6.任务通知

2025年10月11日 16:56

- 一、任务通知的引入
  - 在前面我们学习了**信号量，队列，事件组**等，我们发现在使用他们的时候，**需要从额外的内存分配和数据结构的维护**，这在一些**资源紧张的时候或对效率要求极高的场景中，就会无法发挥其作用。**那有没有一种轻量级的通讯方案呢。
  - FreeRTOS系统为了针对这一痛点，设计了**任务通知（Task Notification）。**
- 二、任务通知的核心
  - 1、核心思想
    - 每一个任务在创建时，都会**分配一个TCB任务结构体**，在TCB中会自动分配一个32位的“通知值”，其他任务或和中断服务程序（ISR）可以**通过调用API来修改这个值，从而向该任务传递数据、信号或状态。**接受任务可以通过阻塞来等待通知的到来，也可以直接读取通知值进行处理。
  - 2、核心概念
    - **每一个FreeRTOS的任务创建时TCB结构体中都会分配一个32位通知值（Notification Value）和一个通知状态（Notification State），（标记任务是否有待处理的通知）。**
    - **任务通知的本质**：向目标任务发送一个通知值，并标记这个任务有未处理的通知。
      - 通知发送方：可以是任务或者中断服务程序（ISR）
      - 通知接收方：只能是任务（每个任务独立维护自己的通知值）
      - 通知值：32位无符号整数
- 三、任务通知的优势
  - 在上面我们引入任务通知的时候就说了，任务通知是在任务创建时候在任务结构体TCB中自动分配的，无需额外的内存资源。
  - 1.高效性
    - **无需额外创建数据结构**，直接操作任务控制块（TCB）中的通知值，减少内存开销和API的调用耗时
  - 2.灵活性
    - 支持**多种通知方式（覆盖、递增、按位操作等）**，可模拟信号，事件组等功能
  - 3.易用性
    - API简洁，适用于**简单场景（如一些任务唤醒，或者状态传递）**
  - 4.支持中断
    - 可在中断服务程序中安全使用（需要带FromISR后缀）
- 四、关键API函数
  - 1.发送通知（任务中调用）
  - **BaseType_t** xTaskNotify(**TaskHandle_t** xTaskToNotify, **uint32_t** ulValue, **eNotifyAction** eAction)

```
BaseType_t xTaskGenericNotify( TaskHandle_t xTaskToNotify,
                               UBaseType_t uxIndexToNotify,
                               uint32_t ulValue,
                               eNotifyAction eAction,
                               uint32_t * pulPreviousNotificationValue ) PRIVILEGED_FUNCTION;
#define xTaskNotify( xTaskToNotify, ulValue, eAction ) \
    xTaskGenericNotify( ( xTaskToNotify ), ( tskDEFAULT_INDEX_TO_NOTIFY ), ( ulValue ), ( eAction ), NULL )
```

    - **功能：**
      - **向指定任务发送通知**，按最后一个参数eAction的方式修改其通知值
    - **参数：**
      - 第一个参数：TaskHandle_t xTaskToNotify-->要**通知任务的任务句柄**（NULL表示当前任务）
      - 第二个参数：uint32_t ulValue-->**发送的32位值**（具体含义由eAction决定）
      - 第三个参数：eNotifyAction eAction-->**通知操作方式**
        - 此参数决定传递的数据的方式，这个参数的值是从以下枚举变量中的其中一个

```
typedef enum
```

- 此参数决定传递的数据的方式，这个参数的值是从以下枚举变量中的其中一个

```
typedef enum
{
    eNoAction = 0,              /* Notify th
    eSetBits,                   /* Set bits
    eIncrement,                 /* Incremen
    eSetValueWithOverwrite,     /* Set the v
    eSetValueWithoutOverwrite /* Set the v
} eNotifyAction;
```

