---

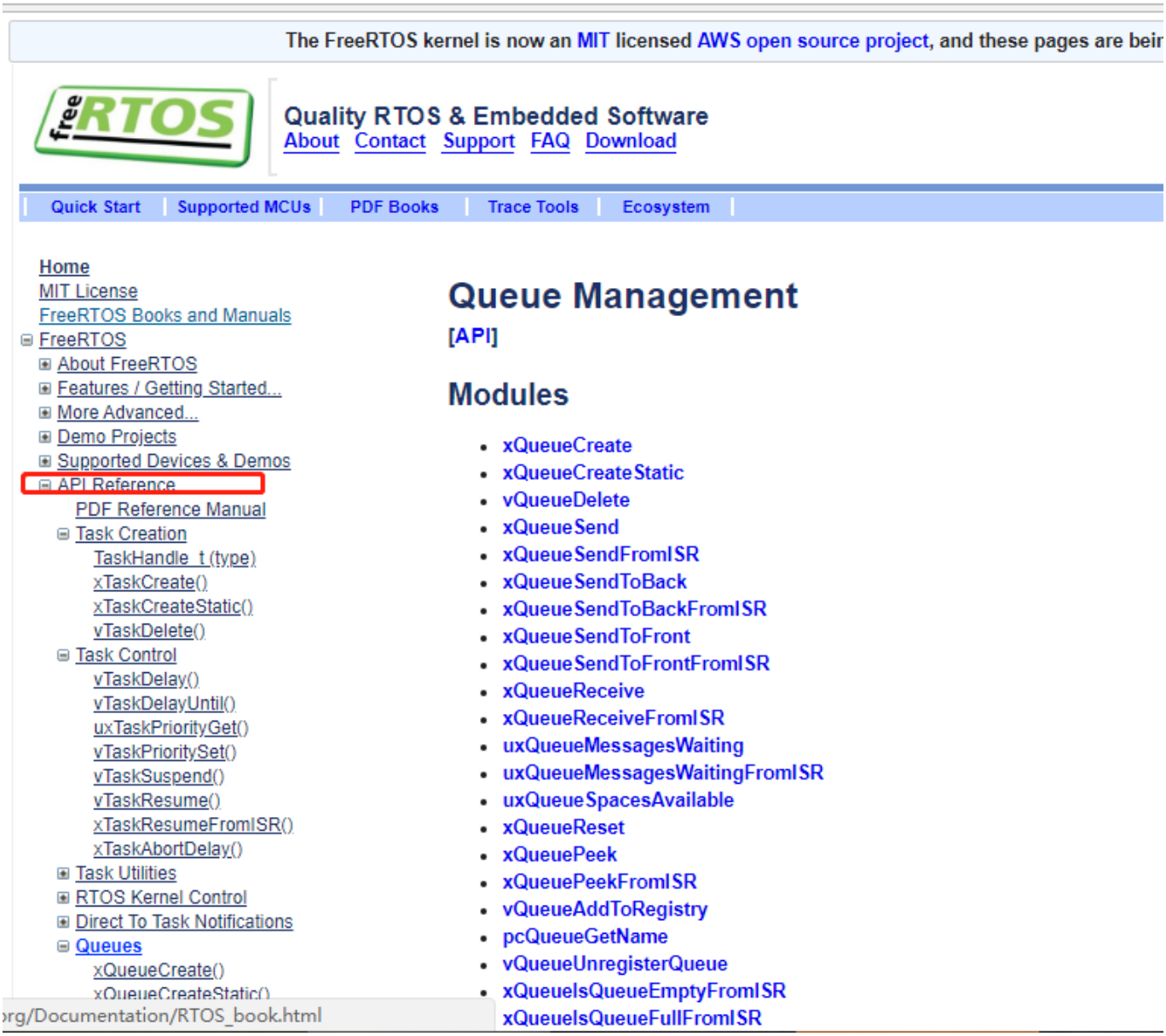
title: 从0开始学FreeRTOS-(消息队列)-5 top: false cover: false toc: true mathjax: false tags:

