FreeRTOS 源码解读

Nrush

如果您觉得此份文档对您有所帮助，

可以前往https://github.com/Nrusher/FreeRTOS-Book或https://gitee.com/nrush/FreeRTOS-Book进行 Star，给予作者鼓励!

2019 年 11 月 3 日

特别说明

- 本文档针对的是 FreeRTOS Kernel V10.2.1 源码的解读，涉及平台相关的函数均以 cortex-m3 为例，文中不再进行特殊声明。
- 由于文档还在更新中，可能会根据后续实际情况进行改动。

目录

特别说明 **1**

1 List **2**
1.1 链表及链表项结构体定义 2
1.2 链表的相关操作 5
1.2.1 初始化 5
1.2.2 插入 6
1.2.3 移除 7
1.2.4 遍历 7
1.2.5 其它操作 8

2 Task **11**
2.1 变量及 TCB_t 结构体说明 11
2.1.1 TCB_t 11
2.1.2 状态链表 13
2.1.3 任务调度器操作相关变量 16
2.1.4 任务删除相关 17
2.1.5 系统信息相关 17
2.2 任务的创建与删除 18
2.2.1 任务创建 18

第 1 章　List

链表这一数据结构是 FreeRTOS 的核心数据结构，有关任务调度、延时、阻塞、事件等操作都是通过对链表进行操作进而实现的。本章将详细分析源码文件 list.c，list.h 的内容，为后续的任务队列等的实现奠定基础。

1.1　链表及链表项结构体定义

FreeRTOS 使用的链表结构是环形的双向链表，其链表项 ListItem_t 的定义如下

```
1  struct xLIST_ITEM
2  {
3      // 第一个和最后一个成员值当configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES被使能的时候
           会被设定为一个固定值，用来检验一个列表项数据是否完整
4      listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
5      // 用来给链表排序的值，在不同的应用场景下有不同的含义。
6      configLIST_VOLATILE TickType_t xItemValue;
7      // 指向前一个链表项
8      struct xLIST_ITEM * configLIST_VOLATILE pxNext;
9      // 指向后一个链表项
10     struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
11     // 类似侵入式链表，指向包含该链表项的对象的地址
12     void * pvOwner;
13     // 指向拥有此链表项的链表
14     struct xLIST * configLIST_VOLATILE pxContainer;
15     listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
16 };
17 typedef struct xLIST_ITEM ListItem_t;
```

Listing 1.1: ListItem_t 的定义

这里如果使用 configLIST_VOLATILE，其会被替换为 volatile 关键字，volatile 关键字是让编译器不对该变量进行优化，所谓的优化可以理解为其在生成汇编时，若多次读取该变量时其可能会将变量值放入寄存器中以加快读取速度，而不是真正的读取，这使得当某个变量会快速变化时，优化后“读取”的值并不是变量真实的值。当使用 volatile 关键字时，其会强迫编译器每次使用该变量时都真正的对它进行一次读取。

```
1  #define configLIST_VOLATILE  volatile
```

Listing 1.2: 使用 configLIST_VOLATILE

在链表项的结构体体构造中值得注意的是 pvOwner 和 pxContainer 这两个成员变量。pvOwner 提供了一种可以快速访问由此链表项代表的对象的方法，pxContainer 则提供了一种快速访问其所属链表的方法。这种处理方式大大提高了链表在任务调度等应用中的处理速度，提高系统效率。

Linux 内核中的侵入式链表设计

Linux 内核中也有侵入式的链表的设计，在 Linux 中提供的链表项的定义为

```
1      struct list_head
2      {
3          struct list_head *next, *prev;
4      };
```

使用链表时只需要将其包含进定义的对象中即可

```c
struct node
{
    // 一些其它成员定义....
    char i;
    // 侵入式链表项
    struct list_head list_item;
    // 一些其它成员定义....
    char j;
};
```

在此它没有定义类似 ListItem_t 中 pxContainer 这样的成员变量，其获得包含该链表项的对象地址是通过一段巧妙的宏定义实现的

```c
#define offsetof(s,m) (size_t)&(((s *)0)->m

#define container_of(ptr, type, member)                    \
({                                                         \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);   \
    (type *)( (char *)__mptr - offsetof(type,member) );    \
})
```

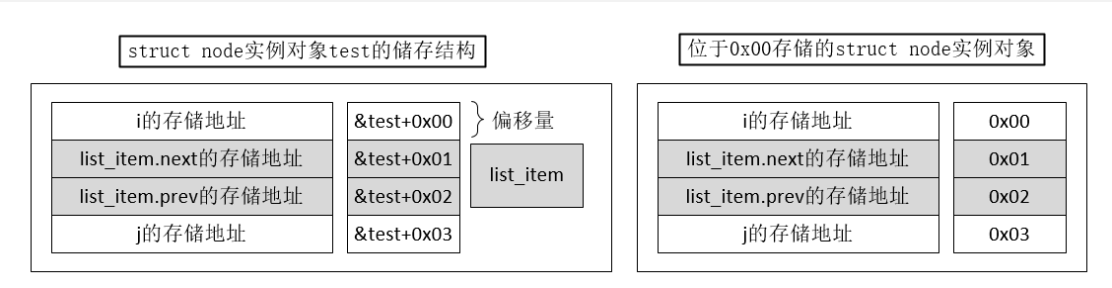
各输入参数的含义为

- ptr：结构体实例成员地址。
- type：结构体名。
- member：成员名。

使用示例

```c
// 包含链表项的对象实例
struct node test;
// 假设通过某种方式（如遍历）获得了链表项地址
struct list_head *list_item_add = &test.list_item;
// 计算对象地址
struct node *test_add = container_of(list_item_add,struct node,
    list_item);
// test_add即为包含list_item_add指向的链表项的对象地址
```

container_of() 的实现思路简单概括就是：将成员变量地址减去成员变量在结构体类型中的便宜量便是实例对象的存储地址。以 struct node 结构体为例，其实例 test 在内存中的存储方式如下图左侧所示。如何获得成员在结构体存储中的偏移量呢？下图右侧给出了解决方法，当 &test=0x00 时，其成员的地址便是所需要的偏移量。



回过头来看 offsetof() 宏，其所作的事就是获得偏移量。而 container_of() 宏中的

```c
(type *)( (char *)__mptr - offsetof(type,member) );
```

便是用成员地址减去偏移量来获得实例的地址。至于 container_of() 宏中的前一句

```c
const typeof( ((type *)0)->member ) *__mptr = (ptr);
```

实时上是起到一个类型检验的作用，拓展关键字 typeof 可以获得变量的类型，如果传入的 ptr 的类型与成员变量类型不符，那么编译器便会抛出警告，便于检查是否出错。注意 typeof 并不是标准 C 中的关键字，如果所用的编译器不支持，可以将第一句删除，将第二句中的 ___mptr 替换为 ptr，宏 container_of() 仍然是正确的。

个人认为，FreeRTOS 并未借鉴这一设计的一个原因可能是出于系统实时性的考虑，毕竟这样的操作是需要耗费一定时间的，而 FreeRTOS 内核中需要频繁使用这一功能，这样的设计必定会降低内核效率。另外 FreeRTOS 的目标平台一般运行速度相对较低，使得内核效率降低更为明显。

```
1  struct xMINI_LIST_ITEM
2  {
3      // 校验值
4      listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
5      // 排序值
6      configLIST_VOLATILE TickType_t xItemValue;
7      // 指向前一个链表项
8      struct xLIST_ITEM * configLIST_VOLATILE pxNext;
9      // 指向后一个链表项
10     struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
11 };
12 typedef struct xMINI_LIST_ITEM MiniListItem_t;
```