- 第一个eNoAction：表示无操作，**仅唤醒任务，不修改通知值（进唤醒阻塞的任务）**
- 第二个eSetBits：将此时**任务的值与ulValue的值进行位或操作**，即设置**对应的位为1**（类似于事件组中的设置标志位）
- 第三个eIncrement：将**任务的通知值加1**，此种情况下**ulValue被忽略**。此种情况**对于简单的计数场景很有用**，例如记录某个事件的发生次数（**类似于计数信号量**）
- 第四个eSetValueWithOverwrite：将**任务的通知值设置为此函数参数中的ulValue的值**。**如果之前有通知值，则会被覆盖**
- 第五个eSetValueWithoutOverwrite：尝试将**任务的通知值设置为ulValue参数传入的值**。但**如果任务当前已有未处理的通知值（即通知值不为0），则不进行设置，函数返回pdFAIL，原通知值保持不变，即不覆盖**

操作方式（宏定义）	含义
eSetNotifyValueWithoutOverwrite	不覆盖：仅当接收任务的通知值为 0 时，才将新值写入（类似二值信号量）。
eSetNotifyValueWithOverwrite	覆盖：直接用新值覆盖接收任务的通知值（无条件更新）。
eIncrementNotifyValue	递增：将接收任务的通知值加 1（类似计数信号量）。
eSetBits	按位或：新值与通知值执行 OR 操作（类似事件组的置位）。
eClearBits	按位与：新值与通知值执行 AND 操作（用于清除特定位）。
eNoAction	无操作：仅唤醒任务，不修改通知值（仅用于唤醒阻塞的任务）。

- **返回值：**
  - 返回**pdPASS表示成功，pdFAIL表示失败**
- 了解完这个函数的大概之后，我们来深入代码底层来看一下底层代码是如何实现的

```
#if ( configUSE_TASK_NOTIFICATIONS == 1 )

    BaseType_t xTaskGenericNotify( TaskHandle_t xTaskToNotify,
                                   UBaseType_t uxIndexToNotify,
                                   uint32_t ulValue,
                                   eNotifyAction eAction,
                                   uint32_t * pulPreviousNotificationValue )
    {
        configASSERT( uxIndexToNotify < configTASK_NOTIFICATION_ARRAY_ENTRIES );
        configASSERT( xTaskToNotify );
        pxTCB = xTaskToNotify;

        taskENTER_CRITICAL();
        {
            if( pulPreviousNotificationValue != NULL )
            {
                *pulPreviousNotificationValue = pxTCB->ulNotifiedValue[ uxIndexToNotify ];
            }

            ucOriginalNotifyState = pxTCB->ucNotifyState[ uxIndexToNotify ];
```

```
            {
                *pulPreviousNotificationValue = pxTCB->ulNotifiedValue[ uxIndexToNotify ];
            }
            ucOriginalNotifyState = pxTCB->ucNotifyState[ uxIndexToNotify ];
            pxTCB->ucNotifyState[ uxIndexToNotify ] = taskNOTIFICATION_RECEIVED;

            switch( eAction )
            {
                case eSetBits:
                    pxTCB->ulNotifiedValue[ uxIndexToNotify ] |= ulValue;
                    break;

                case eIncrement:
                    ( pxTCB->ulNotifiedValue[ uxIndexToNotify ] )++;
                    break;

                case eSetValueWithOverwrite:
                    pxTCB->ulNotifiedValue[ uxIndexToNotify ] = ulValue;
                    break;

                case eSetValueWithoutOverwrite:
                    if( ucOriginalNotifyState != taskNOTIFICATION_RECEIVED )
                    {
                        pxTCB->ulNotifiedValue[ uxIndexToNotify ] = ulValue;
                    }
                    else
                    {
                        xReturn = pdFAIL;
                    }
                    break;

                case eNoAction:
                    break;

                default:
                    configASSERT( xTickCount == ( TickType_t ) 0 );
                    break;
            }

            traceTASK_NOTIFY( uxIndexToNotify );

            if( ucOriginalNotifyState == taskWAITING_NOTIFICATION )
            {
                listREMOVE_ITEM( &( pxTCB->xStateListItem ) );
                prvAddTaskToReadyList( pxTCB );
                configASSERT( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) == NULL );

                #if ( configUSE_TICKLESS_IDLE != 0 )
                {
                    prvResetNextTaskUnblockTime();
                }
                #endif

                /* Check if the notified task has a priority above the currently
                 * executing task. */
                taskYIELD_ANY_CORE_IF_USING_PREEMPTION( pxTCB );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        taskEXIT_CRITICAL();

        traceRETURN_xTaskGenericNotify( xReturn );

        return xReturn;
    }
```