- FreeRTOS
- RTOS
- 操作系统 date: 2019-09-01 18:20:11 author: coverImg: password: summary: categories: 操作系统

问题解答

曾经有人问我，FreeRTOS那么多API，到底怎么记住呢？ 我想说，其实API不难记，就是有点难找，因为FreeRTOS的API很多都是带参宏，所以跳来跳去的比较麻烦，而且注释也很多，要找还真不是那么容易，不过也没啥，一般都会有API手册的，我就告诉大家一下： **FreeRTOS Kernel: Reference Manual** FreeRTOS内核：参考手册，大家可以在官网下载，也能在后台得到。 当然书本是英文的，如果英语像我这样子不咋地的同学，可以用谷歌浏览器在官网直接看API手册，直接翻译一下就行了。传送门：
https://www.freertos.org/a00018.html


书签
FreeRTOS Kernel
Table of Contents
Welcome
Task and Scheduler API
vTaskAllocateMPURegions()
xTaskAbortDelay ()
xTaskCallApplicationTaskHook()
xTaskCheckForTimeOut()
xTaskCreate()
Summary
Parameters
Return Values
Notes
Example
xTaskCreateStatic()
xTaskCreateRestricted()
vTaskDelay()
vTaskDelayUntil()
vTaskDelete()
taskDISABLE_INTERRUPTS()
taskENABLE_INTERRUPTS()
taskENTER_CRITICAL()
taskENTER_CRITICAL_FROM_ISR()
taskEXIT_CRITICAL()
taskEXIT_CRITICAL_FROM_ISR()
xTaskGetApplicationTaskTag()
xTaskGetCurrentTaskHandle()
xTaskGetIdleTaskHandle()
xTaskGetHandle()
uxTaskGetNumberOfTasks()
vTaskGetRunTimeStats()
参数、返回值、
说明、例子都有
FreeRTOS Kernel
Reference Manual


The FreeRTOS kernel is now an MIT licensed AWS open source project, and these pages are beir
Quality RTOS & Embedded Software
About Contact Support FAQ Download
Quick Start | Supported MCUs | PDF Books | Trace Tools | Ecosystem
Home
MIT License
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
TaskHandle_t (type)
xTaskCreate()
xTaskCreateStatic()
vTaskDelete()
Task Control
vTaskDelay()
vTaskDelayUntil()
uxTaskPriorityGet()
vTaskPrioritySet()
vTaskSuspend()
vTaskResume()
xTaskResumeFromISR()
xTaskAbortDelay()
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
Queues
xQueueCreate()
xQueueCreateStatic()
rg/Documentation/RTOS_book.html
Queue Management
[API]
Modules
xQueueCreate
xQueueCreateStatic
vQueueDelete
xQueueSend
xQueueSendFromISR
xQueueSendToBack
xQueueSendToBackFromISR
xQueueSendToFront
xQueueSendToFrontFromISR
xQueueReceive
xQueueReceiveFromISR
uxQueueMessagesWaiting
uxQueueMessagesWaitingFromISR
uxQueueSpacesAvailable
xQueueReset
xQueuePeek
xQueuePeekFromISR
vQueueAddToRegistry
pcQueueGetName
vQueueUnregisterQueue
xQueueIsQueueEmptyFromISR
xQueueIsQueueFullFromISR

FreeRTOS消息队列

基于 FreeRTOS 的应用程序由一组独立的任务构成——每个任务都是具有独立权限的程序。这些独立的任务之间的通讯与同步一般都是基于操作系统提供的IPC通讯机制，而FreeRTOS 中所有的通信与同步机制都是基于队列实现的。 消息队列是一种常用于任务间通信的数据结构，队列可以在任务与任务间、中断和任务间传送信息，实现了任务接收来自其他任务或中断的不固定长度的消息。任务能够从队列里面读取消息，当队列中的消息是空时，挂起读取任务，用户还可以指定挂起的任务时间；当队列中有新消息时，挂起的读取任务被唤醒并处理新消息，消息队列是一种异步的通信方式。

