title: 从0开始学FreeRTOS-(列表与列表项)-3 top: false cover: false toc: true mathjax: false date: 2019-08-31 20:09:02 password: summary: tags:

- FreeRTOS
- RTOS
- 操作系统 categories: 操作系统

FreeRTOS列表&列表项的源码解读

第一次看列表与列表项的时候，感觉很像是链表，虽然我自己的链表也不太会，但是就是感觉很像。

在FreeRTOS中，列表与列表项使用得非常多，是FreeRTOS的一个数据结构，学习过数据结构的同学都知道，数据结构能使我们处理数据更加方便快速，能快速找到数据，在FreeRTOS中，这种列表与列表项更是必不可少的，能让我们的系统跑起来更加流畅迅速。

言归正传，FreeRTOS中使用了大量的列表（List）与列表项（Listitem），在FreeRTOS调度器中，就是用到这些来跟着任务，了解任务的状态，处于挂起、阻塞态、还是就绪态亦或者是运行态。这些信息都会在各自任务的列表中得到。

看任务控制块（tskTaskControlBlock）中的两个列表项：

```
ListItem_t xStateListItem; / * <任务的状态列表项目引用的列表表示该任务的状态（就绪，已阻止，暂停）。*/

ListItem_t xEventListItem; / * <用于从事件列表中引用任务。*/
```

一个是状态的列表项，一个是事件列表项。他们在创建任务就会被初始化，列表项的初始化是根据实际需要来初始化的，下面会说。

FreeRTOS列表&列表项的结构体

既然知道列表与列表项的重要性，那么我们来解读FreeRTOS中的list.c与list.h的源码吧。从头文件lsit.h开始，看到定义了一些结构体：

```
struct xLIST_ITEM
{
listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE / * <如果configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES设置为1，则设置为已知值。* /
configLIST_VOLATILE TickType_t xItemValue; / * <正在列出的值。在大多数情况下，这用于按降序对列表进行排序。 * /
struct xLIST_ITEM * configLIST_VOLATILE pxNext; / * <指向列表中下一个ListItem_t的指针。 * /
struct xLIST_ITEM * configLIST_VOLATILE pxPrevious; / * <指向列表中前一个ListItem_t的指针。 * /
```

```
void * pvOwner; / * <指向包含列表项目的对象（通常是TCB）的指针。因此，包含列表项目的对象与列表项目本身之间存在双向链接。 * /
void * configLIST_VOLATILE pvContainer; / * <指向此列表项目所在列表的指针（如果有）。 * /
listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE / * <如果configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES设置为1，则设置为已知值。* /
};
typedef struct xLIST_ITEM ListItem_t; / *由于某种原因，lint希望将其作为两个单独的定义。 * /
```

列表项结构体的一些注意的地方：

`xItemValue` 用于列表项的排序，类似1—2—3—4

`pxNext` 指向下一个列表项的指针

`pxPrevious` 指向上（前）一个列表项的指针

这两个指针实现了类似双向链表的功能

`pvOwner` 指向包含列表项目的对象（通常是任务控制块TCB）的指针。因此，包含列表项目的对象与列表项目本身之间存在双向链接。

`pvContainer` 记录了该列表项属于哪个列表，说白点就是这个儿子是谁生的。。。

同时定义了一个MINI的列表项的结构体，MINI列表项是删减版的列表项，因为很多时候不需要完全版的列表项。就不用浪费那么多内存空间了，这或许就是FreeRTOS是轻量级操作系统的原因吧，能省一点是一点。MINI列表项：

```
struct xMINI_LIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE           /*< Set to a known value if configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    configLIST_VOLATILE TickType_t xItemValue;
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
};
typedef struct xMINI_LIST_ITEM MiniListItem_t;
```

