTaskToDelete，为待删除任务的任务句柄。空闲任务会负责释放被删除任务中由系统分配的内存，但是由用户在任务删除前申请的内存，则需要由用户在任务被删除前提前释放，否则将导致内存泄露。

函数vTaskSuspend()用于挂起任务，只有一个参数xTaskToSuspend，为待挂起任务的任务句柄；恢复被挂起的任务使用函数 vTaskResume()，此函数页只有一个参数xTaskToResume，为待恢复任务的任务句柄。任务挂起不支持嵌套，即无论任务被挂起多少次，只要恢复一次，此任务就会恢复到就绪态。

# 2. 列表和列表项

列表是 FreeRTOS 中最基本的一种数据结构，其在物理存储单元上是非连续、非顺序的。FreeRTOS 中的列表是一个双向链表，在list.h 文件中有列表的相关定义。



**列表结构体**

其中，校验值我们暂不研究，uxNumberOfItems 用于记录列表中列表项的个数（不包含 xListEnd）；pxIndex 用于指向列表中的某个列表项，一般用于遍历列表中的所有列表项；xListEnd 是一个迷你列表项，一般被设置为最大值，用于将列表中的所有列表项按升序排序时，排在最末尾，同时 xListEnd 也用于挂载其他插入到列表中的列表项。

图形用户界面, 文本, 应用程序

描述已自动生成

**列表项结构体**

列表项是列表中用于存放数据的地方，xItemValue 为列表项的值，对列表进行升序或降序排列时会用到；pxNext和 pxPrevious 分别用于指向列表中列表项的下一个列表项和上一个列表项；pxOwner用于指向包含列表项的对象（通常是任务控制块），因此，列表项和包含列表项的对象之间存在双向链接；pxContainer 用于指向列表项所在列表。

迷你列表项只用于标记列表的末尾和挂载其他插入列表中的列表项，因此不需要成员变量 pxOwner 和 pxContainer，以节省内存开销。

表格

低可信度描述已自动生成

**迷你列表项结构体**

下面介绍常用的列表与列表项相关函数，使用示例见正点原子代码例程。

1）初始化列表：vListInitialise()用于初始化列表，只有一个参数pxList，即待初始化的列表。

2）初始化列表项：vListInitialiseItem()用于初始化列表项，只有一个参数pxItem，即待初始化的列表项。

3）无序插入列表项：vListInsertEnd()插入列表项时为无序插入，直接列表后面插入。

4）有序插入列表项： vListInsert()插入列表项时为有序插入，根据列表项的xItemValue值升序排序后插入到相应位置。

5）移除列表项：uxListRemove()用于移除列表项，只有一个参数pxItemToRemove，为待移除的列表项。

# 3. 系统启动流程

创建好任务后，调用函数 vTaskStartScheduler()即可启动任务调度器，FreeRTOS 便会开始进行任务调度，除非调用函数 xTaskEndScheduler()停止任务调度器，否则不会再返回。

vTaskStartScheduler()主要做了六件事情，介绍如下，具体代码在工程中查看，这里不列出。

1）创建空闲任务，根据是否支持静态内存管理，使用静态方式或动态方式创建空闲任务。

2）创建定时器服务任务，同样根据是否配置支持静态内存管理，使用静态或动态方式创建定时器服务任务。

3）关闭中断，使用 portDISABLE\_INTERRUPT()关闭FreeRTOS管理的中断，主要是为了防止 SysTick 中断在任务调

度器开启之前或过程中产生中断，在开始运行第一个任务时会重新打开中断。

4）初始化一些全局变量，并将任务调度器的运行标志设置为已运行。

5）初始化时基定时器，此定时器用于统计任务运行时间，此功能需要一个硬件定时器提供高精度的计数，这个硬件定时

器就在这里进行配置，如果配置不启用任务运行时间统计功能的，就无需进行这项硬件定时器的配置。

6）最后就是调用函数 xPortStartScheduler()，下面介绍此函数。

函数 xPortStartScheduler()完成启动任务调度器中与硬件架构相关的配置部分，以及启动第一个任务。主要工作如下。

1）如果启用断言，则检测用户在 FreeRTOSConfig.h 文件中对中断的相关配置是否有误。

2）配置 PendSV 和 SysTick 的中断优先级为最低优先级。

3）调用函数 vPortSetupTimerInterrupt()配置SysTick，首先会将SysTick当前计数值清空，并根据FreeRTOSConfig.h 文

件中配置的configSYSTICK\_CLOCK\_HZ（SysTick 时钟源频率）和 configTICK\_RATE\_HZ（系统时钟节拍频率）计算并设置 SysTick的重装载值，然后启动SysTick计数和中断。

4）初始化临界区嵌套计数器为0。

5）调用函数 prvEnableVFP()使能FPU，因为ARM Cortex-M3内核MCU无FPU，此函数仅在ARM Cortex-M4/M7内核

MCU平台上被调用，执行改函数后FPU被开启。

6）将 FPCCR寄存器的[31:30]置1，这样在进出异常时，FPU 的相关寄存器就会自动地保存和恢复，同样地，因为 ARM

Cortex-M3 内核 MCU 无 FPU，此当代码仅在 ARM Cortex-M4/M7 内核 MCU 平台上被调用。

7）调用函数 prvStartFirstTask()启动第一个任务，下面介绍此函数。

函数 prvStartFirstTask()用于初始化启动第一个任务前的环境，主要是重新设置 MSP 指针，并使能全局中断。

MSP指针：

程序在运行过程中需要一定的栈空间来保存局部变量等一些信息。当有信息保存到栈中时，MCU会自动更新SP指针，使SP指针指向最后一个入栈的元素，那么程序就可以根据SP指针来从栈中存取信息。对于正点原子的 STM32F1、STM32F4、STM32F7 和 STM32H7 开发板上使用的 ARM Cortex-M 的 MCU 内核来说，ARM Cortex-M 提供了两个栈空间，这两个栈空间的堆栈指针分别是MSP（主堆栈指针）和PSP（进程堆栈指针）。在FreeRTOS中MSP是给系统栈空间使用的，而PSP是给任务栈使用的，也就是说，FreeRTOS 任务的栈空间是通过 PSP 指向的，而在进入中断服务函数时，则是使用MSP指针。当使用不同的堆栈指针时，SP会等于当前使用的堆栈指针。