队列特性

1.数据存储

队列可以保存有限个具有确定长度的数据单元。队列可以保存的最大单元数目被称为队列的“深度”。在队列创建时需要设定其深度和每个单元的大小。 通常情况下，队列被作为 FIFO(先进先出)缓冲区使用，即数据由队列尾写入，从队列首读出。当然，由队列首写入也是可能的。 往队列写入数据是通过字节拷贝把数据复制存储到队列中；从队列读出数据使得把队列中的数据拷贝删除。

2.读阻塞

当某个任务试图读一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其它任务或中断服务例程往其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转移为就绪态。 由于队列可以被多个任务读取，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列数据有效。这种情况下，一旦队列数据有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待最久的任务。

> 说些题外话，ucos中是具有广播消息的，当有多个任务阻塞在队列上，当发送消息的时候可以选择广播消息，那么这些阻塞的任务都能被解除阻塞。

3.写阻塞

与读阻塞想反，任务也可以在写队列时指定一个阻塞超时时间。这个时间是当被写队列已满时，任务进入阻塞态以等待队列空间有效的最长时间。 由于队列可以被多个任务写入，所以对单个队列而言，也可能有多个任务处于阻塞状态以等待队列空间有效。这种情况下，一旦队列空间有效，只会有一个任务会被解除阻塞，这个任务就是所有等待任务中优先级最高的任务。而如果所有等待任务的优先级相同，那么被解除阻塞的任务将是等待最久的任务。

消息队列的工作流程

1.发送消息

任务或者中断服务程序都可以给消息队列发送消息，当发送消息时，如果队列未满或者允许覆盖入队，FreeRTOS 会将消息拷贝到消息队列队尾，否则，会根据用户指定的阻塞超时时间进行阻塞，在这段时间中，如果队列一直不允许入队，该任务将保持阻塞状态以等待队列允许入队。当其它任务从其等待的队列中读取入了数据（队列未满），该任务将自动由阻塞态转为就绪态。当任务等待的时间超过了指定的阻塞时间，即使队列中还不允许入队，任务也会自动从阻塞态转移为就绪态，此时发送消息的任务或者中断程序会收到一个错误码errQUEUE_FULL。 发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，发送的位置是消息队列队头而非队尾，这样，接收者就能够优先接收到紧急消息，从而及时进行消息处理。 下面是消息队