再定义了一个列表的结构体，可能看到这里，一些同学已经蒙了，列表与列表项是啥关系啊，按照杰杰的理解，是类似父子关系的，一个列表中，包含多个列表项，就像一个父亲，生了好多孩子，而列表就是父亲，列表项就是孩子。

```
typedef struct xLIST
{
listFIRST_LIST_INTEGRITY_CHECK_VALUE / * <如果configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES设置为1，则设置为已知值。* /
configLIST_VOLATILE UBaseType_t uxNumberOfItems;
ListItem_t * configLIST_VOLATILE pxIndex; / * <用于遍历列表。 指向由
```

```
listGET_OWNER_OF_NEXT_ENTRY（）调用返回的后一个列表项。*/
MiniListItem_t xListEnd; / * <List item包含最大可能的项目值，这意味着它始终在列表的末尾，因此用作标记。*/
listSECOND_LIST_INTEGRITY_CHECK_VALUE / * <如果configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES设置为1，则设置为已知值。* /
} List_t;
```

列表的结构体中值得注意的是： `uxNumberOfItems` 是用来记录列表中列表项的数量的，就是记录父亲有多少个儿子，当然女儿也行~。

`pxIndex` 是索引编号，用来遍历列表的，调用宏`listGET_OWNER_OF_NEXT_ENTRY（）`之后索引就会指向返回当前列表项的下一个列表项。

`xListEnd` 指向的是最后一个列表项，并且这个列表项是`MiniListItem`属性的，是一个迷你列表项。

列表的初始化

函数：

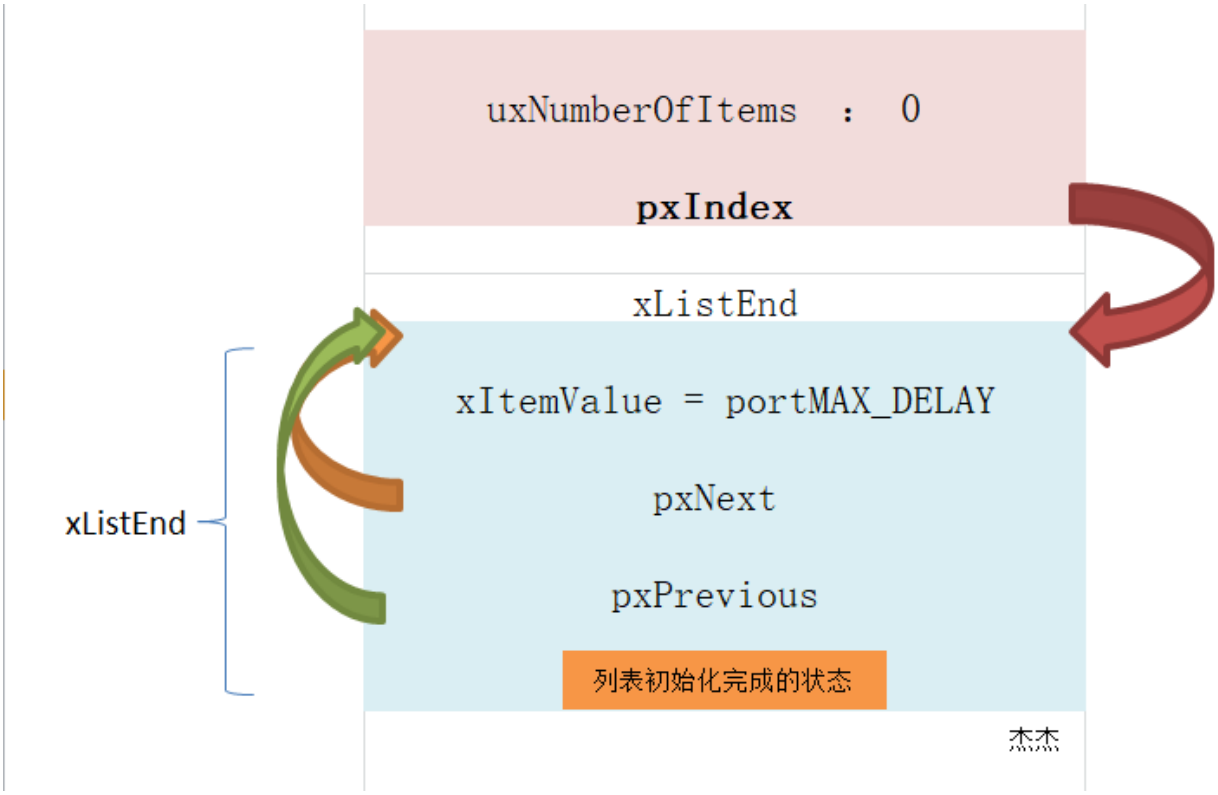
```
void vListInitialise( List_t * const pxList )
{
    pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );           /*lint The mini list structure is used as the list end to save RAM.  This is checked and valid. */
    pxList->xListEnd.xItemValue = portMAX_DELAY;
    pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );   /*lint The mini list structure is used as the list end to save RAM.  This is checked and valid. */
    pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );/*lint The mini list structure is used as the list end to save RAM.  This is checked and valid. */
    pxList->uxNumberOfItems = ( UBaseType_t ) 0U;
    listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
    listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
}
```