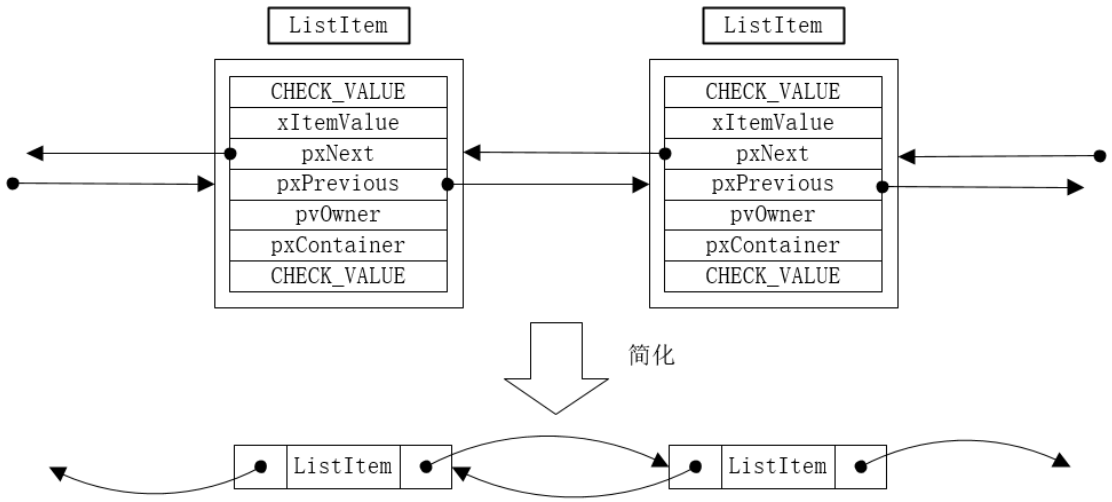
Listing 1.3: MiniListItem_t

链表 List_t 的定义

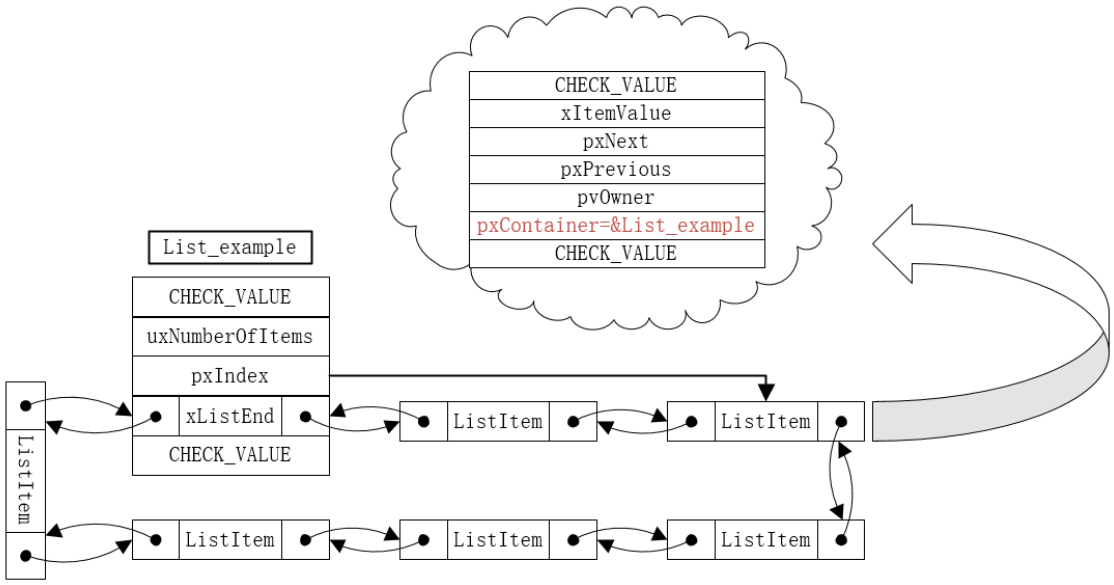
```
1  typedef struct xLIST
2  {
3      // 校验值
4      listFIRST_LIST_INTEGRITY_CHECK_VALUE
5      // 记录该链表里有多少成员
6      volatile UBaseType_t uxNumberOfItems;
7      // 用来遍历链表的指针，listGET_OWNER_OF_NEXT_ENTRY()会使它指向下一个链表项
8      ListItem_t * configLIST_VOLATILE pxIndex;
9      // 链表尾部，为节省内存使用Mini链表项
10     MiniListItem_t xListEnd;
11     // 校验值
12     listSECOND_LIST_INTEGRITY_CHECK_VALUE
13 } List_t;
```

Listing 1.4: List_t 的定义

链表中的链表项间的连接如下图所示



FreeRTOS 中链表的一般结构如下所示结构



1.2 链表的相关操作

1.2.1 初始化

链表项的初始化 vListInitialiseItem() 函数如下

```
1  void vListInitialiseItem( ListItem_t * const pxItem )
2  {
3      // 使该链表项不被任何链表拥有
4      pxItem->pxContainer = NULL;
5      // 写入校验值
6      listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
7      listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
8  }
```

Listing 1.5: vListInitialiseItem() 分析

链表项的初始化十分简单只是将 pxContainer 置为 NULL，设置一下校验值。

链表的初始化函数 vListInitialise() 如下所示

```
1  void vListInitialise( List_t * const pxList )
2  {
3      // step1 把当前指针指向链表尾部
```

```
4       pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );
5
6       // step2 设置排序值为最大，也就是保证xListEnd会被放在链表的尾部
7       pxList->xListEnd.xItemValue = portMAX_DELAY;
8
9       // step3 xListEnd的下一项和前一项都设置为尾部，表示链表为空。
10      pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );
11      pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );
12
13      // step4 链表项数目为0
14      pxList->uxNumberOfItems = ( UBaseType_t ) 0U;
15
16      // step5 写入校验值，用于后续检验，为了保证链表结构体是正确的，没有被覆写
17      listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
18      listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
19  }
```

Listing 1.6: vListInitialise() 分析

1.2.2 插入

链表的插入操作有两个版本的函数，分别是 vListInsertEnd() 和 vListInsert()。其中 vListInsertEnd() 可以理解为是一种无序的插入方法，也就是直接在链表 pxIndex 指向的前一位置插入一个链表项，而 vListInsert() 是有序的插入方法，链表项将会按 xItemValue 的值进行升序插入到链表中去。先看 vListInsertEnd() 插入的实现

```
1   void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
2   {
3   ListItem_t * const pxIndex = pxList->pxIndex;
4
5       // step1 校验链表和链表项
6       listTEST_LIST_INTEGRITY( pxList );
7       listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
8
9       // step2 将链表项嵌入到pxIndex指向的链表项前
10      pxNewListItem->pxNext = pxIndex;
11      pxNewListItem->pxPrevious = pxIndex->pxPrevious;
12
13      // 调试测试用的函数，具体功能不知道，对代码逻辑理解无影响。
14      mtCOVERAGE_TEST_DELAY();
15
16      pxIndex->pxPrevious->pxNext = pxNewListItem;
17      pxIndex->pxPrevious = pxNewListItem;
18
19      // step3 记录链表项属于该链表
20      pxNewListItem->pxContainer = pxList;
21
22      // step5 记录链表中的链表项数目
23      ( pxList->uxNumberOfItems )++;
24  }
```