列的发送API接口，函数中有FromISR则表明在中断中使用的。

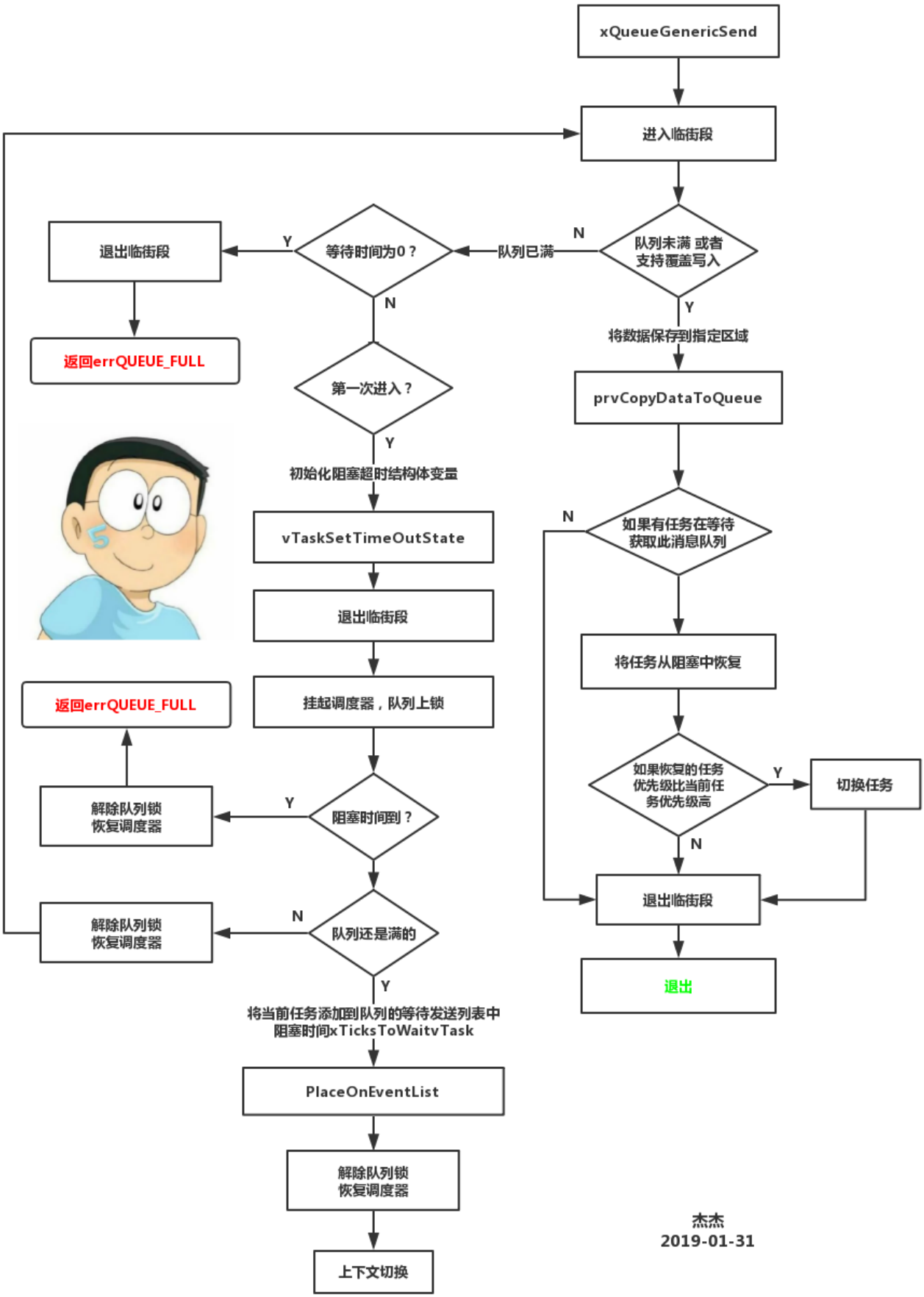
入队方式	API 接口	实际执行函数
从队列尾部入队	xQueueSend()	xQueueGenericSend()
	xQueueSendToBack()	
	xQueueOverWrite()	
从队列首部入队	xQueueSendToFront()	
从队列尾部入队（带中断保护）	xQueueSendFromISR()	xQueueGenericSendFromISR ()
	xQueueSendToBackFromISR ()	
	xQueueOverWriteFromISR ()	
从队列首部入队（带中断保护）	xQueueSendToFront()	


```
1 /*-----------------------------------------------------------*/
 2 BaseType_t xQueueGenericSend( QueueHandle_t xQueue,          (1)
 3                               const void * const pvItemToQueue,      (2)
 4                               TickType_t xTicksToWait,               (3)
 5                               const BaseType_t xCopyPosition )       (4)
 6 {
 7     BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
 8     TimeOut_t xTimeOut;
 9     Queue_t * const pxQueue = ( Queue_t * ) xQueue;
10
11     /* 已删除一些断言操作 */
12
13     for ( ;; ) {
14         taskENTER_CRITICAL();                                          (5)
15         {
16             /* 队列未满 */
17             if ( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
18                  || ( xCopyPosition == queueOVERWRITE ) ) {  (6)
19                 traceQUEUE_SEND( pxQueue );
20                 xYieldRequired =
21           prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition ); (7)
22
23                 /* 已删除使用队列集部分代码 */
24                 /* 如果有任务在等待获取此消息队列 */
25       if ( listLIST_IS_EMPTY(&(pxQueue->xTasksWaitingToReceive))==pdFALSE){ (8)
26                     /* 将任务从阻塞中恢复 */
27             if ( xTaskRemoveFromEventList(
28                   &( pxQueue->xTasksWaitingToReceive ) )!=pdFALSE) { (9)
29                         /* 如果恢复的任务优先级比当前运行任务优先级还高，
30                         那么需要进行一次任务切换 */
31                         queueYIELD_IF_USING_PREEMPTION();    (10)
32                     } else {
33                         mtCOVERAGE_TEST_MARKER();
34                     }
35                 } else if ( xYieldRequired != pdFALSE ) {
36                     /* 如果没有等待的任务，拷贝成功也需要任务切换 */
37                     queueYIELD_IF_USING_PREEMPTION();              (11)
```

```
38                  } else {
39                      mtCOVERAGE_TEST_MARKER();
40                  }
41
42                  taskEXIT_CRITICAL();                            (12)
43                  return pdPASS;