将列表的索引指向列表中的`xListEnd`，也就是末尾的列表项（迷你列表项）

列表项的`xItemValue`数值为`portMAX_DELAY`，也就是`0xffffffffUL`，如果在16位处理器中则为`0xffff`。

列表项的pxNext与pxPrevious这两个指针都指向自己本身`xListEnd`。

初始化完成的时候列表项的数目为`0`个。因为还没添加列表项嘛~。



列表项的初始化

函数：

```
void vListInitialiseItem( ListItem_t * const pxItem )
{
    /* Make sure the list item is not recorded as being on a list. */
    pxItem->pvContainer = NULL;
    /* Write known values into the list item if
    configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
    listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
}
```

只需要让列表项的pvContainer指针指向NULL即可，这样子就使得列表项不属于任何一个列表，因为列表项的初始化是要根据实际的情况来进行初始化的。

例如任务创建时用到的一些列表项初始化：

```
pxNewTCB->pcTaskName[ configMAX_TASK_NAME_LEN - 1 ] = '\0';
pxNewTCB->uxPriority = uxPriority;
pxNewTCB->uxBasePriority = uxPriority;
pxNewTCB->uxMutexesHeld = 0;

vListInitialiseItem( &( pxNewTCB->xStateListItem ) );
vListInitialiseItem( &( pxNewTCB->xEventListItem ) );
```

又或者是在定时器相关的初始化中：

```
pxNewTimer->pcTimerName = pcTimerName;
pxNewTimer->xTimerPeriodInTicks = xTimerPeriodInTicks;
pxNewTimer->uxAutoReload = uxAutoReload;
pxNewTimer->pvTimerID = pvTimerID;
pxNewTimer->pxCallbackFunction = pxCallbackFunction;

vListInitialiseItem( &( pxNewTimer->xTimerListItem ) );
```

列表项的末尾插入

函数：

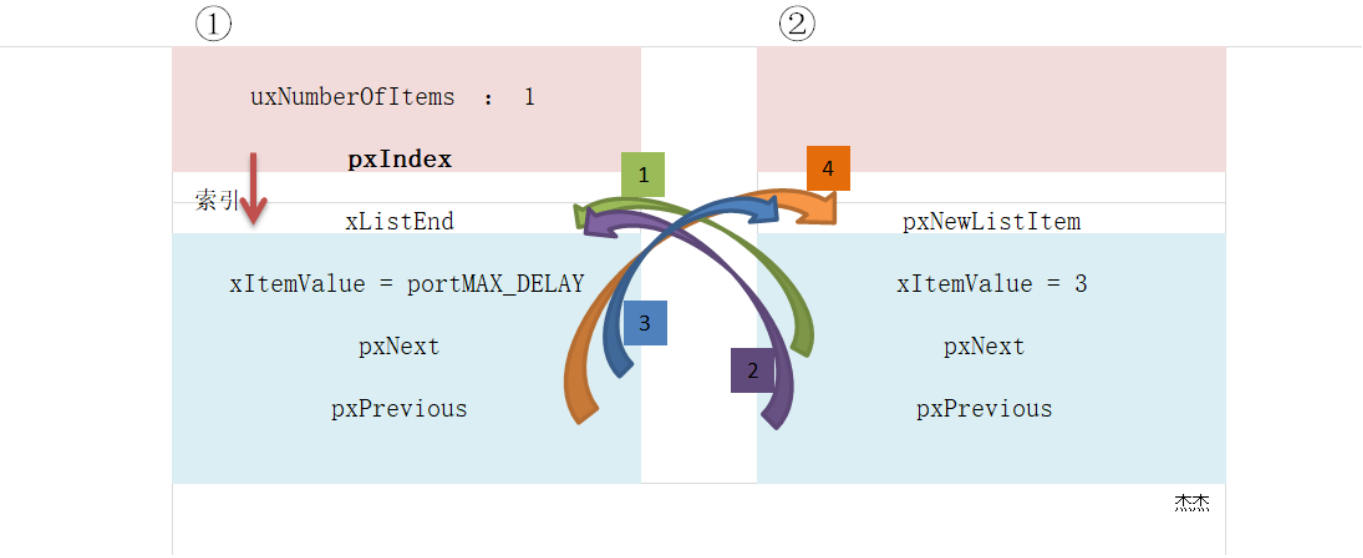
```
void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
{
    ListItem_t * const pxIndex = pxList->pxIndex;
    listTEST_LIST_INTEGRITY( pxList );
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
    listGET_OWNER_OF_NEXT_ENTRY(). */
    pxNewListItem->pxNext = pxIndex;    //  1
    pxNewListItem->pxPrevious = pxIndex->pxPrevious;    //  2
    /* Only used during decision coverage testing. */
    mtCOVERAGE_TEST_DELAY();
    pxIndex->pxPrevious->pxNext = pxNewListItem;        //  3
    pxIndex->pxPrevious = pxNewListItem;                //  4
    /* Remember which list the item is in. */
    pxNewListItem->pvContainer = ( void * ) pxList;
    ( pxList->uxNumberOfItems )++;
}
```