函数底层代码一开始就**调用进入临界区代码**，**关中断**，以**确保函数的原子操作完整**。因为之前函数最后一个参数给了NULL，也就是无需保存当前任务的通知值，同时**将任务的状态设置为已接受状态。**然后就**根据原函数最后的参数eAction对传入的ulValue和通知值进行不同的操作**，上面已经详细阐述了eAction的五种情况。紧接着

给了NULL，也就是无需保存当前任务的通知值，同时**将任务的状态设置为已接受状态**。然后就**根据原函数最后的参数eAction对传入的ulValue和通知值进行不同的操作**，上面已经详细阐述了eAction的五种情况。紧接着下面就是**唤醒等待的任务**，**首先进行判断**，如果**初始状态是等待状态**，则将任务唤醒，**把任务从阻塞态列表转移到就绪态任务列表**。最后就是**调用退出临界区函数**，**开启中断**，恢复系统运行。

- **函数执行流程详解：**
  - **1、进入临界区保护（关中断）**

```c
taskENTER_CRITICAL();
```

    - 进入临界区，防止任务切换和中断干扰，确保操作的原子性
  - **2、状态保存和更新**
    - **ucOriginalNotifyState** = pxTCB->**ucNotifyState**[uxIndexToNotify];
      - **保存原始状态**
    - pxTCB->**ucNotifyState**[uxIndexToNotify] = **taskNOTIFICATION_RECEIVED**;
      - **将任务通知状态设置为“已接受状态”**
  - **3、根据操作类型处理通知值**
    - switch(**eAction**)
      {
        case eSetBits:
          pxTCB->ulNotifiedValue[uxIndexToNotify] |= ulValue; // 位或操作
          break;

        case eIncrement:
          (pxTCB->ulNotifiedValue[uxIndexToNotify])++; // 递增操作
          break;

        case eSetValueWithOverwrite:
          pxTCB->ulNotifiedValue[uxIndexToNotify] = ulValue; // 直接覆盖
          break;

        case eSetValueWithoutOverwrite:
          if(ucOriginalNotifyState != taskNOTIFICATION_RECEIVED)
          {
            pxTCB->**ulNotifiedValue**[uxIndexToNotify] = **ulValue**; // 非覆盖设置
          }
          else
          {
            xReturn = pdFAIL; // 如果已有通知，返回失败
          }
          break;

    a) case eNoAction:
          break; // 只发送通知，不修改值
      }
  - **4、唤醒等待任务**
    if(**ucOriginalNotifyState** == taskWAITING_NOTIFICATION)
    {
      listREMOVE_ITEM(&(pxTCB->xStateListItem));
      prvAddTaskToReadyList(pxTCB);
      configASSERT(listLIST_ITEM_CONTAINER(&(pxTCB->xEventListItem)) == NULL);
    - 首先进行**条件检查**：判断任务是否正在等待通知
    - 然后**移除阻塞状态**：将任务从阻塞状态列表移除
    - **加入就绪队列**：将任务转移至就绪态列表
    - 最后进行**事件列表检查**：确保任务不在事件列表中

- 5、**退出临界区和返回**

```
taskEXIT_CRITICAL();
traceRETURN_xTaskGenericNotify( xReturn );
return xReturn;
```