44              }
45              /* 队列已满 */
46              else {                                              (13)
47                  if ( xTicksToWait == ( TickType_t ) 0 ) {
48                      /* 如果用户不指定阻塞超时时间，退出 */
49                      taskEXIT_CRITICAL();                        (14)
50                      traceQUEUE_SEND_FAILED( pxQueue );
51                      return errQUEUE_FULL;
52                  } else if ( xEntryTimeSet == pdFALSE ) {
53                       /* 初始化阻塞超时结构体变量，初始化进入
54               阻塞的时间xTickCount和溢出次数xNumOfOverflows */
55                      vTaskSetTimeOutState( &xTimeOut );              (15)
56                      xEntryTimeSet = pdTRUE;
57                  } else {
58                      mtCOVERAGE_TEST_MARKER();
59                  }
60              }
61          }
62          taskEXIT_CRITICAL();                                    (16)
63          /* 挂起调度器 */
64          vTaskSuspendAll();
65          /* 队列上锁 */
66          prvLockQueue( pxQueue );
67
68          /* 检查超时时间是否已经过去了 */
69          if (xTaskCheckForTimeOut(&xTimeOut, &xTicksToWait)==pdFALSE){ (17)
70              /* 如果队列还是满的 */
71              if ( prvIsQueueFull( pxQueue ) != pdFALSE ) {    (18)
72                  traceBLOCKING_ON_QUEUE_SEND( pxQueue );
73                  /* 将当前任务添加到队列的等待发送列表中
74                     以及阻塞延时列表，延时时间为用户指定的超时时间xTicksToWait */
75                  vTaskPlaceOnEventList(
76                     &( pxQueue->xTasksWaitingToSend ), xTicksToWait );(19)
77                  /* 队列解锁 */
78                  prvUnlockQueue( pxQueue );                  (20)
79
80                  /* 恢复调度器 */
81                  if ( xTaskResumeAll() == pdFALSE ) {
82                      portYIELD_WITHIN_API();
83                  }
84              } else {
85                  /* 队列有空闲消息空间，允许入队 */
86                  prvUnlockQueue( pxQueue );                  (21)
87                  ( void ) xTaskResumeAll();
88              }
89          } else {
90              /* 超时时间已过，退出 */
91              prvUnlockQueue( pxQueue );                          (22)
```

```
92             ( void ) xTaskResumeAll();
93
94             traceQUEUE_SEND_FAILED( pxQueue );
95             return errQUEUE_FULL;
96         }
97     }
98 }
99 /*-----------------------------------------------------------*/
```