Listing 1.7: vListInsertEnd() 分析

vListInsert() 的实现比 vListInsertEnd() 稍微复杂一些，其先要查找插入位置再进行插入操作

```
1   void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
2   {
3   ListItem_t *pxIterator;
4   const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
5
6       // step1 校验链表和链表项
7       listTEST_LIST_INTEGRITY( pxList );
8       listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
9
10      // step2 寻找插入位置
11      if( xValueOfInsertion == portMAX_DELAY )
12      {
```

```
13          // 如果链表项的排序数最大，直接在尾部插入，这里相当于做了一个小小的优化。
14          pxIterator = pxList->xListEnd.pxPrevious;
15      }
16      else
17      {
18          // 升序寻找插入位置
19          for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext
                ->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
20          {
21              // 无操作
22          }
23      }
24
25      // step3 进行插入操作
26      pxNewListItem->pxNext = pxIterator->pxNext;
27      pxNewListItem->pxNext->pxPrevious = pxNewListItem;
28      pxNewListItem->pxPrevious = pxIterator;
29      pxIterator->pxNext = pxNewListItem;
30
31      // step4 记录链表项属于该链表
32      pxNewListItem->pxContainer = pxList;
33
34      // step5 记录链表中的链表项数目
35      ( pxList->uxNumberOfItems )++;
36  }
```

Listing 1.8: vListInsert() 分析

1.2.3 移除

FreeRTOS 的链表项移除由函数 uxListRemove() 实现，在链表移除的过程中需要注意的一点是当链表当前的 pxIndex 指向该待移除的链表项时，需要更改 pxIndex 以保证其指向一个合法的值，其具体实现过程如下

```
1   UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
2   {
3   List_t * const pxList = pxItemToRemove->pxContainer;
4
5       // step1 调整前后项指针
6       pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
7       pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
8
9       mtCOVERAGE_TEST_DELAY();
10
11      // step2 保证当前的链表索引指向有效项
12      if( pxList->pxIndex == pxItemToRemove )
13      {
14          pxList->pxIndex = pxItemToRemove->pxPrevious;
15      }
16      else
17      {
18          mtCOVERAGE_TEST_MARKER();
19      }
20
21      // step3 移除的链表项不再被链表拥有
22      pxItemToRemove->pxContainer = NULL;
23
24      // step4 减少链表项数目
25      ( pxList->uxNumberOfItems )--;
26
27      // step5 返回链表的链表项数目
28      return pxList->uxNumberOfItems;
29  }
```

1.2.4 遍历

FreeRTOS 中链表的遍历操作由宏 listGET_OWNER_OF_NEXT_ENTRY() 提供，宏的具体实现如下

```
#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )\
{\
List_t * const pxConstList = ( pxList );\
    // 寻找下一个链表项
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;\
    // 判断是不是不是到了xListEnd，如果是，跳过这一项，因为没有意义
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )->
        xListEnd ) )\
    {\
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;\
    }\
    // 返回拥有该链表项的对象的地址
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;\
}
```

可以看到，这里定义的变量名是 pxTCB(Task Control Block 任务控制块)，这个遍历操作主要是给任务调度管理用的。通过多次调用这一宏可以对链表进行遍历操作。

1.2.5 其它操作

在 list.h 中 FreeRTOS 还以宏的方式实现了其它的一些基本操作，例如 listLIST_IS_INITIALISED() 获取链表是否被初始化等操作。这些宏的实现都较为简单，不再详细列写。

宏定义函数与普通函数的区别

FreeRTOS 的 list.h（其它文件中也有）中定义了大量的宏定义函数。单单从形式看宏定义的函数和普通函数并无太大的区别，但事实上两者还是有很大不同。

- **宏定义函数与普通函数在编译过程上不同**。在编译时，对于宏定义函数而言，预编译时会将这些宏定义函数按展开的规则直接展开成语句，并且宏定义函数在代码中书写多少次，便展开多少次，拷贝相应的代码插入，生成相应的指令，而对于普通函数而言其只会生成一份相应的指令，调用处会生成传参指令和调用指令实现对函数的调用。
- **宏定义函数与普通函数在执行过程上不同**。在实际执行过程中，宏定义式函数所有的语句都是普通语句执行，而普通函数由于需要调用的缘故，需要进行开辟栈空间、压栈、出栈等操作。

基于以上两点不同使得宏函数和普通函数在使用和设计上有很大差异。

由于编译时，宏函数是按照宏定义展开规则进行展开的，这一点在提高它的灵活性的同时也会带来许多问题。例如它可以是一些函数的使用变得更为简单

```
#define DEF_MALLOC(n, type)   ( (type *) malloc((n)* sizeof(type)))
int *ptr;
ptr = (int *) malloc( (5) * sizeof(int) );
ptr = DEF_MALLOC( 5, int );
```

显然宏定函数在书写上要比普通函数简洁美观的多。但宏展开的一大弊端是它不会对参数类型进行检查，也不符合函数调用的一些规则，而且在定义时要格外小心其展开的结果是否是我们想要的结果，往往需要加上 {} 或 () 加以限制，例如以下的代码

```
#define MAX(x, y)   ((x)>(y)?(x):(y))
int x = 1;
int y = 3
MAX(++x,--y);
```

以上代码的本意可能是比较 ++x 和–y 谁更大，但实际情况却是

```
((++x)>(--y)?(++x):(--y))
```

违背了代码原本想要表达的含义。同时宏定义函数实际上被多次展开这一特点使得其在相同代码量和相同插入次数下，编译出的代码量要比普通函数多得多。例如

```
// 若函数fun()和DEF_FUN()代码完全一样则，以下普通函数代码编译结果为
    100byte
fun();
```

```
3  fun();
4  fun();
5  fun();
6  // 那么以下宏定义函数编译结果可能要接近400byte，是普通函数的4倍
7  DEF_FUN();
8  DEF_FUN();
9  DEF_FUN();
10 DEF_FUN();
```

在执行过程中由于出入栈等操作开销，相同的普通函数要比宏定义函数的执行效率低，使用以下代码在 PC 机上进行实验验证

```
1  #include "stdio.h"
2  #include "time.h"
3
4  #define DEF_MAX(x,y) ((x)>(y)?(x):(y))
5
6  int max(int x,int y)
7  {
8      return ((x)>(y)?(x):(y));
9  }
10
11 int main(void)
12 {
13     unsigned long long int i = 0;
14     unsigned long long int times = 100;
15     int x = 1;
16     int y = 2;
17     clock_t start, finish;
18
19     start = clock();
20     for(i=0;i<times;i++)
21     {
22         max(x, y);
23     }
24     finish = clock();
25     printf("common fun time %d\r\n",finish - start);
26
27     start = clock();
28     for(i=0;i<times;i++)
29     {
30         DEF_MAX(x, y);
31     }
32     finish = clock();
33     printf("macro fun time %d\r\n",finish - start);
34
35     return 0;
36 }
```

