阿熊的FreeRTOS教程系列！

哈喽大家好！我是你们的老朋友阿熊！STM32教程系列更新完结已经有一段时间了，视频反馈还是不错的，从今天开始我们将会更新我们的FreeRTOS的教程

由于东西真的太多了，也纠结了很久要不要讲这个系列，毕竟难度真的很大，怕在难以做到那么通俗易懂，经过一段时间的考虑，还是决定好了给大家做一个入门级的讲解使用，由于FreeRTOS的内容真的很多，作为还是学生的我使用的也相对较少，操作系统层面的东西，我会用最大的能力去让大家理解，主要讲述主要功能，学完以后保证大伙可以理解80%以上的FreeRTOS的使用场景，好了废话不多说，开始我们的课程吧！

第六章：消息队列**(queue)**

队列(queue)可以用于"任务到任务"、"任务到中断"、"中断到任务"直接传输信息，可以实现任务之间的通信

壹：消息队列的介绍

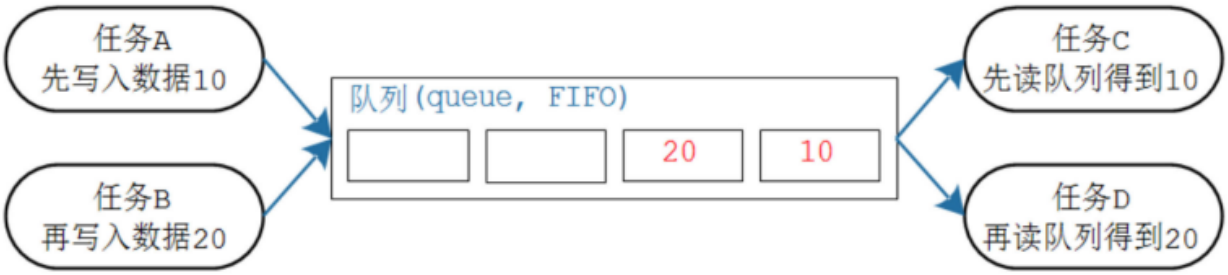
消息队列就是一个像容器一样的东西，我们所有的任务都可以往内部写消息，然后队列会将我们的消息按顺序存下来，所有的任务也可以按顺序将其读出来

需要注意的几点：

队列需要明确数据的大小以及队列的长度

写队列和读队列都是采用复制的方式将数据复制过去使用

数据的操作默认采用先进先出的方法**(FIFO，First In First Out)**：写数据时放到尾部，读数据时从头部读



也存在后进先出的模式**(LIFO),**不过使用较少

存在函数可以强制写队列头部：覆盖头部数据

这里都是一些简单的概述，让大伙对我们的消息队列有一个基本的概念，然后我们现在开始教大家如何进行创建队列以及我们队列的基本使用

贰：消息队列的最基本操作

队列的创建：

动态创建

```
QueueHandle_t    xQueueCreate(UBaseType_t     uxQueueLength,
UBaseType_t  uxItemSize);
//传入参数（队列长度，每个数据的大小：以字节为单位）
//返回  非零：创建成功返回消息队列的句柄     NULL：创建失败
```

静态创建

```
QueueHandle_t   xQueueCreateStatic(UBaseType_t uxQueueLength,
                                   UBaseType_t uxItemSize,
                                   uint8_t *pucQueueStorageBuffer,
                                   StaticQueue_t *pxQueueBuffer);
```

队列复位

队列刚被创建时，里面没有数据；使用过程中可以调用 xQueueReset()把队列恢复为初始状态

```
BaseType_t  xQueueReset( QueueHandle_t  pxQueue);//传入队列句柄即可
```

删除队列

删除队列的函数为 vQueueDelete()，只能删除使用动态方法创建的队列，它会释放内存

```
void    vQueueDelete( QueueHandle_t xQueue );//传入队列句柄即可
```

写队列

可以把数据写到队列头部，也可以写到尾部，这些函数有两个版本：在任务中使用、在 ISR（中断） 中使用

往后写入

```
BaseType_t xQueueSend(QueueHandle_t xQueue,
                      const void *pvItemToQueue,
                      TickType_t xTicksToWait);
//参数:
//      需要写的队列、写入数据的指针、等待时间
//      往队列尾部写入数据，如果没有空间，阻塞时间为 xTicksToWait
//（如果被设为 0，无法写入数据时函数会立刻返回；如果被设为 portMAX_DELAY，则会一直阻塞直到有数据可写）
```

```
BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
                            const void *pvItemToQueue,
                            TickType_t xTicksToWait);
//参数:
//        需要写的队列、写入数据的指针、等待时间
//        往队列尾部写入数据，如果没有空间，阻塞时间为 xTicksToWait
//（如果被设为 0，无法写入数据时函数会立刻返回；如果被设为 portMAX_DELAY，则会一直阻塞直到有数据可写）
```