如果阻塞时间不为 0，任务会因为等待入队而进入阻塞， 在将任务设置为阻塞的过程中， 系统不希望有其它任务和中断操作这个队列的 xTasksWaitingToReceive 列表和 xTasksWaitingToSend 列表，因为可能引起其它任务解除阻塞，这可能会发生优先级翻转。比如任务 A 的优先级低于当前任务，但是在当前任务进入阻塞的过程中，任务 A 却因为其它原因解除阻塞了，这显然是要绝对禁止的。因此FreeRTOS 使用挂起调度器禁止其它任务操作队列，因为挂起调度器意味着任务不能切换并且不准调用可能引起任务切换的 API 函数。但挂起调度器并不会禁止中断，中断服务函数仍然可以操作队列阻塞列表，可能会解除任务阻塞、可能会进行上下文切换，这也是不允许的。于是，**FreeRTOS解决办法是不但挂起调度器，还要给队列上锁，禁止任何中断来操作队列。** 下面来看看流程图：



相比在任务中调用的发送函数，在中断中调用的函数会更加简单一些， 没有任务阻塞操作。 函数xQueueGenericSend中插入数据后， 会检查等待接收链表是否有任务等待，如果有会恢复就绪。如果恢复的任务优先级比当前任务高， 则会触发任务切换；但是在中断中调用的这个函数的做法是返回一个参数标志是否需要触发任务切换，并不在中断中切换任务。 在任务中调用的函数中有锁定和解锁队列的操作， 锁定队列的时

候， 队列的事件链表不能被修改。 而在被中断中发送消息的处理是： 当遇到队列被锁定的时候， 将新数据插入到队列后， 并不会直接恢复因为等待接收的任务， 而是累加了计数， 当队列解锁的时候， 会根据这个计数， 对应恢复几个任务。 遇到队列满的情况， 函数会直接返回， 而不是阻塞等待， 因为在中断中阻塞是不允许的！！！

```
BaseType_t xQueueGenericSendFromISR(
        QueueHandle_t xQueue,
        const void * const pvItemToQueue,
        /* 不在中断函数中触发任务切换， 而是返回一个标记 */
        BaseType_t * const pxHigherPriorityTaskWoken,
        const BaseType_t xCopyPosition )
{
    BaseType_t xReturn;
    UBaseType_t uxSavedInterruptStatus;
    Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();
    {
        // 判断队列是否有空间插入新内容
        if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) || ( xCopyPosition == queueOVERWRITE ) )
        {
            const int8_t cTxLock = pxQueue->cTxLock;

            // 中断中不能使用互斥锁， 所以拷贝函数只是拷贝数据，
            // 没有任务优先级继承需要考虑
            ( void ) prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition );


            // 判断队列是否被锁定
            if( cTxLock == queueUNLOCKED )
            {
            #if ( configUSE_QUEUE_SETS == 1 )
                // 集合相关代码
            #else /* configUSE_QUEUE_SETS */
                {
                    // 将最高优先级的等待任务恢复到就绪链表
                    if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE )
                    {
                        if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE)
                        {
                            // 如果有高优先级的任务被恢复
                            // 此处不直接触发任务切换， 而是返回一个标记
                            if( pxHigherPriorityTaskWoken != NULL )
                            {
                                *pxHigherPriorityTaskWoken = pdTRUE;
                            }
                        }
                    }
                }
```

```
44            #endif /* configUSE_QUEUE_SETS */
45            }
46            else
47            {
48                // 队列被锁定， 不能修改事件链表
49                // 增加计数， 记录需要接触几个任务到就绪
50                // 在解锁队列的时候会根据这个计数恢复任务
51                pxQueue->cTxLock = ( int8_t ) ( cTxLock + 1 );
52            }
53            xReturn = pdPASS;
54        }
55        else
56        {
57            // 队列满 直接返回 不阻塞
58            xReturn = errQUEUE_FULL;
59        }
60    }
61
62    // 恢复中断的优先级
63    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );
64
65    return xReturn;
66}
```

消息队列读取

出队方式	API 接口	实际执行函数
出队并删除队列项	xQueueReceive ()	xQueueGenericReceive ()
出队不删除队列项	xQueuePeek ()	
出队并删除队列项（带中断保护）	xQueueReceiveFromISR()	xQueueReceiveFromISR ()
出队不删除队列项（带中断保护）	xQueuePeekFromISR()	xQueuePeekFromISR()