  - **退出临界区，恢复中断**
  - 记录跟踪信息
  - **返回操作结果（成功或失败）**

- 这就是这个函数底层代码的实现流程，这个函数是用于任务中发送通知，接下来我们将这种函数的一种特殊的情况，即调用函数传入特定的值

- 2.发送通知简化版（任务中调用）

BaseType_t xTaskNotifyGive(**TaskHandle_t** xTaskToNotify)

```
#define xTaskNotifyGive( xTaskToNotify ) \
    xTaskGenericNotify( ( xTaskToNotify ), ( tskDEFAULT_INDEX_TO_NOTIFY ), ( 0 ), eIncrement, NULL )
```

- 我们可以看到**函数的原型和xTaskNotify函数的原型相同底层调用的同一个函数**，只是在传参是产生了不同

```
c
#define xTaskNotifyGive( xTaskToNotify ) \
    xTaskGenericNotify( ( xTaskToNotify ), ( tskDEFAULT_INDEX_TO_NOTIFY ), ( 0 ), eIncrement, NULL )
```

```
c
// 假设 xTaskNotify 的定义（需要查看具体实现）
#define xTaskNotify(xTaskToNotify, ulValue, eAction) \
    xTaskGenericNotify((xTaskToNotify), tskDEFAULT_INDEX_TO_NOTIFY, (ulValue), (eAction), NULL)
```

- **功能：**
  - 因为此函数是**xTaskNotify的简化版**，所以该函数的功能和原函数的功能类似
  - 此函数是**向原函数指定的任务发送一个通知**，**并将通知值+1**，类似于获取计数信号量的操作
- **参数：**

  TaskHandle_t xTaskToNotify：指向**要操作的任务句柄**
- **返回值：**
  - 和原函数一样，**成功返回pdPASS，失败返回pdFAIL**
- 因为这个发送任务通知函数是**前一个任务发送通知函数的简化版**，底层**调用了相同的函数**，所以在**底层代码实现方面和之前的函数相同**
- 那么既然有发送任务通知（简化版），那么肯定有接收任务通知
- 3.接收任务通知（与xTaskNotifyGive相对应的函数）
- **BaseType_t** ulTaskNotifyTake（xClearCountOnExit, xTicksToWait）

```
uint32_t ulTaskGenericNotifyTake( UBaseType_t uxIndexToWaitOn,
                                  BaseType_t xClearCountOnExit,
                                  TickType_t xTicksToWait ) PRIVILEGED_FUNCTION;
#define ulTaskNotifyTake( xClearCountOnExit, xTicksToWait ) \
    ulTaskGenericNotifyTake( ( tskDEFAULT_INDEX_TO_NOTIFY ), ( xClearCountOnExit ), ( xTicksToWait ) )
```

- **功能：**
  - 这是**和xTaskNotifyGIve相对应的函数**，调用函数可以**根据传入xClearCountOnExit**（**pdFASLE--递减通知值（计数信号量）**，**pdTRUE--清零通知值（二值信号量）**）的值，选择性的将任务的通知值-1或者直接清零
- **返回值：**
  - **成功返回当前的通知值**，**超时返回0**，失败返回错误信息
- **参数：**
  - 我们需要传入两个参数，而原函数需要三个参数
  - UBaseType_t uxIndexToWaitOn:要**等待通知的索引值**，这里传入了默认值0
  - BaseType_t xClearCountOnExit:退出时**如何处理通知值**
    - 传入**pdTRUE**：**清零通知值**，此时效果上类似于**二值信号量**
    - 传入**pdFALSE**：**递减通知值**，效果上类似**计数信号量**
  - TickTyep_t xTicksToWait：**阻塞等待时间**

