title: STM32进阶之串口环形缓冲区实现 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-05 22:21:12 img: coverImg: password: summary: tags:

- Cortex-M
- STM32
- fifo categories: STM32

队列的概念

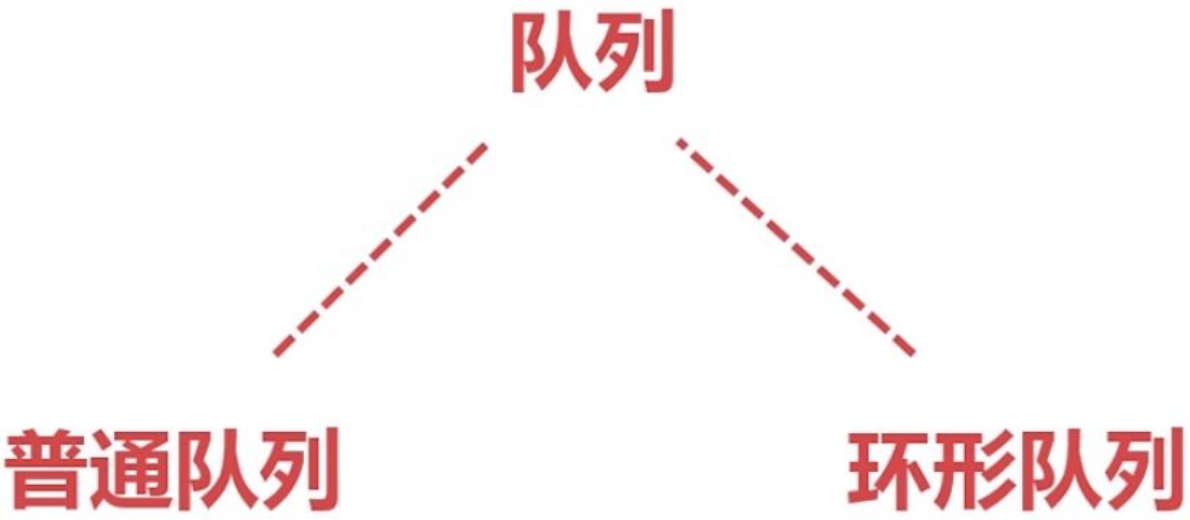
在此之前，我们来回顾一下队列的基本概念：

队列 (Queue)：是一种先进先出(First In First Out ,简称 FIFO)的线性表，只允许在一端插入（入队），在另一端进行删除（出队）。

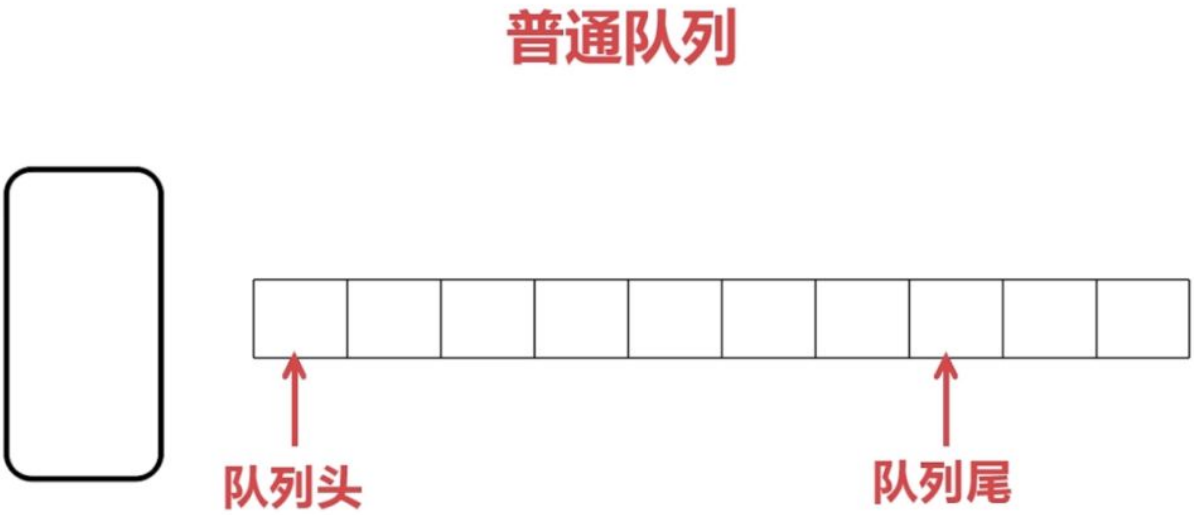