任务调用接收函数收取队列消息， 函数首先判断当前队列是否有未读消息， 如果没有， 则会判断参数 xTicksToWait, 决定直接返回函数还是阻塞等待。 如果队列中有消息未读， 首先会把待读的消息复制到传进来的指针所指内，然后判断函数参数 xJustPeeking == pdFALSE的时候， 符合的话， 说明这个函数读取了数据， 需要把被读取的数据做出队处理， 如果不是， 则只是查看一下（peek），只是返回数据，但是不会把数据清除。 对于正常读取数据的操作， 清除数据后队列会空出空位， 所以查看队列中的等待列表中是否有任务等发送数据而被挂起，有的话恢复一个任务就绪， 并根据优先级判断是否需要出进行任务切换。 对于只是查看数据的， 由于没有清除数据， 所以没有空间新空出，不需要检查发送等待链表， 但是会检查接收等待链表， 如果有任务挂起会切换其到就绪并判断是否需要切换。

消息队列出队过程分析，其实跟入队差不多，请看注释：

```
1 /*-----------------------------------------------------------*/
2 BaseType_t xQueueGenericReceive( QueueHandle_t xQueue,                (1)
3                                  void * const pvBuffer,               (2)
```

```
 4                                   TickType_t xTicksToWait,    (3)
 5                                   const BaseType_t xJustPeeking )     (4)
 6 {
 7     BaseType_t xEntryTimeSet = pdFALSE;
 8     TimeOut_t xTimeOut;
 9     int8_t *pcOriginalReadPosition;
10     Queue_t * const pxQueue = ( Queue_t * ) xQueue;
11
12     /* 已删除一些断言 */
13     for ( ;; ) {
14         taskENTER_CRITICAL();                                           (5)
15         {
16             const UBaseType_t uxMessagesWaiting = pxQueue->uxMessagesWaiting;
17
18             /* 看看队列中有没有消息 */
19             if ( uxMessagesWaiting > ( UBaseType_t ) 0 ) {   (6)
20                 /*防止仅仅是读取消息，而不进行消息出队操作*/
21                 pcOriginalReadPosition = pxQueue->u.pcReadFrom;      (7)
22                 /* 拷贝消息到用户指定存放区域pvBuffer */
23                 prvCopyDataFromQueue( pxQueue, pvBuffer );   (8)
24
25                 if ( xJustPeeking == pdFALSE ) {             (9)
26                     /* 读取消息并且消息出队 */
27                     traceQUEUE_RECEIVE( pxQueue );
28
29                     /* 获取了消息，当前消息队列的消息个数需要减一 */
30                     pxQueue->uxMessagesWaiting = uxMessagesWaiting - 1;  (10)
31                     /* 判断一下消息队列中是否有等待发送消息的任务 */
32                     if ( listLIST_IS_EMPTY(                  (11)
33                              &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE ) {
34                         /* 将任务从阻塞中恢复 */
35                         if ( xTaskRemoveFromEventList(               (12)
36                                  &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE
) {
37                             /* 如果被恢复的任务优先级比当前任务高，会进行一次任务切
换 */
38                             queueYIELD_IF_USING_PREEMPTION();        (13)
39                         } else {
40                             mtCOVERAGE_TEST_MARKER();
41                         }
42                     } else {
43                         mtCOVERAGE_TEST_MARKER();
44                     }
45                 } else {                                         (14)
46                     /* 任务只是看一下消息（peek），并不出队 */
47                     traceQUEUE_PEEK( pxQueue );
48
49                     /* 因为是只读消息 所以还要还原读消息位置指针 */
50                     pxQueue->u.pcReadFrom = pcOriginalReadPosition; (15)
51
52                     /* 判断一下消息队列中是否还有等待获取消息的任务 */
53                     if ( listLIST_IS_EMPTY(                  (16)
54                              &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE
) {
```

```
55                          /* 将任务从阻塞中恢复 */
56                          if ( xTaskRemoveFromEventList(
57                                &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE
) {
58                              /* 如果被恢复的任务优先级比当前任务高，会进行一次任务切
换 */
59                              queueYIELD_IF_USING_PREEMPTION();