传入的参数：

`pxList`：列表项要插入的列表。

`pxNewListItem`：要插入的列表项是什么。

从末尾插入，那就要先知道哪里是头咯，我们在列表中的成员`pxIndex`就是用来遍历列表项的啊，那它指向的地方就是列表项的头，那么既然FreeRTOS中的列表很像数据结构中的双向链表，那么，我们可以把它看成一个环，是首尾相连的，那么函数中说的末尾，就是列表项头的前一个，很显然其结构图应该是下图这样子的（初始化结束后`pxIndex`指向了`xListEnd`）：



为什么是这样子的呢，一句句代码来解释：

一开始：

```
ListItem_t * const pxIndex = pxList->pxIndex;
```

保存了一开始的索引列表项（xListEnd）的指向。

```
pxNewListItem->pxNext = pxIndex;         //  1
```

新列表项的下一个指向为索引列表项，也就是绿色的箭头。

```
pxNewListItem->pxPrevious = pxIndex->pxPrevious;     //  2
```

刚开始我们初始化完成的时候pxIndex->pxPrevious的指向为自己xListEnd，那么xNewListItem->pxPrevious的指向为xListEnd。如2紫色的箭头。

```
pxIndex->pxPrevious->pxNext = pxNewListItem;         //  3
```

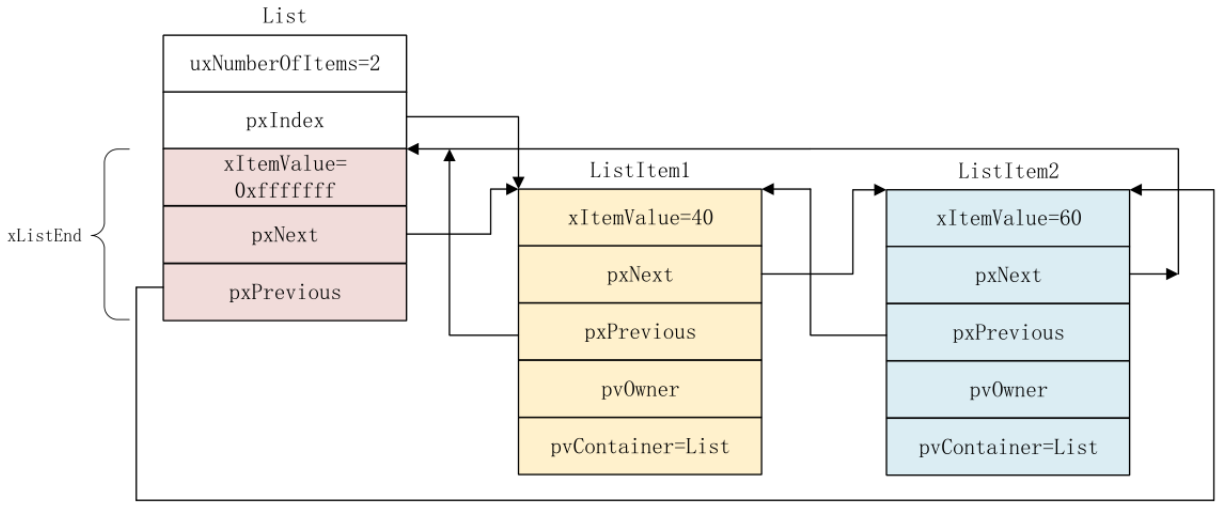
索引列表项（xListEnd）的上一个列表项还是自己，那么自己的下一个列表项指向就是指向了pxNewListItem。