- 接下来我们来分析一下底层代码的实现流程

```
uint32_t ulTaskGenericNotifyTake( UBaseType_t uxIndexToWaitOn,
                                  BaseType_t xClearCountOnExit,
                                  TickType_t xTicksToWait )
{
    uint32_t ulReturn;
    BaseType_t xAlreadyYielded, xShouldBlock = pdFALSE;
    traceENTER_ulTaskGenericNotifyTake( uxIndexToWaitOn, xClearCountOnExit, xTicksToWait );
    configASSERT( uxIndexToWaitOn < configTASK_NOTIFICATION_ARRAY_ENTRIES );
    vTaskSuspendAll();
    {
        taskENTER_CRITICAL();
        {
            if( pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] == 0U )
            {
                pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] = taskWAITING_NOTIFICATION;
                if( xTicksToWait > ( TickType_t ) 0 )
                {
                    xShouldBlock = pdTRUE;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        taskEXIT_CRITICAL();
        {
            traceTASK_NOTIFY_TAKE_BLOCK( uxIndexToWaitOn );
            prvAddCurrentTaskToDelayedList( xTicksToWait, pdTRUE );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    xAlreadyYielded = xTaskResumeAll();
    if( ( xShouldBlock == pdTRUE ) && ( xAlreadyYielded == pdFALSE ) )
    {
        taskYIELD_WITHIN_API();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
    taskENTER_CRITICAL();
    {
        traceTASK_NOTIFY_TAKE( uxIndexToWaitOn );
        ulReturn = pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ];
        if( ulReturn != 0U )
        {
            if( xClearCountOnExit != pdFALSE )
            {
                pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] = ( uint32_t ) 0U;
            }
            else
            {
                pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] = ulReturn - ( uint32_t ) 1;
            }
        }
```

```
            {
                pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] = ulReturn - ( uint32_t ) 1;
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] = taskNOT_WAITING_NOTIFICATION;
    }
    taskEXIT_CRITICAL();

    traceRETURN_ulTaskGenericNotifyTake( ulReturn );

    return ulReturn;
}
```

返回通知值

- **函数流程实现**
  - 函数一开始**初始化各个变量**，紧接着**挂起任务调度器**，然后**调用进入临界区API函数**，**关闭系统的中断**，然后进行**条件判断**，看一下任务此时的通知值是否等于0，如果任务通知值等于0，那么此时**将任务状态改为等待任务通知的状态。**再根据阻塞等待时间是否为0来判断是进入阻塞等待还是直接返回错误。如果是**进入阻塞的话**，此时会将**任务由就绪态列表转入阻塞态列表**中。然后**退出临界区开启系统中断**，调用**恢复任务调度器**，恢复系统正常运行。然后函数**再次调用进入临界区函数**，**关闭中断**，**防止任务被打断**，确保系统的完成性。在进入临界区之后就进行**获取任务通知值**，如果任务的**通知值此时不为0**，那么**根据xClearCountOnExit的值**，**对任务通知值进行不同的操作**，选择**对通知值清零或者是递减操作。**最后退出临界区，**返回此的任务通知值**
- ulTaskNotifyTake（）函数需要与xTaskNotifyGive（）函数**配对使用**，**形成完整的信号量机制**，分别**类似于信号量的xSemaphoreTake（）和xSemaphoreGive（）函数**
- 4.等待通知（带位或操作）

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

-

```
BaseType_t xTaskGenericNotifyWait( UBaseType_t uxIndexToWaitOn,
                                   uint32_t ulBitsToClearOnEntry,
                                   uint32_t ulBitsToClearOnExit,
                                   uint32_t * pulNotificationValue,
                                   TickType_t xTicksToWait ) PRIVILEGED_FUNCTION;
#define xTaskNotifyWait( ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait ) \
    xTaskGenericNotifyWait( tskDEFAULT_INDEX_TO_NOTIFY, ( ulBitsToClearOnEntry ), ( ulBitsToClearOnExit ), ( pulNotificationValue ), ( xTicksToWait ) )
```