队列的特点

类似售票排队窗口，先到的人看到能先买到票，然后先走，后来的人只能后买到票

队列的常见两种形式



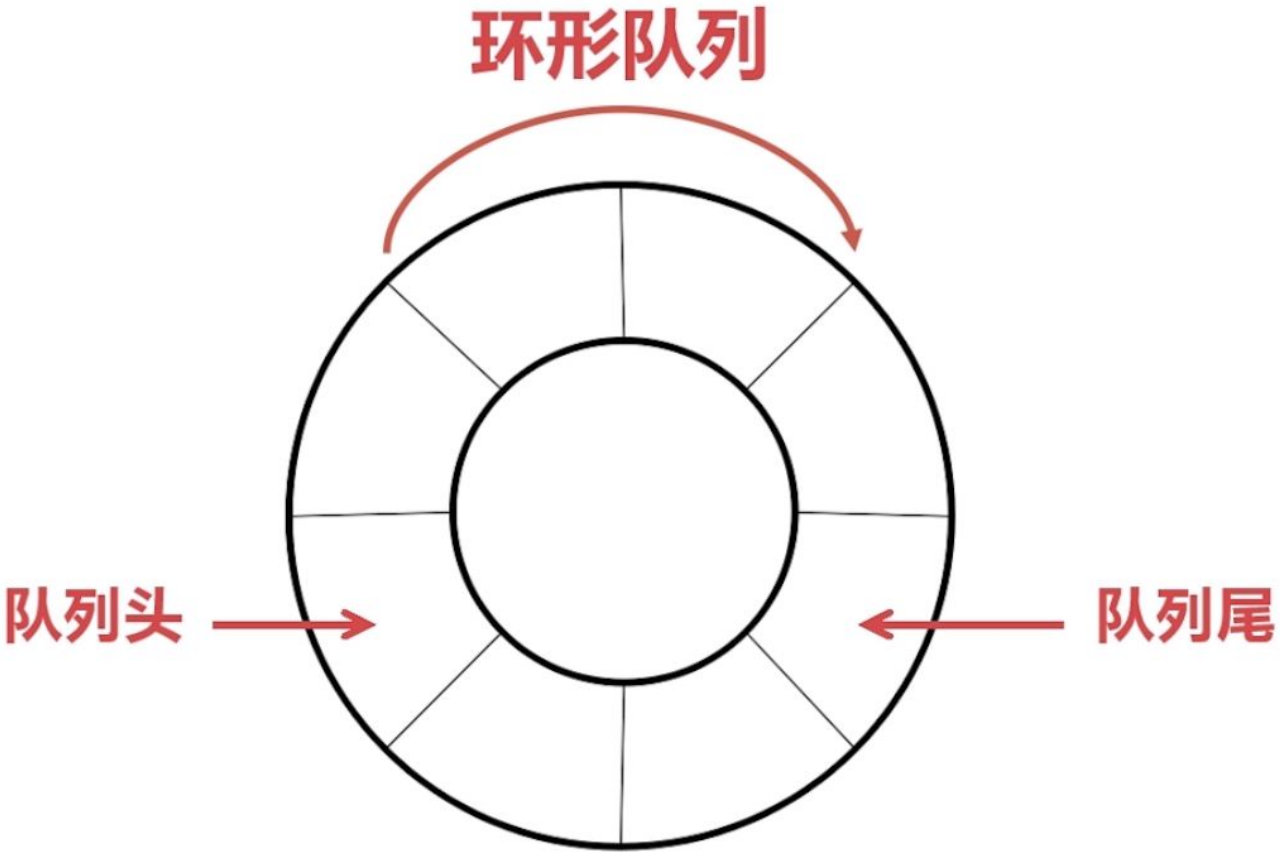
普通队列



在计算机中，每个信息都是存储在存储单元中的，比喻一下吧，上图的一些小正方形格子就是一个个存储单元，你可以理解为常见的数组，存放我们一个个的信息。

当有大量数据的时候，我们不能存储所有的数据，那么计算机处理数据的时候，只能先处理先来的，那么处理完后呢，就会把数据释放掉，再处理下一个。那么，已经处理的数据的内存就会被浪费掉。因为后来的数据只能往后排队，如过要将剩余的数据都往前移动一次，那么效率就会低下了，肯定不现实，所以，环形队列就出现了。