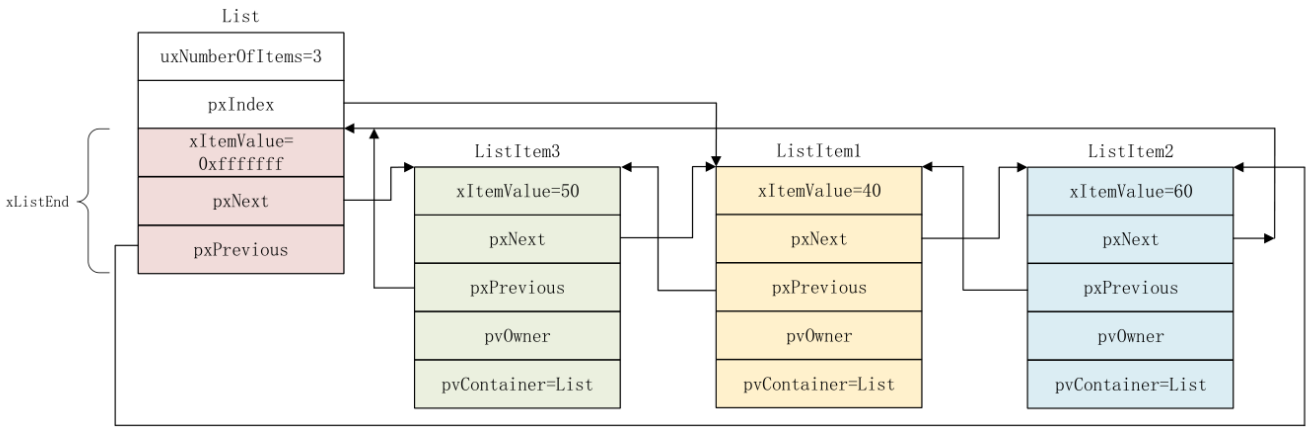
```
pxIndex->pxPrevious = pxNewListItem;             //  4
```

这句就很容易理解啦。如图的4橙色的箭头。

插入完毕的时候标记一下新的列表项插入了哪个列表，并且将uxNumberOfItems进行加一，以表示多了一个列表项。

为什么源码要这样子写呢？因为这只是两个列表项，一个列表含有多个列表项，那么这段代码的通用性就很强了。无论原本列表中有多少个列表项，也无论pxIndex指向哪个列表项！





看看是不是按照源码中那样插入呢？

列表项的插入

源码：

```
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
{
ListItem_t *pxIterator;
const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
    listTEST_LIST_INTEGRITY( pxList );
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
     if( xValueOfInsertion == portMAX_DELAY )
   {
        pxIterator = pxList->xListEnd.pxPrevious;
    }
    else
    {
```

```
        for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator-
>pxNext->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext ) /*lint
!e826 !e740 The mini list structure is used as the list end to save RAM.  This is
checked and valid. */
        {
            /* There is nothing to do here, just iterating to the wanted
            insertion position. */
        }
    }
    pxNewListItem->pxNext = pxIterator->pxNext;
    pxNewListItem->pxNext->pxPrevious = pxNewListItem;
    pxNewListItem->pxPrevious = pxIterator;
    pxIterator->pxNext = pxNewListItem;
    /* Remember which list the item is in.  This allows fast removal of the
    item later. */
    pxNewListItem->pvContainer = ( void * ) pxList;
    ( pxList->uxNumberOfItems )++;
}
```