- **功能：**
  - 相比于上面那个等待任务通知函数来说，此函数相当于上面那个函数的加强版，此函数可以**更灵活的去等待任务通知**
- **参数：**
  - 这里函数和原函数在参数上只少了一个要等待通知的索引值，这里也是传入了默认值0
  - uint32_t ulBitsToClearOnEntry:进入**等待前要清除的位**
    - 即在检查任务状态之前，先清除任务通知值中指定的位
    - 传入0xffffffff：表示清除整个通知值
    - 传入0：表示保持通知值不变
    - 传入特定的掩码：清除指定的位
  - uint32_t ulBItsToClearOnExit：**退出等待后要清除的位**
    - 在任务获取通知值之后，清除指定的位
  - uint32_t *pulNotificationValue：**输出参数，用于返回通知值**
    - 传入NULL：不返回通知值
    - 传入非NULL：返回通知值
  - TickType_t xTicksToWait：**阻塞等待时间**
- **返回值：**
  - **成功返回pdPASS**，或者**根据pulNotificationValue来决定是否返回通知值**，**失败返回pdFALSE**
- 接下来我们深入探讨一下函数底层的代码实现

```c
BaseType_t xTaskGenericNotifyWait( UBaseType_t uxIndexToWaitOn,
                                   uint32_t ulBitsToClearOnEntry,
                                   uint32_t ulBitsToClearOnExit,
                                   uint32_t * pulNotificationValue,
                                   TickType_t xTicksToWait )
{
    BaseType_t xReturn, xAlreadyYielded, xShouldBlock = pdFALSE;
    traceENTER_xTaskGenericNotifyWait( uxIndexToWaitOn, ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait );
    configASSERT( uxIndexToWaitOn < configTASK_NOTIFICATION_ARRAY_ENTRIES );
    vTaskSuspendAll();
    {
        taskENTER_CRITICAL();
        {
            if( pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] != taskNOTIFICATION_RECEIVED )
            {
                pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] &= ~ulBitsToClearOnEntry;
                pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] = taskWAITING_NOTIFICATION;

                if( xTicksToWait > ( TickType_t ) 0 )
                {
                    xShouldBlock = pdTRUE;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        taskEXIT_CRITICAL();

        if( xShouldBlock == pdTRUE )
        {
            traceTASK_NOTIFY_WAIT_BLOCK( uxIndexToWaitOn );

            prvAddCurrentTaskToDelayedList( xTicksToWait, pdTRUE );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    xAlreadyYielded = xTaskResumeAll();

    if( ( xShouldBlock == pdTRUE ) && ( xAlreadyYielded == pdFALSE ) )
    {
        taskYIELD_WITHIN_API();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    taskENTER_CRITICAL();
    {
        traceTASK_NOTIFY_WAIT( uxIndexToWaitOn );

        if( pulNotificationValue != NULL )
        {
            *pulNotificationValue = pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ];
        }

        if( pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] != taskNOTIFICATION_RECEIVED )
        {
            xReturn = pdFALSE;
        }

        else
        {
            pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] &= ~ulBitsToClearOnExit;
            xReturn = pdTRUE;
        }
        pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] = taskNOT_WAITING_NOTIFICATION;
    }
    taskEXIT_CRITICAL();
```

```
        else
        {
            pxCurrentTCB->ulNotifiedValue[ uxIndexToWaitOn ] &= ~ulBitsToClearOnExit;
            xReturn = pdTRUE;
        }
        pxCurrentTCB->ucNotifyState[ uxIndexToWaitOn ] = taskNOT_WAITING_NOTIFICATION;
    }
    taskEXIT_CRITICAL();
    traceRETURN_xTaskGenericNotifyWait( xReturn );
    return xReturn;
}
```

- **函数执行流程：**
  - 函数一开始**先初始化各种变量**，然后**关闭任务调度器**，紧接着**进入临界区**，**关闭中断**。然后进行任务状态的判断，如果**任务状态不等于已接收状态**，就**根据传入的参数ulBitsToClearOnEntry的值来处理任务的通知值**，并**将任务状态设置为等待接收状态**。如果此时阻塞等待时间大于0，**将阻塞标志位置成pdTREU**。然后**退出临界区，开启中断**。紧接着就**把任务放入任务阻塞状态列表**。然后函数**开启任务调度**，恢复系统运行。然后**再次调用进入临界区，关闭中断**，进行下一步的原子操作。此时判断函数**第三个参数pulNotificationValue是否为NULL**，如果是，我们就用**传入的指针来接收此时的任务通知值**。然后**再次判断此时的任务状态**，如果**不是已接受状态**，函数**就返回pdFALSE**，表等待失败。**否则根据传入的ulBitsToClearOnExit的值，来确定如何处理通知值**，函数返回pdTRUE，表示等待成功。