环形队列

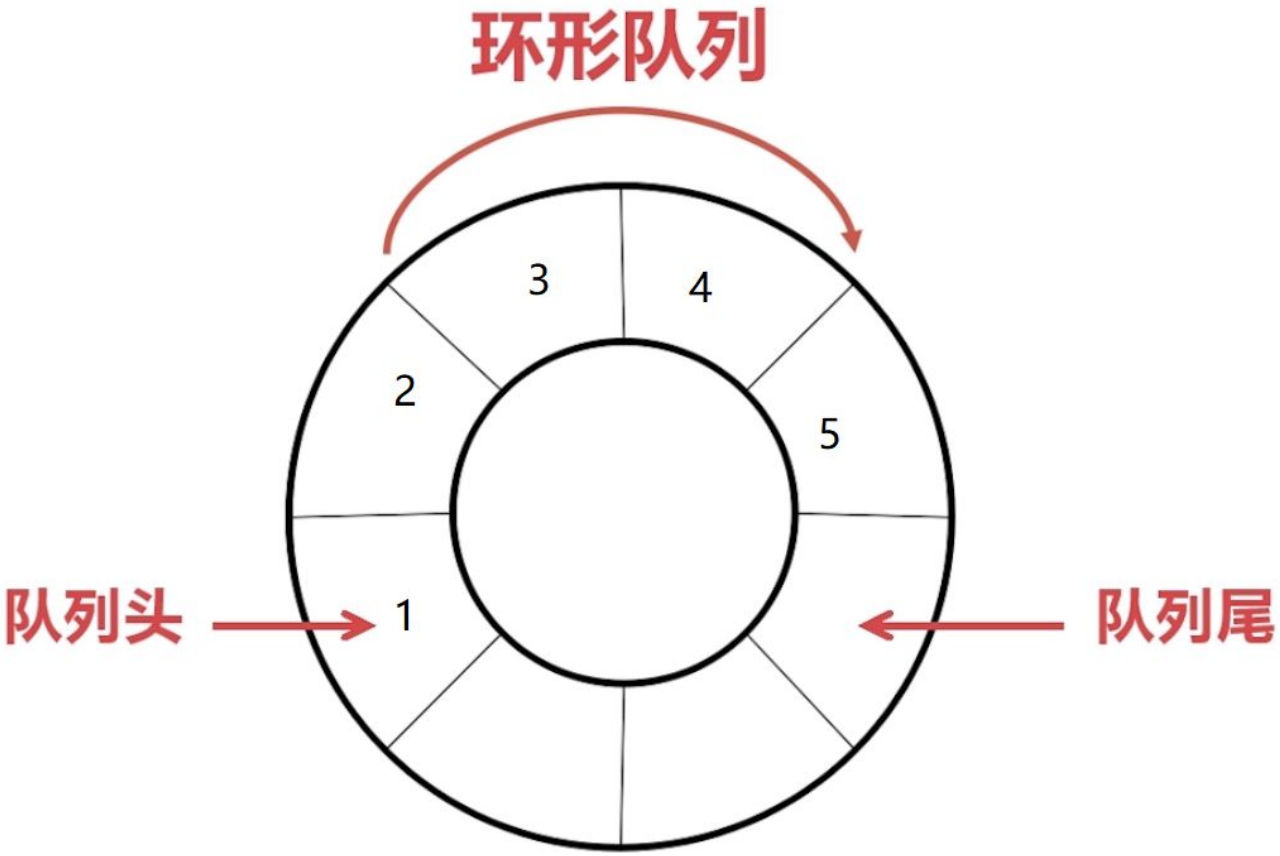


它的队列就是一个环，它避免了普通队列的缺点，就是有点难理解而已，其实它就是一个队列，一样有队列头，队列尾，一样是先进先出（FIFO）。我们采用顺时针的方式来对队列进行排序。