这两个默认是等效的，一般使用前面的

往前写入

```
BaseType_t  xQueueSendToFront(QueueHandle_t xQueue,
                              const void  *pvItemToQueue,
                              TickType_t  xTicksToWait);
```

中断中的函数

中断写入

```
BaseType_t  xQueueSendToBackFromISR(QueueHandle_t xQueue,
                                    const void *pvItemToQueue,
                                    BaseType_t
*pxHigherPriorityTaskWoken);
BaseType_t  xQueueSendToFrontFromISR(QueueHandle_t xQueue,
                                    const void *pvItemToQueue,
                                    BaseType_t
*pxHigherPriorityTaskWoken);
//对某个队列而言，可能有不止一个任务处于阻塞态在等待其数据有效。调用xQueueSendToFrontFromISR()或 xQueueSendToBackFromISR()会使得队列数据变为有效，所以会让其中一个等待任务切出阻塞态。如果调用这两个 API 函数使得一个任务解除阻塞，并且这个任务的优先级高于当前任务(也就是被中断的任务)，那么 API 会在函数内部将*pxHigherPriorityTaskWoken 设为 pdTRUE。如果这两个 API 函数将此值设为 pdTRUE，则在中断退出前应当进行一次上下文切换。这样才能保证中断直接返回到就绪态任务中优先级最高的任务中。
```

读队列

普通读

```
BaseType_t  xQueueReceive( QueueHandle_t xQueue,
                           void * const pvBuffer,
                           TickType_t xTicksToWait );
//参数
//        队列句柄、数据存放地指针、等待时间
//（如果被设为 0，无法读出数据时函数会立刻返回；如果被设为 portMAX_DELAY，则会一直阻塞直到有数据可读）
```

中断读

```
BaseType_t  xQueueReceiveFromISR(QueueHandle_t xQueue,
                                 void *pvBuffer,
                                 BaseType_t *pxTaskWoken);
```

队列查询

查询队列可用数据个数

```
UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
//返回队列中可用数据的个数
```

查询队列可用空间个数

```
UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
//返回队列中可用空间的个数
```

队列覆盖（只有长度是一才可用）

普通覆盖

```
BaseType_t xQueueOverwrite(QueueHandle_t xQueue,
                           const void * pvItemToQueue);
//xQueue: 写哪个队列
//pvItemToQueue: 数据地址
//返回值: pdTRUE 表示成功, pdFALSE 表示失败
```

中断覆盖

```
BaseType_t  xQueueOverwriteFromISR(QueueHandle_t xQueue,
                                   const void * pvItemToQueue,
                                   BaseType_t
*pxHigherPriorityTaskWoken
 );
```

队列数据偷窥

正常情况下我们读取完队列中的数据那个数据就会被移除掉，也就是说他只能被读取一次，这时候就要用到我们的偷窥的函数了

普通偷窥

```
BaseType_t xQueuePeek(QueueHandle_t xQueue,
                      void * const pvBuffer,
                      TickType_t xTicksToWait);
//xQueue: 偷看哪个队列
//pvBuffer: 数据地址，用来保存复制出来的数据
//xTicksToWait: 没有数据的话阻塞一会
//返回值: pdTRUE 表示成功，pdFALSE 表示失败
```

中断偷窥

```
BaseType_t  xQueuePeekFromISR(QueueHandle_t xQueue,
                              void *pvBuffer,);
```

叁：队列的基本使用

实验一：创建一个消息队列（使用**Cube MX**创建），两个发送任务，一个接收任务

发送任务一设置等待时间为0，发送任务二设置等待时间为最大（portMAX_DELAY），接受任务等待时间为0，然后观察实验现象

现象：

队列满了以后，发送任务一无法正常发送，任务二将会死等待，队列空闲以后将会完成发送

分析：

我们的发送任务二在队列满了之后将会进入等待的状态，也就是阻塞状态，当任务有空闲的位置，它就会立刻补上空缺位置，然后结束我们的阻塞状态

实验二：实验一基础上修改接收任务等待时间（手动创建队列）

将接收任务等待时间改为最大（portMAX_DELAY），观察实验现象

现象：

队列满了以后，任务一无法发送任务，任务二进入死等状态，然后我们的接收任务可以连续接收到两个数据

分析：

当队列中的数据全部被接收任务取走之后，我们再次按一下按键，我们的接收任务因为无法接收到数据将会进入阻塞状态，并等到我们的队列中有数据，然后才会读取数据结束阻塞状态

实验三：覆盖和偷看实验

创建一个长度为一的队列，然后发送任务二发送函数改为覆盖函数，接收函数改为偷窥函数

现象：

按一下我们第2个按键，我们的队列内容被覆盖，并且无论我们怎样偷窥我们的数据队列中的数据都不会被清除

分析：

我们成功覆盖并且偷看到了我们的队列内容，不过这里要注意，我们有一个前提就是必须队列长度为一，并且我们是覆盖原来的数据，所以它不存在有阻塞的情况