FreeRTOS 源码解读

Nrush

2019 年 10 月 22 日

目录

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

```
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

```
#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) );    \
})
```

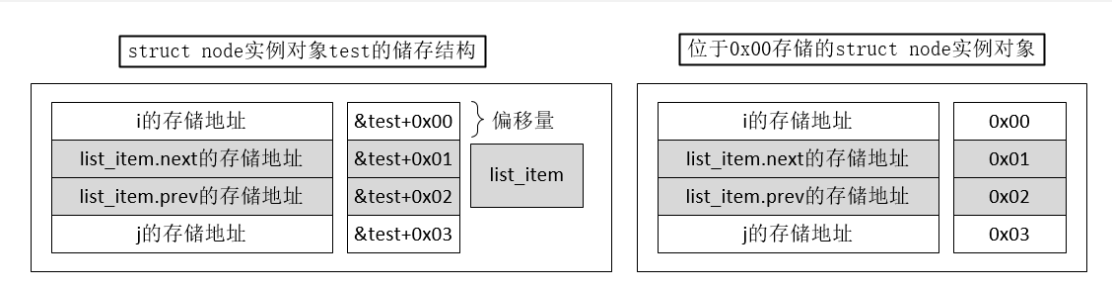
各输入参数的含义为

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

使用示例

```
// 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

```
(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

```
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;
```

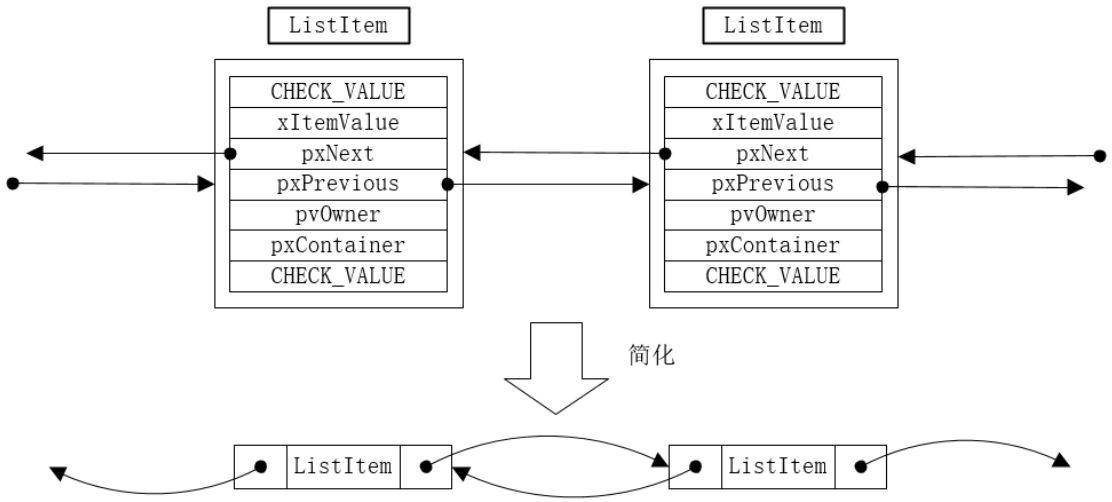
Listing 1.3: MiniListItem_t

链表 List_t 的定义

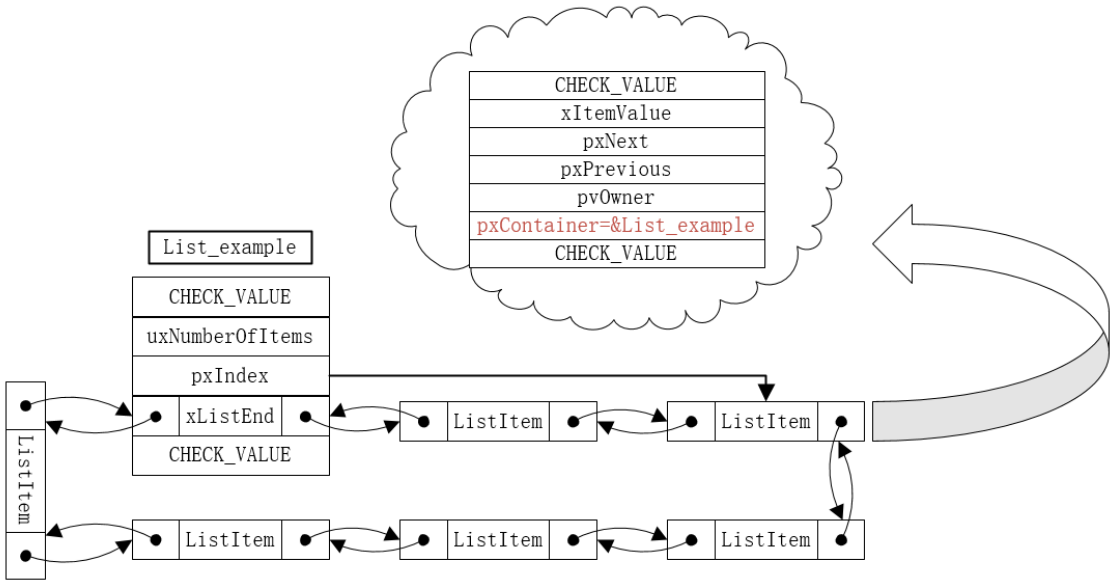
```
1  typedef struct xLIST
2  {
3      // 校验值
4      listFIRST_LIST_INTEGRITY_CHECK_VALUE
5      // 记录该链表里有多少成员
6      volatile UBaseType_t uxNumberOfItems;
7      // 用来遍历链表的指针，listGET_OWNER_OF_NEXT_ENTRY()会使它指向下一个链表项
8      ListItem_t * configLIST_VOLATILE pxIndex;
9      // 链表尾部，为节省内存使用Mini链表项
10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