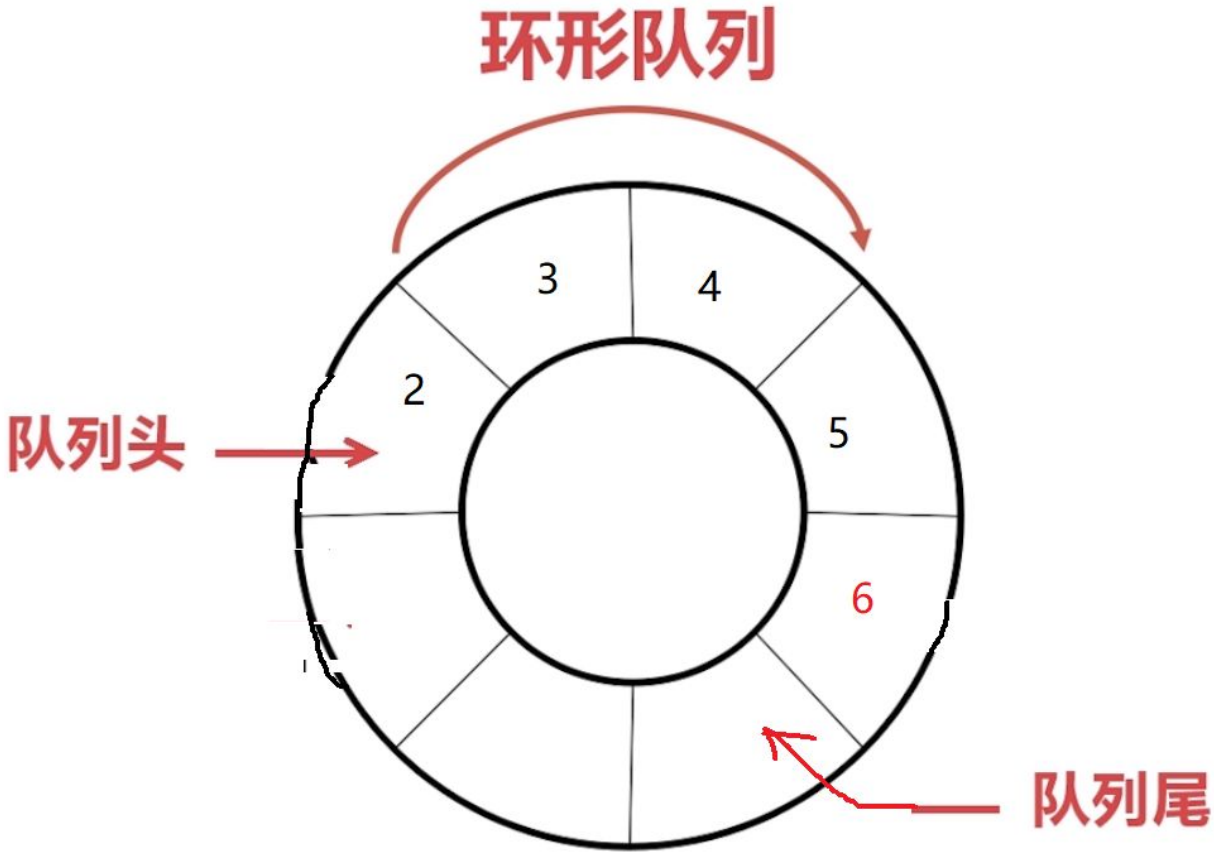
- 队列头 (Head) : 允许进行删除的一端称为队首。
- 队列尾 (Tail) : 允许进行插入的一端称为队尾。

环形队列的实现：在计算机中，也是没有环形的内存的，只不过是我们将顺序的内存处理过，让某一段内存形成环形，使他们首尾相连，简单来说，这其实就是一个数组，只不过有两个指针，一个指向列队头，一个指向列队尾。指向列队头的指针(Head)是缓冲区可读的数据，指向列队尾的指针(Tail)是缓冲区可写的数据，通过移动这两个指针(Head) &(Tail)即可对缓冲区的数据进行读写操作了，直到缓冲区已满（头尾相接），将数据处理完，可以释放掉数据，又可以进行存储新的数据了。