实验结果如下

次数	10	100	1000	10000	100000
普通函数耗时	10	14	56	482	4808
宏定义函数耗时	5	8	22	178	1672

从实验结果看，宏定义函数效率的确高于普通函数。

在实际应用中到底使用宏定义函数还是普通函数需要根据实际需求而定，个人认为当出现小段代码需要多次调用时可以使用宏函数来提高效率，宏函数应该保持短小简洁。

为结合普通函数定义更加安全，而宏函数较为高效的优点，有些编译器提供 inline 关键字，可以用来声明内连函数。

```
1  inline void fun()
```

```
2  {
3      \\ 一些代码
4  }
```

inline 函数和宏函数一样，它会在代码书写处直接拷贝一份指令，不会像普通函数一样单独生成指令然后调用，这使得其相对普通函数而言其仍然是高效的。但与宏函数不同，它是在编译阶段展开到生成指令中的，而不是预编译阶段展开到代码中，它会进行参数类型的检查，符合函数的一般直觉。另外，在实际运行过程中 inline 函数是可以使用调试器调试的，而宏函数不行。并不是所有函数都适合作为内联函数，内联函数应该满足以下要求，否则编译器可能会忽略 inline 关键字将其作为普通函数处理。

- 不使用循环（for，while）。
- 不使用 switch。
- 不能进行递归调用。
- 代码应短小，不能过长。

第 2 章　Task

任务的调度和管理是 RTOS 中的重要功能，本章将对 task.c 的部分源码进行分析，主要涉及以下几个关键的问题

- 任务调度器的启动与结束
- 任务调度器的挂起与恢复
- 任务的创建
- 任务删除
- 任务挂的挂起和恢复
- 任务间的切换过程
- 任务的延时阻塞的实现

task.c 中的大部分源码是为以上几个问题服务的，但也有不少源码是与事件相关的，这些源码不在本章的讨论范围之内，在后续理解分析源码的过程中，如果涉及相关问题会对其进行相应的分析。

2.1　变量及 TCB_t 结构体说明

本节内容旨在于对 task.c 中运用到的重要变量及结构体进行简要介绍，以方便后续源码的理解。

2.1.1　TCB_t

TCB_t 的全称为 Task Control Block，也就是任务控制块，这个结构体包含了一个任务所有的信息，它的定义以及相关变量的解释如下

```
1  typedef struct tskTaskControlBlock
2  {
3      // 这里栈顶指针必须位于TCB第一项是为了便于上下文切换操作，详见xPortPendSVHandler
           中任务切换的操作。
4      volatile StackType_t    *pxTopOfStack;
5
6      // MPU相关暂时不讨论
7      #if ( portUSING_MPU_WRAPPERS == 1 )
8          xMPU_SETTINGS   xMPUSettings;
9      #endif
10
11     // 表示任务状态，不同的状态会挂接在不同的状态链表下
12     ListItem_t          xStateListItem;
13     // 事件链表项，会挂接到不同事件链表下
14     ListItem_t          xEventListItem;
15     // 任务优先级，数值越大优先级越高
16     UBaseType_t         uxPriority;
17     // 指向堆栈起始位置，这只是单纯的一个分配空间的地址，可以用来检测堆栈是否溢出
18     StackType_t         *pxStack;
19     // 任务名
20     char                pcTaskName[ configMAX_TASK_NAME_LEN ];
21
22     // 指向栈尾，可以用来检测堆栈是否溢出
```

```
23      #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS == 1 ) )
24          StackType_t     *pxEndOfStack;
25      #endif
26
27      // 记录临界段的嵌套层数
28      #if ( portCRITICAL_NESTING_IN_TCB == 1 )
29          UBaseType_t     uxCriticalNesting;
30      #endif
31
32      // 跟踪调试用的变量
33      #if ( configUSE_TRACE_FACILITY == 1 )
34          UBaseType_t     uxTCBNumber;
35          UBaseType_t     uxTaskNumber;
36      #endif
37
38      // 任务优先级被临时提高时，保存任务原本的优先级
39      #if ( configUSE_MUTEXES == 1 )
40          UBaseType_t     uxBasePriority;
41          UBaseType_t     uxMutexesHeld;
42      #endif
43
44      // 任务的一个标签值，可以由用户自定义它的意义，例如可以传入一个函数指针可以用来做
            Hook    函数调用
45      #if ( configUSE_APPLICATION_TASK_TAG == 1 )
46          TaskHookFunction_t pxTaskTag;
47      #endif
48
49      // 任务的线程本地存储指针，可以理解为这个任务私有的存储空间
50      #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
51          void            *pvThreadLocalStoragePointers[
                configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
52      #endif
53
54      // 运行时间变量
55      #if( configGENERATE_RUN_TIME_STATS == 1 )
56          uint32_t        ulRunTimeCounter;
57      #endif
58
59      // 支持NEWLIB的一个变量
60      #if ( configUSE_NEWLIB_REENTRANT == 1 )
61          struct  _reent xNewLib_reent;
62      #endif
63
64      // 任务通知功能需要用到的变量
65      #if( configUSE_TASK_NOTIFICATIONS == 1 )
66          // 任务通知的值
67          volatile uint32_t ulNotifiedValue;
68          // 任务通知的状态
69          volatile uint8_t ucNotifyState;
70      #endif
71
72      // 用来标记这个任务的栈是不是静态分配的
73      #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
74          uint8_t ucStaticallyAllocated;
75      #endif
76
77      // 延时是否被打断
78      #if( INCLUDE_xTaskAbortDelay == 1 )
79          uint8_t ucDelayAborted;
80      #endif
81
82      // 错误标识
83      #if( configUSE_POSIX_ERRNO == 1 )
84          int iTaskErrno;
85      #endif
86
```

```
87  } tskTCB;
88  typedef tskTCB TCB_t;
```