传入的参数：

`pxList`：列表项要插入的列表。 `pxNewListItem`：要插入的列表项是什么。

pxList决定了插入哪个列表，`pxNewListItem`中的`xItemValue`值决定了列表项插入列表的位置。

```
ListItem_t *pxIterator;
const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
```

定义一个辅助的列表项pxIterator，用来迭代找出插入新列表项的位置，并且保存获取要插入的列表项`pxNewListItem`的xItemValue。

如果打开了列表项完整性检查，就要用户实现`configASSERT()`，源码中有说明。

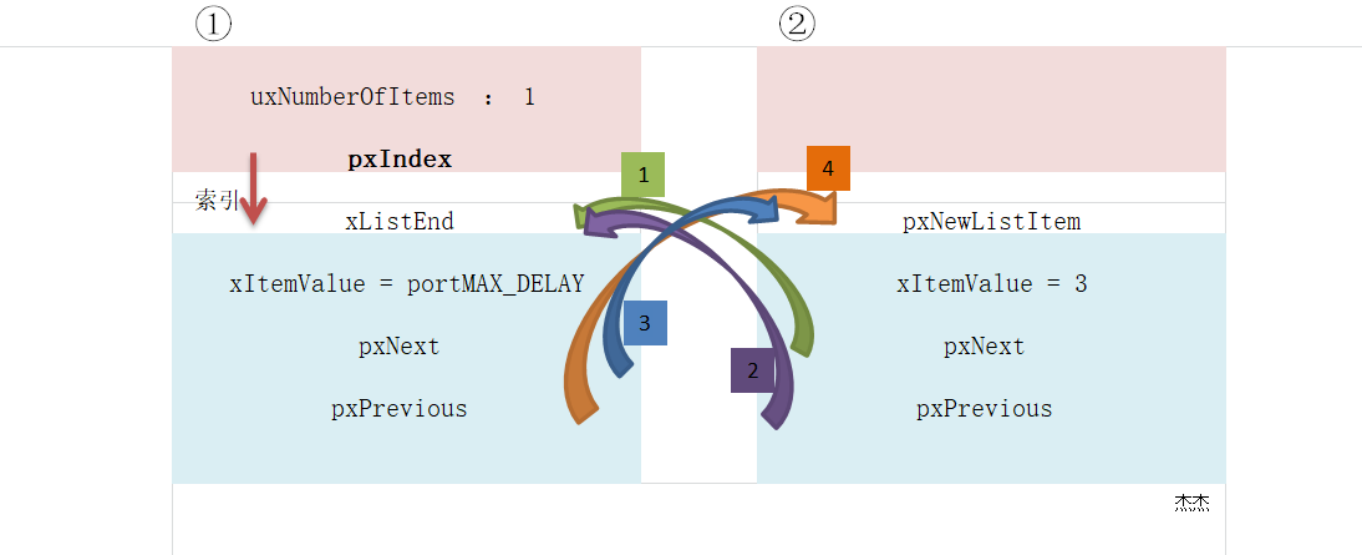
既然是要插入列表项，那么肯定是要知道列表项的位置了，如果新插入列表项的`xItemValue`是最大的话（`portMAX_DELAY`），就直接插入列表项的末尾。否则就需要比较列表中各个列表项的`xItemValue`的大小来进行排列。然后得出新列表项插入的位置。

```
for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext-
>xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
```

上面源码就是实现比较的过程。

与上面的从列表项末尾插入的源码一样，FreeRTOS的代码通用性很强，逻辑思维也很强。

如果列表中列表项的数量为0，那么插入的列表项就是在初始化列表项的后面。如下图所示：



过程分析：

新列表项的pxNext指向pxIterator->pxNext，也就是指向了xListEnd（pxIterator）。

```
pxNewListItem->pxNext = pxIterator->pxNext;
```

而xListEnd（pxIterator）的pxPrevious指向则为pxNewListItem。

```
pxNewListItem->pxNext->pxPrevious = pxNewListItem;
```