实现的原理：初始化的时候，列队头与列队尾都指向0，当有数据存储的时候，数据存储在‘0’的地址空间，列队尾指向下一个可以存储数据的地方‘1’，再有数据来的时候，存储数据到地址‘1’，然后队列尾指向下一个地址‘2’。当数据要进行处理的时候，肯定是先处理‘0’空间的数据，也就是列队头的数据，处理完了数据，‘0’地址空间的数据进行释放掉，列队头指向下一个可以处理数据的地址‘1’。从而实现整个环形缓冲区的数据读写。



看图，队列头就是指向已经存储的数据，并且这个数据是待处理的。下一个CPU处理的数据就是1；而队列尾则指向可以进行写数据的地址。当1处理了，就会把1释放掉。并且把队列头指向2。当写入了一个数据6，那么队列尾的指针就会指向下一个可以写的地址。



从队列到串口缓冲区的实现

串口环形缓冲区收发：在很多入门级教程中，我们知道的串口收发都是：接收一个数据，触发中断，然后把数据发回来。这种处理方式是没有缓冲的，当数量太大的时候，亦或者当数据接收太快的时候，我们来不及处理已经收到的数据，那么，当再次收到数据的时候，就会将之前还未处理的数据覆盖掉。那么就会出现丢包的现象了，对我们的程序是一个致命的创伤。

那么如何避免这种情况的发生呢，很显然，上面说的一些队列的特性很容易帮我们实现我们需要的情况。将接受的数据缓存一下，让处理的速度有些许缓冲，使得处理的速度赶得上接收的速度，上面又已经分析了普通队列与环形队列的优劣了，那么我们肯定是用环形队列来进行实现了。下面就是代码的实现：

定义一个结构体：

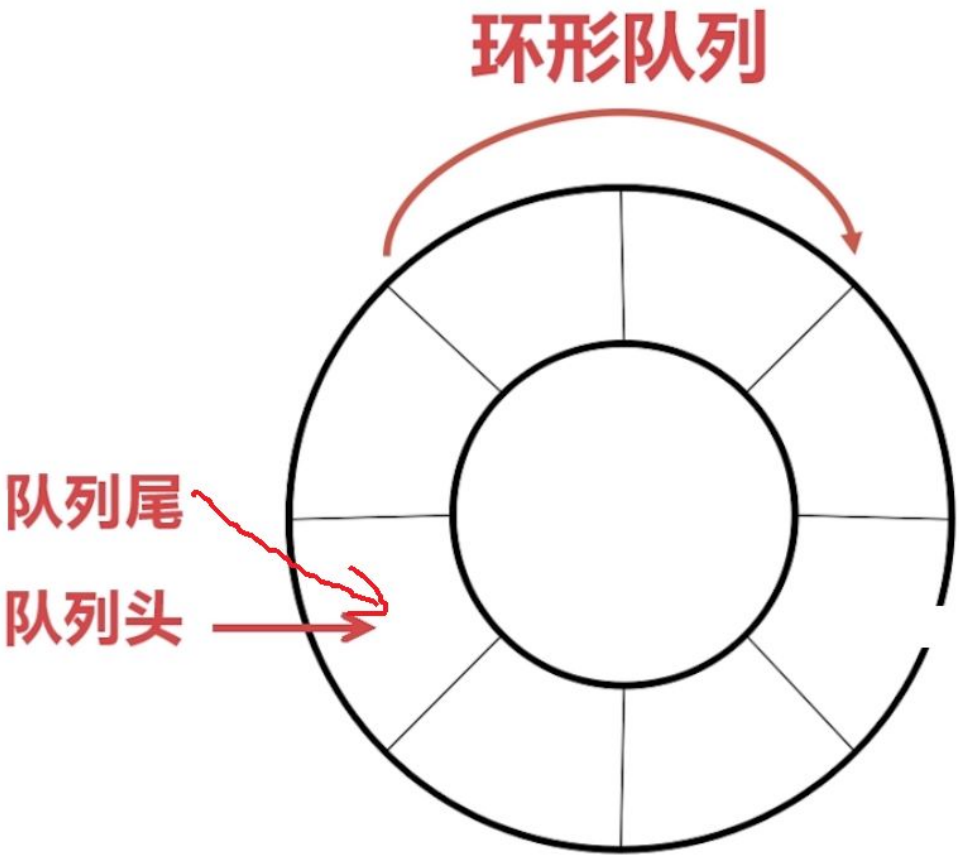
```
typedef struct
{
    u16 Head;
    u16 Tail;
    u16 Lenght;
    u8 Ring_Buff[RINGBUFF_LEN];
}RingBuff_t;
RingBuff_t ringBuff;//创建一个ringBuff的缓冲区
```