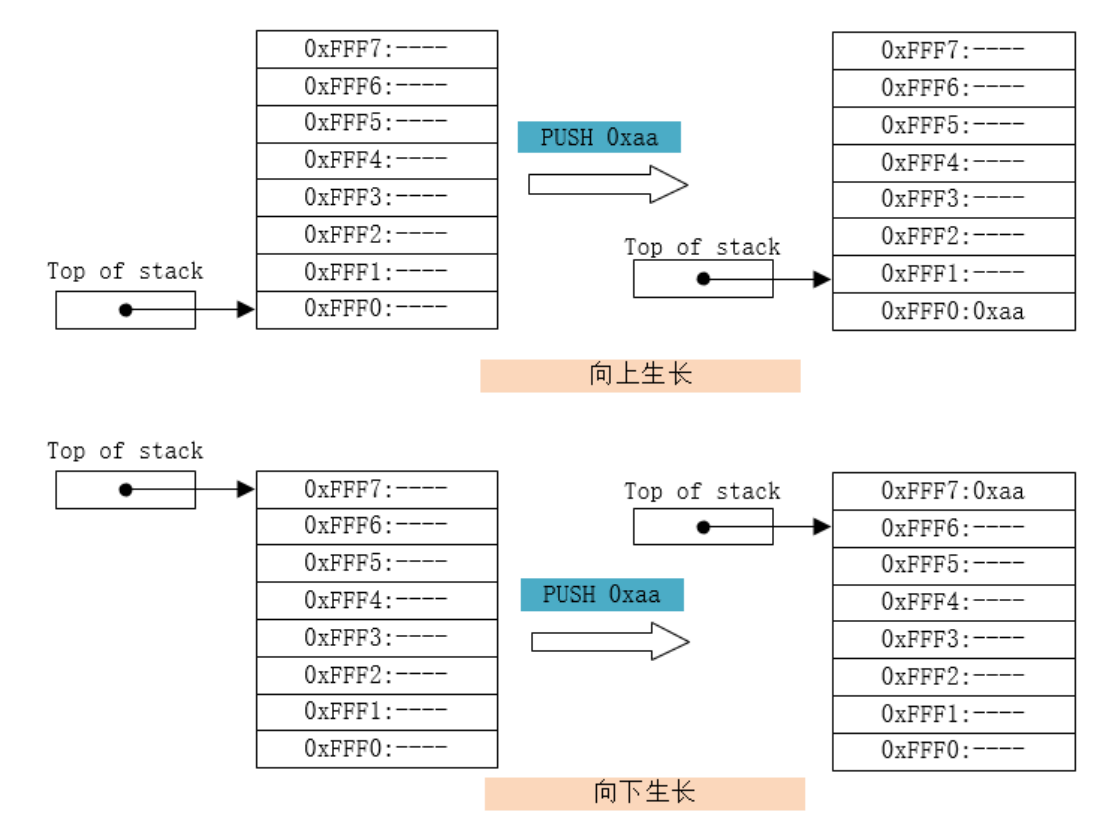
栈的生长方向

在 TCB_t 结构体的定义中可以看到根据栈的生长方式的不同，其将具有不同的成员变量 pxEndOfStack，在这里说明一下栈的生长方式是如何定义的，以及为何生长方式会存在 pxEndOfStack 这一变量的差异。

栈的生长方式可以分为两种，一种是向下生长，一种是向上生长，FreeRTOS 中用 portSTACK_GROWTH 来区分这两种生长方式，portSTACK_GROWTH 大于 0 为向上生长，小于零为向下生长。两种生长方式的区别可以简单概括如下

- 向上生长：入栈时栈顶指针增加，出栈时栈顶指针减小。
- 向下生长：入栈时栈顶指针减小，出栈时栈顶指针增加。

以下是两种生长方式的入栈图，很容易看出区别



为什么会有这两种出入栈方式呢？为何不将所有芯片统一成一种生长方式？这一点应该是芯片设计的实际需要，具体原因无法解答。

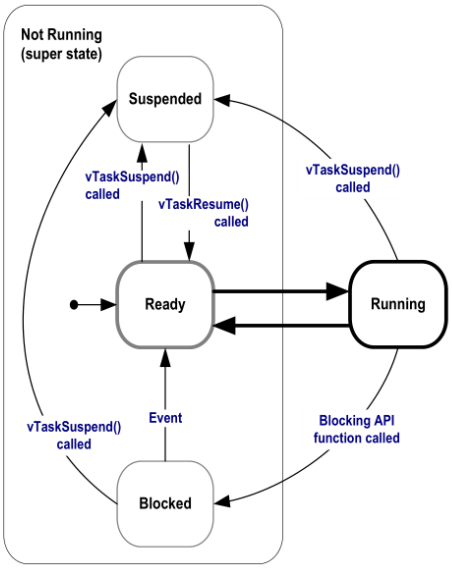
有了上图栈的生长方式为什么会影响成员变量的个数很好理解了，pxStack 是指向栈内存分配的起始地址（低地址），pxEndOfStack 是指向栈的尾部的，当栈是向下生长时，pxStack 和 pxEndOfStack 值是一致的，再定义 pxEndOfStack 浪费了内存，而栈是向上生长时 pxStack 与 pxEndOfStack 的值不一致，如果想知道栈的结束地址，必须要定义一个变量 pxEndOfStack 来存储，以用于后续的栈溢出检测等操作。

2.1.2 状态链表

FreeRTOS 中的任务一共有四种状态分别是运行状态（Running State），就绪状态（Ready State），阻塞状态（Blocked State），挂起状态（Suspended State），其含义可以简单理解为

- 运行状态: 正在执行的任务。
- 就绪状态：等待获得执行权的任务。
- 阻塞状态：直到某些条件达成才会重新进入就绪态等待获得执行权，否则不会执行的任务。
- 挂起状态：除非被主动恢复，否则永远不会执行。

《Mastering the FreeRTOS Real Time Kernel》对这四种任务状态的转换关系描述如下图



这四种链表分别对应着 pxCurrentTCB，pxReadyTasksLists，pxDelayedTaskList，xSuspendedTaskList 这四个变量。除运行状态外，任务处于其它状态时，都是通过将任务 TCB 中的 xStateListItem 挂到相应的链表下来表示的。

pxCurrentTCB

pxCurrentTCB 定义如下

```
1  PRIVILEGED_DATA TCB_t * volatile pxCurrentTCB = NULL;
```

当前运行的任务只可能有一个，因此 pxCurrentTCB 只是单个 TCB_t 指针，它始终指向当前运行的任务。

pxReadyTasksLists

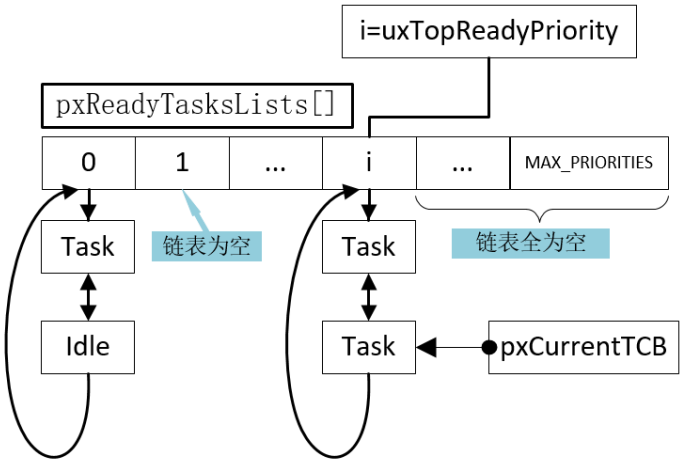
pxReadyTasksLists 定义如下

```
1  PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];
```