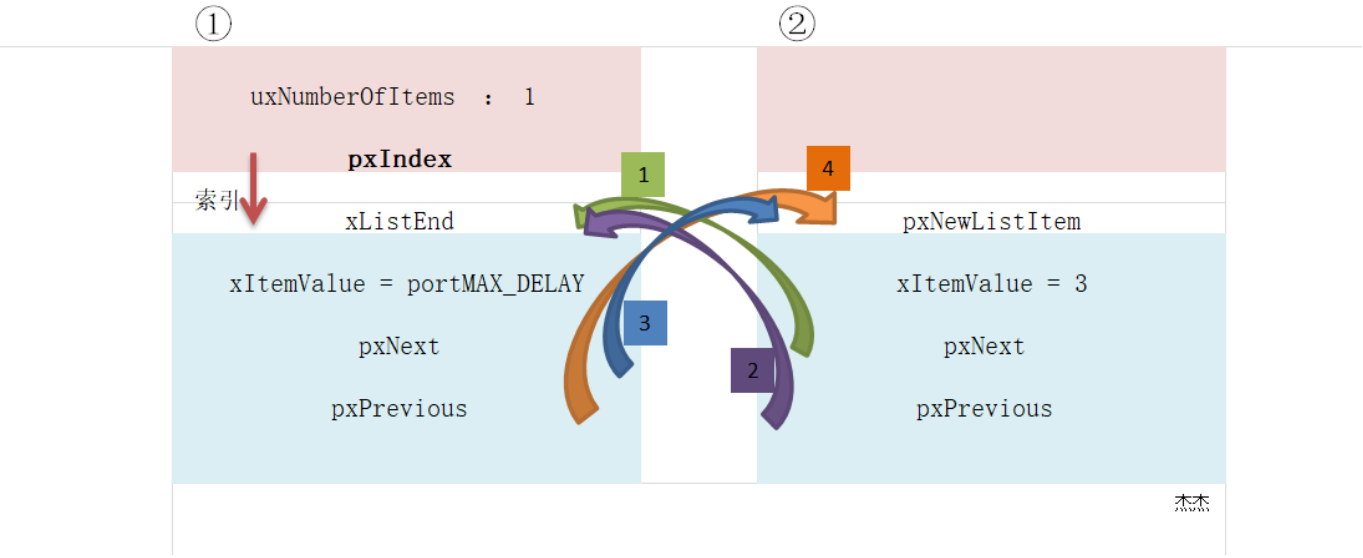
新列表项的（pxPrevious）指针指向xListEnd（pxIterator）

pxIterator 的 pxNext 指向了新列表项

```
pxNewListItem->pxPrevious = pxIterator;
pxIterator->pxNext = pxNewListItem;
```

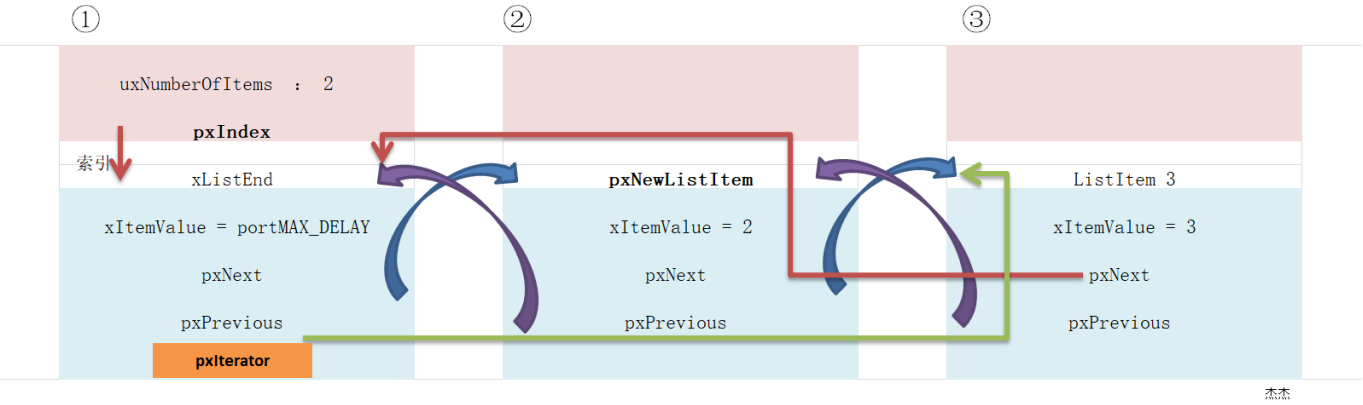
与从末尾插入列表项其实是一样的，前提是当前列表中列表项的数目为0。

假如列表项中已经有了元素呢，过程又是不一样的了。原来的列表是下图这样子的：



假设插入的列表项的xItemValue是2，而原有的列表项的xItemValue值是3，那么，按照源码，我们插入的列表项是在中间了。而pxIterator则是①号列表项。