60                          } else {
61                              mtCOVERAGE_TEST_MARKER();
62                          }
63                      } else {
64                          mtCOVERAGE_TEST_MARKER();
65                      }
66                  }
67
68                  taskEXIT_CRITICAL();                            (17)
69                  return pdPASS;
70              } else {                                            (18)
71                  /* 消息队列中没有消息可读 */
72                  if ( xTicksToWait == ( TickType_t ) 0 ) {    (19)
73                      /* 不等待，直接返回 */
74                      taskEXIT_CRITICAL();
75                      traceQUEUE_RECEIVE_FAILED( pxQueue );
76                      return errQUEUE_EMPTY;
77                  } else if ( xEntryTimeSet == pdFALSE ) {
78                      /* 初始化阻塞超时结构体变量，初始化进入
79                      阻塞的时间xTickCount和溢出次数xNumOfOverflows */
80                      vTaskSetTimeOutState( &xTimeOut );               (20)
81                      xEntryTimeSet = pdTRUE;
82                  } else {
83                      mtCOVERAGE_TEST_MARKER();
84                  }
85              }
86          }
87          taskEXIT_CRITICAL();
88
89          vTaskSuspendAll();
90          prvLockQueue( pxQueue );                                (21)
91
92          /* 检查超时时间是否已经过去了*/
93          if ( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
{(22)
94              /* 如果队列还是空的 */
95              if ( prvIsQueueEmpty( pxQueue ) != pdFALSE ) {
96                  traceBLOCKING_ON_QUEUE_RECEIVE( pxQueue );   (23)
97                  /* 将当前任务添加到队列的等待接收列表中
98                     以及阻塞延时列表，阻塞时间为用户指定的超时时间xTicksToWait */
99                  vTaskPlaceOnEventList(
100                      &( pxQueue->xTasksWaitingToReceive ), xTicksToWait );
101                  prvUnlockQueue( pxQueue );
102                  if ( xTaskResumeAll() == pdFALSE ) {
103                      /* 如果有任务优先级比当前任务高，会进行一次任务切换 */
104                      portYIELD_WITHIN_API();
105                  } else {
```

```
106                     mtCOVERAGE_TEST_MARKER();
107                 }
108             } else {
109                 /* 如果队列有消息了，就再试一次获取消息 */
110                 prvUnlockQueue( pxQueue );                  (24)
111                 ( void ) xTaskResumeAll();
112             }
113         } else {
114             /* 超时时间已过，退出 */
115             prvUnlockQueue( pxQueue );                              (25)
116             ( void ) xTaskResumeAll();
117 
118             if ( prvIsQueueEmpty( pxQueue ) != pdFALSE ) {
119                 /* 如果队列还是空的，返回错误代码errQUEUE_EMPTY */
120                 traceQUEUE_RECEIVE_FAILED( pxQueue );
121                 return errQUEUE_EMPTY;                          (26)
122             } else {
123                 mtCOVERAGE_TEST_MARKER();
124             }
125         }
126     }
127 }
128 /*-----------------------------------------------------------*/
```

提示

如果队列存储的数据较大时，那最好是利用队列来传递数据的指针而不是数据本身，因为传递数据的时候是需要CPU一字节一字节地将数据拷贝进队列或从队列拷贝出来。而传递指针无论是在处理速度上还是内存空间利用上都更有效。但是，当利用队列传递指针时，一定要十分小心地做到以下两点：

1.指针指向的内存空间的所有权必须明确

当任务间通过指针共享内存时，应该从根本上保证所不会有任意两个任务同时修改共享内存中的数据，或是以其它行为方式使得共享内存数据无效或产生一致性问题。原则上，共享内存在其指针发送到队列之前，其内容只允许被发送任务访问；共享内存指针从队列中被读出之后，其内容亦只允许被接收任务访问。

2.指针指向的内存空间必须有效

如果指针指向的内存空间是动态分配的，只应该有一个任务负责对其进行内存释放。当这段内存空间被释放之后，就不应该有任何一个任务再访问这段空间。 并且最最最重要的是禁止使用指针访问任务栈上的空间，也就是局部变量。因为当栈发生改变后，栈上的数据将不再有效。

喜欢就关注我吧！

相关代码可以在公众号后台获取。