我们需要将MSP指向中断向量表的第一个成员，即栈底指针。这个操作相当于丢弃了程序之前保存在栈中的数据，因为FreeRTOS从开启任务调度器到启动第一个任务都是不会返回的，是一条不归路，因此将栈中的数据丢弃，也不会有影响。

图形用户界面

低可信度描述已自动生成

重新赋值 MSP后，接下来就重新使能全局中断，因为之前在函数 vTaskStartScheduler()中关闭了FreeRTOS管理的中断，最后使用 SVC 指令，并传入系统调用号0，触发SVC中断。SVC中断函数vPortSVCHandler()就是用来跳转到第一个任务函数中去的。首先会获取优先级最高的任务（也就是即将运行的任务）的任务栈地址，将栈中内容出栈到CPU寄存器中，任务栈中的内容在调用任务创建函数的时候，已经初始化了。然后再设置 PSP 指针，那么，这么一来，任务的运行环境就准备好了。注意，SVC中断服务函数只在这里使用了一次，之后均不调用。

# 4. 任务相关常用API

|  |  |  |  |
| --- | --- | --- | --- |
| **函数** | **描述** | **函数** | **描述** |
| uxTaskPriorityGet() | 获取任务优先级 | vTaskPrioritySet() | 设置任务优先级 |
| uxTaskGetSystemState() | 获取所有任务的状态信息 | vTaskGetInfo() | 获取单个任务的状态信息 |
| xTaskGetApplicationTaskTag() | 获取任务Tag | xTaskGetCurrentTaskHandle() | 获取当前任务的任务句柄 |
| xTaskGetHandle() | 获取指定任务的任务句柄 | xTaskGetIdleTaskHandle() | 获取空闲任务的任务句柄 |
| uxTaskGetStackHighWaterMark() | 获取任务的任务栈历史剩余最小值 | eTaskGetState() | 获取任务状态 |
| pcTaskGetName | 获取任务名 | xTaskGetTickCount() | 获取系统时钟节拍计数器的值 |
| xTaskGetTickCountFromISR() | 中断中获取系统使用节拍计数器的值 | xTaskGetSchedulerState() | 获取任务调度器状态 |
| uxTaskGetNumberOfTasks() | 获取系统中任务的数量 | vTaskList() | 以表格形式获取所有任务的信息 |
| vTaskGetRunTimeStats() | 获取任务的运行时间等信息 | vTaskSetApplicationTaskTag() | 设置任务 |
| TagSetThreadLocalStoragePointer() | 设置任务的独有数据记录数组指针 | GetThreadLocalStoragePointer() | 获取任务的独有数据记录数组指针 |

要使用如上函数，需要先将相关的宏配置为1，如将INCLUDE\_uxTaskPriorityGet配置为1，则使能uxTaskPriorityGet()函数。

# 5. 队列

项目开发中经常会遇到任务与任务之间、任务与中断之间进行信息传递。不使用操作系统时，一般使用全局变量进行信息传递，但在操作系统中使用这种方式会涉及资源管理的问题，比如读写冲突等，FreeRTOS为此提供了队列机制。队列是一种任务到任务、任务到中断、中断到任务数据交流的一种机制。队列不属于某个特定的任务，可以在任何的任务或中断中往队列中写入消息，或者从队列中读取消息。

①数据存储

在队列中可以存储数量有限、大小固定的多个数据，队列中的每一个数据叫做队列项目，队列能够存储队列项目的最大数量称为队列的长度，在创建队列的时候，就需要指定所创建队列的长度及队列项目的大小。因为队列是用来在任务与任务或任务于中断之间传递消息的一种机制，因此队列也叫做消息队列。

队列通常采用FIFO（先进先出）的存储缓冲机制，当有新的数据被写入队列中时，永远都是写入到队列的尾部，而从队列中读取数据时，永远都是读取队列的头部数据。但同时 FreeRTOS的队列也支持将数据写入到队列的头部，并且还可以指定是否覆盖先前已经在队列头部的数据。

②队列读取阻塞

任务从队列读取信息时，可以设定一个阻塞时间，当任务读取的队列内容为空时，任务会变成阻塞态，等待其他任务向队列中添加信息，随后即可读取队列中的信息，读取结束后任务自动变成就绪态。若阻塞时间结束前队列仍然为空，则任务也自动变成就绪态，且不再读取队列中的信息。

一个队列允许被多个任务读取，当多个任务因等待读取队列而进入阻塞态时，若队列中有有效信息，则会根据阻塞的前后和任务优先级决定解除哪个队列读取阻塞任务。

③队列写入阻塞

与队列读取一样，队列写入也可以指定一个阻塞时间，当写入的队列已满时，任务就进入阻塞态，等待队列有空闲位置，若有位置，则写入队列后任务自动变为就绪态，若阻塞时间结束前队列仍没有空闲位置，则任务自动进入就绪态，不再写入。