插入后的效果：



分析一下插入的过程：

新的列表项的pxNext指向的是pxIterator->pxNext，也就是③号列表项。因为一开始pxIterator->pxNext=指向的就是③号列表项！！

```
pxNewListItem->pxNext = pxIterator->pxNext;
```

而pxNewListItem->pxNext 即③号列表项的指向上一个列表项指针（pxPrevious）的则指向新插入的列表项，也就是②号列表项了。

```
pxNewListItem->pxNext->pxPrevious = pxNewListItem;
```

新插入列表项的指向上一个列表项的指针pxNewListItem->pxPrevious指向了辅助列表项pxIterator。很显然要连接起来嘛！

```
pxNewListItem->pxPrevious = pxIterator;
```

同理，pxIterator列表项的指向下一个列表项的指针则指向新插入的列表项了pxNewListItem。

```
pxIterator->pxNext = pxNewListItem;
```

而其他没改变指向的地方不需改动。（图中的两条直线做的连接线是不需要改动的）

当插入完成的时候，记录一下新插入的列表项属于哪个列表。并且让该列表下的列表项数目加一。

```
pxNewListItem->pvContainer = ( void * ) pxList;
         ( pxList->uxNumberOfItems )++;
```

删除列表项

源码：

```
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
{
/* The list item knows which list it is in.  Obtain the list from the list
item. */
List_t * const pxList = ( List_t * ) pxItemToRemove->pvContainer;
    pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
    pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
    /* Only used during decision coverage testing. */
    mtCOVERAGE_TEST_DELAY();
    /* Make sure the index is left pointing to a valid item. */
    if( pxList->pxIndex == pxItemToRemove )
    {
        pxList->pxIndex = pxItemToRemove->pxPrevious;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
    pxItemToRemove->pvContainer = NULL;
    ( pxList->uxNumberOfItems )--;
    return pxList->uxNumberOfItems;
}
```

其实删除是很简单的，不用想都知道，要删除列表项，那肯定要知道该列表项是属于哪个列表吧，pvContainer就是记录列表项是属于哪个列表的。

删除就是把列表中的列表项从列表中去掉，其本质其实就是把他们的连接关系删除掉，然后让删除的列表项的前后两个列表连接起来就行了，假如是只有一个列表项，那么删除之后，列表就回到了初始化的状态了。

```
pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
```

这两句代码就实现了将删除列表项的前后两个列表项连接起来。

按照上面的讲解可以理解这两句简单的代码啦。

假如删除的列表项是当前索引的列表项，那么在删除之后，列表中的pxIndex就要指向删除列表项的上一个列表项了。

```
if( pxList->pxIndex == pxItemToRemove )
{
     pxList->pxIndex = pxItemToRemove->pxPrevious;
}
```

当然还要把当前删除的列表项的pvContainer指向NULL，让它不属于任何一个列表，因为，删除的本质是删除的仅仅是列表项的连接关系，其内存是没有释放掉的，假如是动态内存分配的话。

并且要把当前列表中列表项的数目返回一下。

至此，列表的源码基本讲解完毕。

最后

大家还可以了解一下遍历列表的宏，它在list.h文件中：

```
define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )
        \
{
          \
List_t * const pxConstList = ( pxList );
        \
    /* Increment the index to the next item and return the item, ensuring */
        \
    /* we don't return the marker used at the end of the list.  */
        \
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
        \
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )-
>xListEnd ) )  \
    {
         \
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
          \
```

```
    }
          \
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;
         \
}
```

这是一个宏，用于列表的遍历，返回的是列表中列表项的`pxOwner`成员，每次调用这个宏（函数）的时候，其pxIndex索引会指向当前返回列表项的下一个列表项。

喜欢就关注我吧！

相关代码可以在公众号后台获取。