pxReadyTasksLists 不是单个链表，它是 configMAX_PRIORITIES 个链表组成的链表数组。链表数组中的每一个成员都是由处于就绪态而又有着相同任务优先级的任务组成的的链表。与之相关的还有一个变量 uxTopReadyPriority。uxTopReadyPriority 的定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxTopReadyPriority = tskIDLE_PRIORITY;
```

uxTopReadyPriority 存储的是有任务挂接的最高优先级。pxReadyTasksLists、pxCurrentTCB 和 uxTopReadyPriority 三者之间的关系可由以下的图来表示



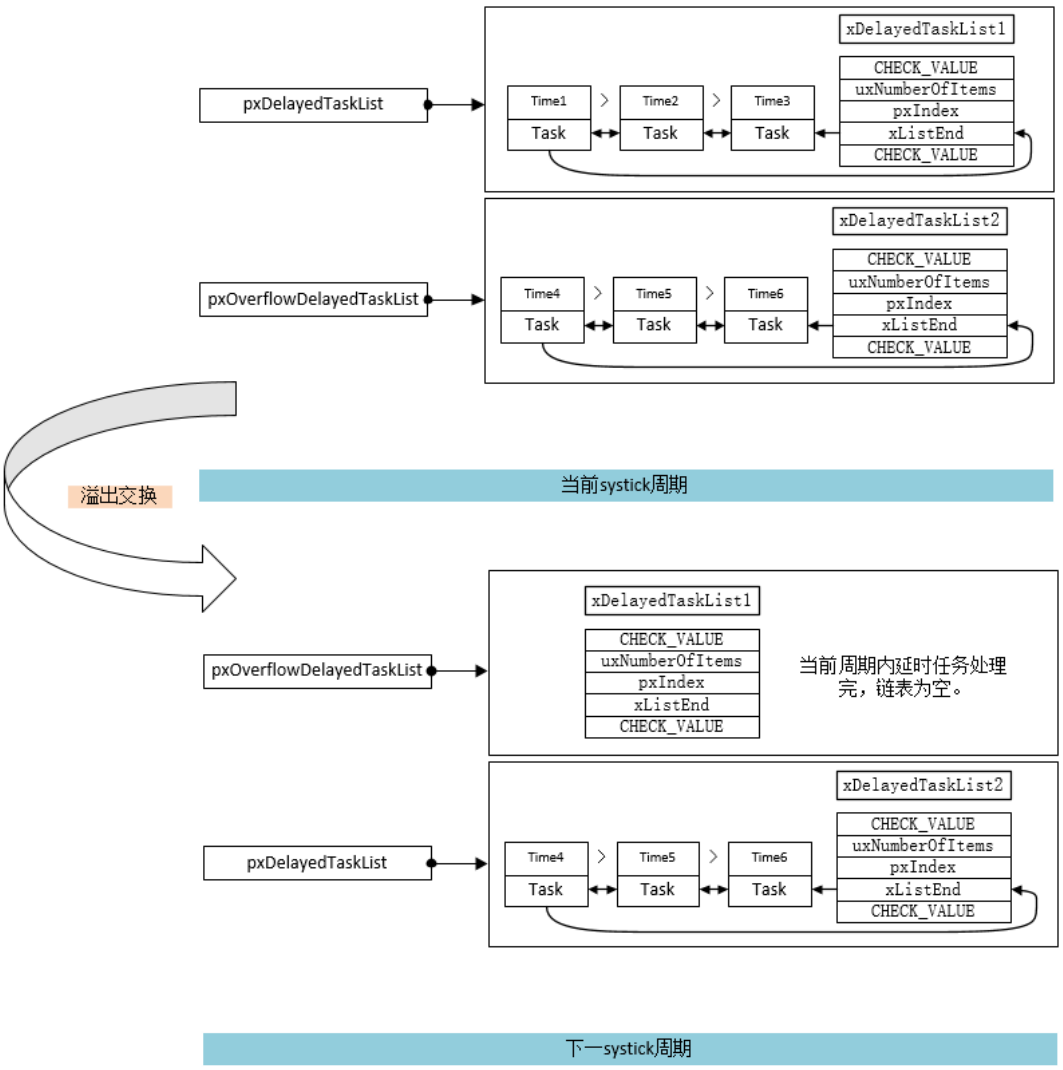
当使用时间片时，pxCurrentTCB 会在有任务挂接的最高优先级链表中遍历，以实现它们对处理器资源的分时共享，这些具体过程会在后面进行详细分析。

pxDelayedTaskList

延时链表的作用不仅是用来处理任务的延时，任务的阻塞也是由它进行实现的（至少目前分析的源码看是这样的）。由于 pxDelayedTaskList 要处理和时间相关的信息，因此需要考虑到系统的 systick 溢出的处理。为了解决这一繁琐的问题，FreeRTOS 设计了两个延时链表和两个延时链表指针来处理溢出问题，它们的定义如下

```
1  PRIVILEGED_DATA static List_t xDelayedTaskList1;
2  PRIVILEGED_DATA static List_t xDelayedTaskList2;
3  PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;
4  PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;
```

其中 pxDelayedTaskList 指向当前工作的延时任务链表，而 pxOverflowDelayedTaskList 指向溢出后的链表，xDelayedTaskList1 和 xDelayedTaskList2 是两个实际链表，其中任务的排列顺序是按退出阻塞时间排序的，也就是链表的第一个成员任务是将最早退出阻塞，而最后一个成员任务是最后退出阻塞的。当系统的 systick 溢出时，pxDelayedTaskList 和 pxOverflowDelayedTaskList 指向的链表地址也会随之交换一次，实现对溢出的处理。对于溢出的处理会在“任务的延时阻塞的实现”进行分析。以下是四个变量之间的关系



与延时任务链表变量为 xNextTaskUnblockTime。其定义如下

```
1  PRIVILEGED_DATA static volatile TickType_t xNextTaskUnblockTime = ( TickType_t ) 0U;
```

xNextTaskUnblockTime 存储的是下一个任务进行解除阻塞操作的时间，用来判断在何时进行解除阻塞操作。

xSuspendedTaskList

xSuspendedTaskList 的定义如下

```
1  PRIVILEGED_DATA static List_t xSuspendedTaskList;
```

其是一个普通的链表，下面挂接的是处于挂起状态的任务。

2.1.3 任务调度器操作相关变量

xSchedulerRunning

xSchedulerRunning 定义如下

```
1  PRIVILEGED_DATA static volatile BaseType_t xSchedulerRunning = pdFALSE;
```

该变量表示任务调度器是否已经运行（挂起的任务调度器也算在运行状态）。

uxSchedulerSuspended

uxSchedulerSuspended 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxSchedulerSuspended = ( UBaseType_t )
       pdFALSE;
```

uxSchedulerSuspended 的作用是记录任务调度器被挂起的次数，当这个变量为 0（dFALSE）时，任务调度器不被挂起，任务调度正常执行，当这个变量大于 0 时代表任务调度器被挂起的次数。如果执行挂起任务调度器操作该变量值会增加，如果执行恢复任务调度器操作，该变量值会减一，直到它为 0 时才会真正的执行实际的调度器恢复操作，这样可以有效的提高执行效率，这点在后面关于任务调度器的操作上会进一步探讨。

uxPendedTicks

uxPendedTicks 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxPendedTicks = ( UBaseType_t ) 0U;
```

任务调度器在被挂起期间，系统的时间，仍然是需要增加的。挂起期间漏掉的 systick 数目便会被存储在这个变量中，以用于恢复调度器时补上漏掉的 systick。

xPendingReadyList

xPendingReadyList 定义如下

```
1  PRIVILEGED_DATA static List_t xPendingReadyList;
```

这个链表中挂接的是在任务调度器挂起期间解除阻塞条件得到满足的阻塞任务，在任务调度器恢复工作后，这些任务会被移动到就绪链表组中，变为就绪状态。

2.1.4 任务删除相关

xTasksWaitingTermination

xTasksWaitingTermination 定义如下

```
1  PRIVILEGED_DATA static List_t xTasksWaitingTermination;
```

当任务自己删除自己时，其是不能立刻自己释放自己所占用的内存等资源的，其需要将自己挂接到 xTasksWaitingTermination 这个链表下，然后让 IdleTask 来回收其所占用的资源。

uxDeletedTasksWaitingCleanUp

uxDeletedTasksWaitingCleanUp 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxDeletedTasksWaitingCleanUp = (
       UBaseType_t ) 0U;
```

uxDeletedTasksWaitingCleanUp 记录了等待 IdleTask 处理的自己删除自己的任务的数目。

xIdleTaskHandle

xIdleTaskHandle 定义如下

```
1  PRIVILEGED_DATA static TaskHandle_t xIdleTaskHandle = NULL;
```

TaskHandle_t 本质上是指向任务 TCB 的指针，IdleTask 是任务调度器在启动时便自动创建的空闲任务，用于回收内存等操作，这个任务句柄指向 IdleTask。

2.1.5 系统信息相关

xTickCount

xTickCount 定义如下

```
1  PRIVILEGED_DATA static volatile TickType_t xTickCount = ( TickType_t )
       configINITIAL_TICK_COUNT;
```