初始化

初始化结构体相关信息：使得我们的环形缓冲区是头尾相连的，并且里面没有数据，也就是空的队列。

```
/**
 * @brief  RingBuff_Init
 * @param  void
 * @return void
 * @author 杰杰
 * @date   2018
 * @version v1.0
 * @note   初始化环形缓冲区
 */
void RingBuff_Init(void)
{
    //初始化相关信息
    ringBuff.Head = 0;
    ringBuff.Tail = 0;
    ringBuff.Lenght = 0;
}
```

初始化效果如下：



写入环形缓冲区的代码实现：

```
/**
 * @brief  Write_RingBuff
 * @param  u8 data
 * @return FLASE:环形缓冲区已满，写入失败;TRUE:写入成功
 * @author 杰杰
```

```
 * @date   2018
 * @version v1.0
 * @note   往环形缓冲区写入u8类型的数据
 */
u8 Write_RingBuff(u8 data)
{
   if(ringBuff.Lenght >= RINGBUFF_LEN) //判断缓冲区是否已满
    {
      return FLASE;
    }
    ringBuff.Ring_Buff[ringBuff.Tail]=data;
//    ringBuff.Tail++;
    ringBuff.Tail = (ringBuff.Tail+1)%RINGBUFF_LEN;//防止越界非法访问
    ringBuff.Lenght++;
    return TRUE;
}
```

读取缓冲区的数据的代码实现：

```
/**
 * @brief  Read_RingBuff
 * @param  u8 *rData，用于保存读取的数据
 * @return FLASE:环形缓冲区没有数据，读取失败;TRUE:读取成功
 * @author 杰杰
 * @date   2018
 * @version v1.0
 * @note   从环形缓冲区读取一个u8类型的数据
 */
u8 Read_RingBuff(u8 *rData)
{
   if(ringBuff.Lenght == 0)//判断非空
    {
       return FLASE;
    }
   *rData = ringBuff.Ring_Buff[ringBuff.Head];//先进先出FIFO，从缓冲区头出
//   ringBuff.Head++;
   ringBuff.Head = (ringBuff.Head+1)%RINGBUFF_LEN;//防止越界非法访问
   ringBuff.Lenght--;
   return TRUE;
}
```