- 以上就是任务通知所有的关键API函数了，掌握了上述函数的用法就可以掌握任务通知的精髓。

五、任务通知的使用场景

1、模拟二值信号量

- 任务通知的API中有两个函数和信号量中信号量获取和释放函数类似，所以任务通知可以用来模仿二值信号量
  - **调用ulTaskNotifyTake（）函数来等待任务通知（获取信号量），调用xTaskNotifyGive（）函数来发送任务通知（释放信号量）**。要注意，**如果要模拟二值信号量**，ulTaskNotifyTake（）的参数**xClearCountOnExit要给pdTRUE**，实现**任务通知值的清零**
  - **示例：**
    - 任务A（发送方）：**调用xTaskNotifyGive（TaskB）来向TaskB任务发送任务通知**
    - 任务B（等待方）：**调用ulTaskNotifyTake（pdTRUE，portMAX_DELAY）来等待任务通知**

2、模拟计数信号量

- 任务通知既然可以模拟二值信号量，那么计数信号量肯定也不在话下
  - 我们**调用ulTaskNotifyTake（）函数来等待任务通知（获取信号量）**，**调用xTaskNotify（）函数来发送任务通知（释放信号量）**。要注意，**如果要模拟计数信号量**，ulTaskNotifyTake（）的参数**xClearCountOnExit要给pdFASLE**，实现**任务通知值的递减**
  - **示例：**
    - 任务A（发送方）：**调用xTaskNotify（TaskB，0，eIncrement）来向TaskB任务发送任务通知**
    - 任务B（等待方）：**调用ulTaskNotifyTake（pdFALSE，portMAX_DELAY）来等待任务通知**

3、模拟事件组

- 任务通知中的**xTaskNotify（）API函数中的第三个参数中有设置标志位的操作**，这和事件组的位操作很像，所以任务通知也可以**模拟事件组**
  - 我们**调用xTaskNotifyWait（）函数来等待任务通知**，**调用xTaskNotify（）函数来发送任务通知**。
  - **示例：**
    - 任务A（发送方）：**调用xTaskNotify（TaskB，0x01，eSetBits）置Bit0位为1，表示事件1发生**
    - 任务B（等待方）：**调用xTaskNotifyWait（0，0x01，&ulValue，portMAX_DELAY）等待Bit0置位，处理后退出清零**

4、直接传递数据

- 任务**通知中的xTaskNotify（）API函数中的第三个参数中有覆盖和不覆盖通知值的操作**，我们可以**用来传递数据**
  - 发送方通过 **eSetNotifyValueWithOverwrite** 直接**将 32 位数据写入接收任务的通知值**，**接收方通过 xTaskNotifyWait 读取。**

六、注意事项

1.单接收方限制

- **每个通知只能发给一个任务。适合一对一通信**

- 2.中断安全
  - 在中断中调用函数一定是**带FromISR后缀的函数**，且**正常处理任务切换标志**
- 3.通知丢失风险
  - 若**发送方连续发送通知且接收方未及时处理，使用 eSetNotifyValueWithOverwrite 可能覆盖之前的值**（需根据场景选择操作方式）。
- 4.任务阻塞状态
  - **接收任务**在**调用 ulTaskNotifyTake 或 xTaskNotifyWait 时**，若**通知值已满足条件**（如非 0 或指定位已置位），**会立即返回，不会阻塞。**

七、总结

- **任务通知**时FreeRTOS中**轻量高效的通信机制**，通过**直接操作32位的通信值实现通信**，**适合替代简单场景下的队列，信号量，事件组。**总之，任务通知可用于解决系统在资源紧张时，各种操作的复用。