存储 systick 的值，用来给系统提供时间信息。

xNumOfOverflows

xNumOfOverflows 定义如下

```
1  PRIVILEGED_DATA static volatile BaseType_t xNumOfOverflows  = ( BaseType_t ) 0;
```

这个值保存了 xTickCount 溢出的次数。

uxTaskNumber

uxTaskNumber 定义如下

```
1  PRIVILEGED_DATA static UBaseType_t uxTaskNumber = ( UBaseType_t ) 0U;
```

每创建一个任务，这个值便会增加一次，为每个任务生成一个唯一的序号，供调试工具使用。注意与 uxCurrentNumberOfTasks 区分。

uxCurrentNumberOfTasks

uxCurrentNumberOfTasks 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxCurrentNumberOfTasks  = ( UBaseType_t )
       0U;
```

存储当前任务的数目。

2.2 任务的创建与删除

2.2.1 任务创建

FreeRTOS 提供了以下 4 种任务创建函数

- xTaskCreateStatic()：以静态内存分配的方式创建任务，也就是在编译时便要分配好 TCB 等所需要内存。
- xTaskCreateRestrictedStatic()：以静态内存分配的方式创建任务，需要 MPU。
- xTaskCreate()：以动态内存分配方式创建任务，需要提供 portMolloc() 函数的实现，在程序实际运行时分配 TCB 等所需要内存。
- xTaskCreateRestricted()：以动态内存分配方式创建任务，需要 MPU。

任务创建函数大致可以按内存分配的方式分为静态和动态两种，简单来说就是是否用到了 portMolloc() 函数（关于 portMolloc() 会在 FreeRTOS 内存管理的章节中单独分析），其内容大致相同（MPU 没有研究）。以 xTaskCreate() 函数为例分析任务创建的过程。xTaskCreate() 函数的原型为

```
1  BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
2                          const char * const pcName,
3                          const configSTACK_DEPTH_TYPE usStackDepth,
4                          void * const pvParameters,
5                          UBaseType_t uxPriority,
6                          TaskHandle_t * const pxCreatedTask )
```

各参数的含义如下

- pxTaskCode：指向任务函数的函数指针。
- pcName：任务的名称。
- usStackDepth：栈的深度，这里的栈的单位不是 byte 而是根据平台的位数决定的，8 位，16 位，32 位分别对应 1，2，3，4byte。
- pvParameters：传入任务的参数。
- uxPriority：任务的优先级。数值越大，任务的优先级越高。
- pxCreatedTask：创建的任务的句柄，本质就是一个指向创建任务 TCB 的指针。
- 返回值：pdPass 代表创建任务成功，errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY（pdFalse）代表分配内存时出现错误。

个人认为任务初始化的工作主要分为三步：

- 第一步分配存储空间；
- 第二步初始化栈、填充 TCB 结构体；
- 第三步是将 TCB 挂接到就绪链表中并根据优先级进行任务切换；

下面就对每一步的相关代码进行分析

分配存储空间

根据栈的生长方向，FreeRTOS 采用了两种内存分配顺序。暂且将栈向上生长的分配方式放一边，看一下在栈向下生长时，内存分配过程是怎样的

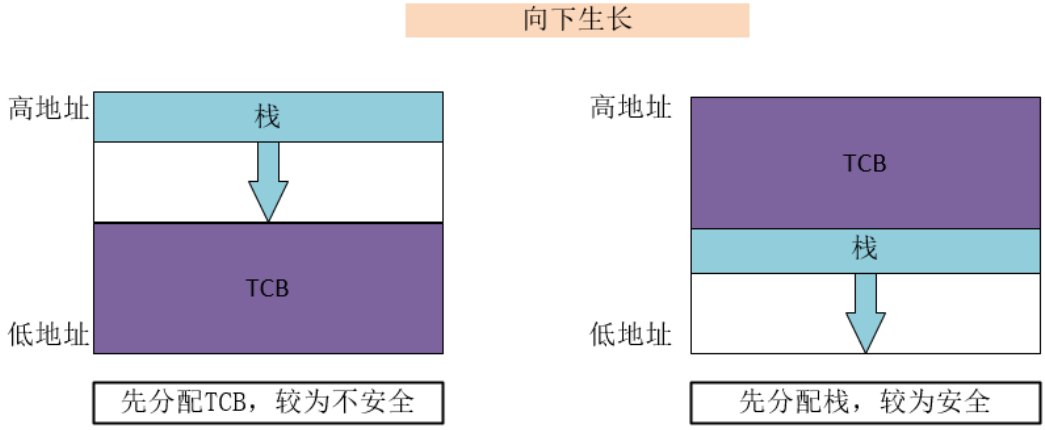
```
1  TCB_t *pxNewTCB;
2  BaseType_t xReturn;
3
4  StackType_t *pxStack;
5
6  // 分配栈空间
7  pxStack = pvPortMalloc( ( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) ) );
8
9  if( pxStack != NULL )
10 {
11     // 分配栈空间成功，分配TCB结构体空间
12     pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
13
14     if( pxNewTCB != NULL )
15     {
16         // 存储栈地址到TCB
17         pxNewTCB->pxStack = pxStack;
18     }
19     else
20     {
21         // 分配TCB结构体空间失败，释放栈空间
22         vPortFree( pxStack );
23     }
24 }
25 else
26 {
27     // 分配栈空间失败
28     pxNewTCB = NULL;
29 }
```

这段代码并不难理解十分简单,唯一值得注意的是栈空间时分配的字节数是 usStackDepth*sizeof(StackType_t)，这意味着我们在设置任务栈空间大小是不是按字节数来分配的，而是和平台相关，例如 32 位的 stm32 分配 1 栈大小等于 4 字节，其余就不多做解释。

再来对比一下栈向上生长的分配方式

```
1  TCB_t *pxNewTCB;
2  BaseType_t xReturn;
3
4  pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
5
6  if( pxNewTCB != NULL )
7  {
8      pxNewTCB->pxStack = ( StackType_t * ) pvPortMalloc( ( ( ( size_t ) usStackDepth )
           * sizeof( StackType_t ) ) );
9
10     if( pxNewTCB->pxStack == NULL )
11     {
12         vPortFree( pxNewTCB );
13         pxNewTCB = NULL;
14     }
15 }
```

两段代码唯一的区别是，分配栈空间和 TCB 结构体空间的顺序不同。由于这两部分的内存分配操作是连续的，这导致其在地址空间上也是连续的，使用合理的分配顺序可以避免栈上溢时本任务的 TCB 数据被破坏。



初始化栈、填充 TCB 结构体

该步主要是由 prvInitialiseNewTask() 函数完成的，其函数原型和参数定义如下

```
1  static void prvInitialiseNewTask(    TaskFunction_t pxTaskCode,
2                                       const char * const pcName,
3                                       const uint32_t ulStackDepth,
4                                       void * const pvParameters,
5                                       UBaseType_t uxPriority,
6                                       TaskHandle_t * const pxCreatedTask,
7                                       TCB_t *pxNewTCB,
8                                       const MemoryRegion_t * const xRegions )
```

- pxNewTCB：TCB 地址。
- xRegions：MPU 相关暂时不讨论。
- 其余参数定义同 xTaskCreate()。

prvInitialiseNewTask() 函数的执行过程大致可以分为

- MPU 相关设置；
- 将栈值设定为特定值，以用于栈最高使用大小检测等功能；
- 计算栈顶指针、栈底指针；
- 复制任务名、写入优先级等相关 TCB 结构体成员赋初值；
- 初始化链表项；
- 对栈进行初始化；

这部分代码整体简单，基本都是一些赋值操作。这里仅对栈顶指针、栈底指针计算和栈的初始化的一些操作进行说明。

在计算栈顶、栈底指针时，都要进行如下的位与操作