对于读写操作需要注意的地方有两个：

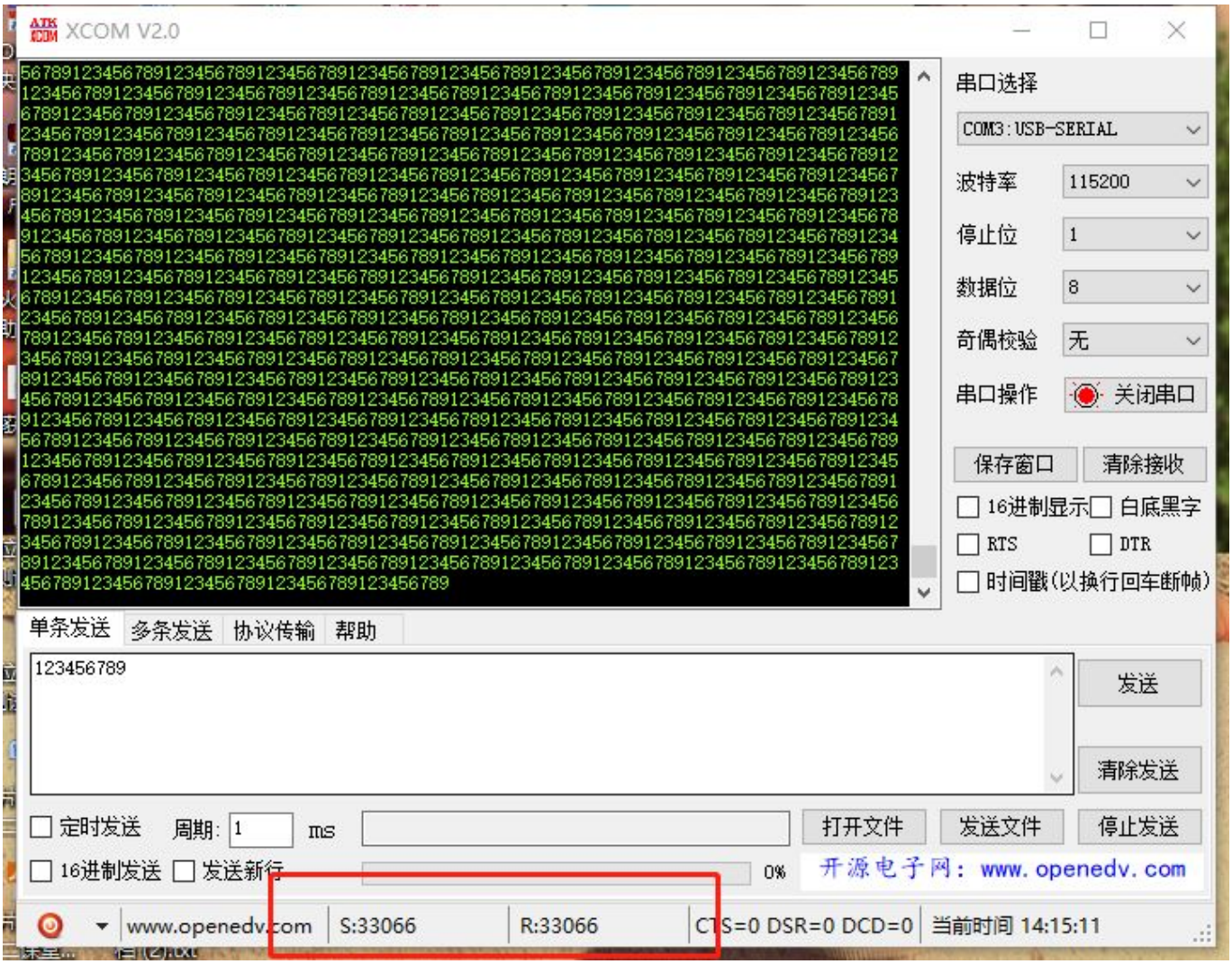
1. 判断队列是否为空或者满，如果空的话，是不允许读取数据的，返回FLASE。如果是满的话，也是不允许写入数据的，避免将已有数据覆盖掉。那么如果处理的速度赶不上接收的速度，可以适当增大缓冲区的大小，用空间换取时间。

2. 防止指针越界非法访问，程序有说明，需要使用者对整个缓冲区的大小进行把握。

那么在串口接收函数中：

```
void USART1_IRQHandler(void)
{
   if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)  //接收中断
                    {
            USART_ClearITPendingBit(USART1,USART_IT_RXNE);        //清楚标志位
            Write_RingBuff(USART_ReceiveData(USART1));       //读取接收到的数据
        }
}
```

测试效果



测试数据没有发生丢包现象

补充

对于现在的阶段，杰杰我本人写代码也慢慢学会规范了。所有的代码片段均使用了可读性很强的，还有可移植性也很强的。我使用了宏定义来决定是否开启环形缓冲区的方式来收发数据，移植到大家的代码并不会有其他副作用，只需要开启宏定义即可使用了。

```
#define USER_RINGBUFF  1  //使用环形缓冲区形式接收数据
#if  USER_RINGBUFF
/**如果使用环形缓冲形式接收串口数据***/
#define  RINGBUFF_LEN          200     //定义最大接收字节数 200
#define  FLASE   1
#define  TRUE    0
void RingBuff_Init(void);
u8 Write_RingBuff(u8 data);
u8 Read_RingBuff(u8 *rData);
#endif
```

当然，我们完全可以用空闲中断与DMA传输，效率更高，但是某些单片机没有空闲中断与DMA，那么这种环形缓冲区的作用就很大了，并且移植简便。

> 说明：文章部分截图来源慕课网james_yuan老师的课程

喜欢就关注我吧！

相关代码可以在公众号后台获取。