```
1  pxTopOfStack = ( StackType_t * ) ( ( ( portPOINTER_SIZE_TYPE ) pxTopOfStack ) & ( ~(
       ( portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK ) ) );
```

这个操作是为了使得栈指针地址是对齐的，portBYTE_ALIGNMENT_MASK 一般被定义为 0x00000007，也就是 8 字节对齐。stm32f103c8t6 在设置时也需要 8 字节对齐，为什么需要字节对齐？又为什么是 8 字节而不是其它呢？

首先为什么需要字节对齐？对于普通的变量而言，字节对齐是为了提高变量的读取效率，具体的原因和处理器的硬件设计有关，其寻址的地址通常不是任意的而是有规律的，例如 32 位处理器，其寻址范围可能只是 4 的倍数，这使得其在处理非字节对齐变量时需要花费更多的时间。对于堆栈地址而言，其存在硬件上的限制也存在软件上的限制，例如 stm32f103c8t6 的 cortex-m3 内核的手册《Cortex-M3 权威指南（中文版初稿）》中的第 27 页明确写道“堆栈指针的最低两位永远是 0，这意味着堆栈总是 4 字节对齐的。”，而在第 39 页也说明“寄存器的 PUSH 和 POP 操作永远都是 4 字节对齐的——也就是说他们的地址必须是 0x4,0x8,0xc,……。”因此，stm32f103c8t6 保证堆栈地址 4 字节对齐是必须的。同时 ARM 编程涉及调用时都需要遵循 AAPCS:《Procedure Call Standard for the ARM Architecture》，规约中表示

> - 5.2.1.1 Universal stack constraints At all times the following basic constraints must hold:Stack-limit < SP <= stack-base. The stack pointer must lie within the extent of the stack.SP mod 4 = 0. The stack must at all times be aligned to a word boundary.
> - 5.2.1.2 Stack constraints at a public interface The stack must also conform to the following constraint at a public interface: SP mod 8 = 0. The stack must be double-word aligned.”

这套规约在限制堆栈地址必须是 4 字节对齐的同时，也要求了在调用入口得 8 字节对齐。4 字节对齐是必须的，8 字节对齐可以不遵守，但这已成为 ARM 堆栈处理的一个标准，如果不遵循程序运行可能会出现问题，也相当于是半强制的了。8 字节对齐的原因暂且分析到这里，更为细节的问题笔者也无能为力。

栈的初始化过程是和硬件平台相关的，其主要目的是将栈“伪装”成这个任务已经执行过一次上下文切换的状态，以保证在任务切换时，任务可以正常运行。由于每个平台该函数的实现都是不一样的，这部分内容将会在任务切换中结合 cortex-m3 来详细分析代码的具体含义。

使任务处于就绪态和任务切换

这部分工作是由 prvAddNewTaskToReadyList() 这一函数实现的，函数的原型如下

```
1  static void prvAddNewTaskToReadyList( TCB_t *pxNewTCB )
```

当任务调度器已经处于工作状态时，也就是 xSchedulerRunning = True 时，prvAddNewTaskToReadyList() 需要做的主要工作只有三件事

1. 记录当前任务数量。
2. 将任务添加到就绪链表中。
3. 根据新加入的优先级判断是否需要进行一次任务切换。

此时执行的相关代码如下

```
1  taskENTER_CRITICAL();
2
3  // step1 使用全局变量uxCurrentNumberOfTasks记录任务数
4  uxCurrentNumberOfTasks++;
5
6  // step2 把新创建任务添加到就绪链表中
7  prvAddTaskToReadyList( pxNewTCB );
8  taskEXIT_CRITICAL();
9
10 // step3 切换任务
11 if( xSchedulerRunning != pdFALSE )
12 {
13     // 如果任务调度器已经工作了，且当前任务的优先级低于新建任务优先级，启动一次任务调
       度切换到新创建的任务
14     if( pxCurrentTCB->uxPriority < pxNewTCB->uxPriority )
15     {
16         taskYIELD_IF_USING_PREEMPTION();
17     }
18     else
19     {
20         mtCOVERAGE_TEST_MARKER();
21     }
```

```
22  }
23  else
24  {
25      mtCOVERAGE_TEST_MARKER();
26  }
```

任务调度器已经处于工作状态时，也就是未调用 xSchedulerRunning = False 函数，prvAddNewTaskToReadyList() 需要负责将相关链表及 pxCurrentTCB 设置成任务调度器已经运行的状态，保证任务调度启动时可以正常工作。此时其工作变为以下三项

1. 记录当前任务数量。
2. 初始化就绪链表以及相关链表，让 pxCurrentTCB 指向优先级最高的任务。
3. 将任务添加到就绪链表中。

此时执行的相关代码如下

```
1   taskENTER_CRITICAL();
2
3   // step1 使用全局变量uxCurrentNumberOfTasks记录任务数
4   uxCurrentNumberOfTasks++;
5
6   // step2 初始化就绪链表以及相关链表，让pxCurrentTCB指向优先级最高的任务。
7   if( pxCurrentTCB == NULL )
8   {
9       // 没有任务，将新添加任务作为当前任务
10      pxCurrentTCB = pxNewTCB;
11      if( uxCurrentNumberOfTasks == ( UBaseType_t ) 1 )
12      {
13          // 如果是首次添加任务，初始化任务链表
14          prvInitialiseTaskLists();
15      }
16      else
17      {
18          mtCOVERAGE_TEST_MARKER();
19      }
20  }
21  else
22  {
23      // 如果任务调度器还没工作，将新添加的任务与当前待执行任务进行优先级比较，优先级高
            则替换当前任务待执任务，任务调度器启动时将执行优先级最高的任务。
24      if( xSchedulerRunning == pdFALSE )
25      {
26          if( pxCurrentTCB->uxPriority <= pxNewTCB->uxPriority )
27          {
28              pxCurrentTCB = pxNewTCB;
29          }
30          else
31          {
32              mtCOVERAGE_TEST_MARKER();
33          }
34      }
35      else
36      {
37          mtCOVERAGE_TEST_MARKER();
38      }
39  }
40
41  // step3 把新创建任务添加到就绪链表中
42  prvAddTaskToReadyList( pxNewTCB );
43  taskEXIT_CRITICAL();
```

在将任务插入就绪链表中时采用的宏 prvAddTaskToReadyList() 相关代码如下

```
1   // 记录就绪任务的最高先级
2   #define taskRECORD_READY_PRIORITY( uxPriority )\
3   {\
4       if( ( uxPriority ) > uxTopReadyPriority )\
```

```
5       {\
6           uxTopReadyPriority = ( uxPriority );\
7       }\
8   }
9
10  #define prvAddTaskToReadyList( pxTCB )\
11      taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );\
12      // 按优先级放到对应的链表下
13      vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->
            xStateListItem ) );\
```

可以看到在插入就绪链表中是，其插入的方式是无序的插入方式 vListInsertEnd()，这也就意味着同等优先级的任务有着同等的地位。

在以上的代码中，首次出现了 taskENTER_CRITICAL(); taskEXIT_CRITICAL(); 这样的临界段操作，对于这些代码，其将会与 vTaskSuspendAll();xTaskResumeAll(); 放在一起进行比较说明。

以上便是 FreeRTOS 中一个任务创建的全部过程。