



The Real-Time Kernel

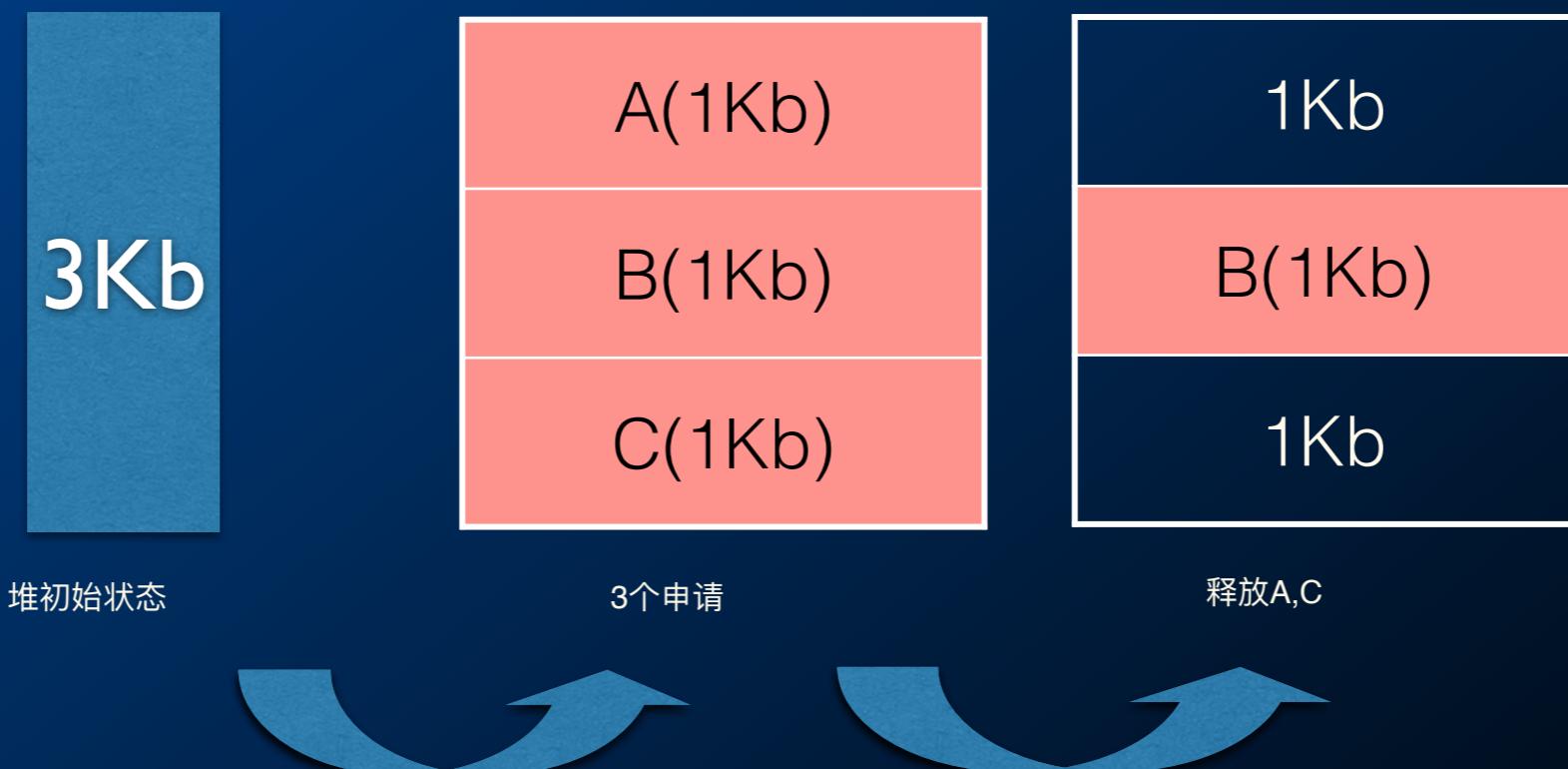
存储管理

概述

- μC/OS中是实模式存储管理
 - 不划分内核空间和用户空间，整个系统只有一个地址空间，即物理内存空间，应用程序和内核程序都能直接对所有的内存单元进行访问；
 - 系统中的“任务”，实际上都是线程——只有运行上下文和栈是独享的，其他资源都是共享的。
- 内存布局
 - 代码段(text)、数据段(data)、bss段、堆空间、栈空间；

malloc/free?

- 在ANSI C中可以用malloc()和free()两个函数动态地分配内存和释放内存。在嵌入式实时操作系统中，容易产生碎片。
- 由于内存管理算法的原因，malloc()和free()函数执行时间是不确定的。 μ C/OS-II 对malloc()和free()函数进行了改进，使得它们可以分配和释放固定大小的内存块。这样一来，malloc()和free()函数的执行时间也是固定的了



malloc()与free()

- 虽然可以调用标准的malloc()与free()库函数，但有以下一些问题。
 - 在小型嵌入式系统中可能不可用。
 - 具体实现可能会相对较大，会占用较多的代码空间。
 - 通常不具备线程安全特性。
 - 具有不确定性。每次调用的时的时间和开销可能不同。
 - 会产生内存碎片。
 - 会使得链接器配置的复杂。

μ C/OS中的存储管理

- μ C/OS采用的是固定分区的存储管理方法
 - μ C/OS把连续的大块内存按分区来管理，每个分区包含有整数个大小相同的块；
 - 在一个系统中可以有多个内存分区，这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区；
 - 采用这样的内存管理算法，上面的内存碎片问题就得到了解决。

内存分区示意图



内存控制块

- 为了便于管理，在μC/OS中使用内存控制块MCB（Memory Control Block）来跟踪每一个内存分区，系统中的每个内存分区都有它自己的 MCB。

```
typedef struct {  
    void *OSMemAddr; /*分区起始地址*/  
    void *OSMemFreeList; //下一个空闲内存块  
    INT32U OSMemBlkSize; /*内存块的大小*/  
    INT32U OSMemNBlks; /*内存块数量*/  
    INT32U OSMemNFree; /*空闲内存块数量*/  
} OS_MEM;
```

内存管理初始化

- 如果要在μC/OS-II中使用内存管理，需要在OS_CFG.H文件中将开关量OS_MEM_EN设置为1。这样μC/OS-II 在系统初始化OSInit()时就会调用OSMemInit()，对内存管理器进行初始化，建立空闲的内存控制块链表。

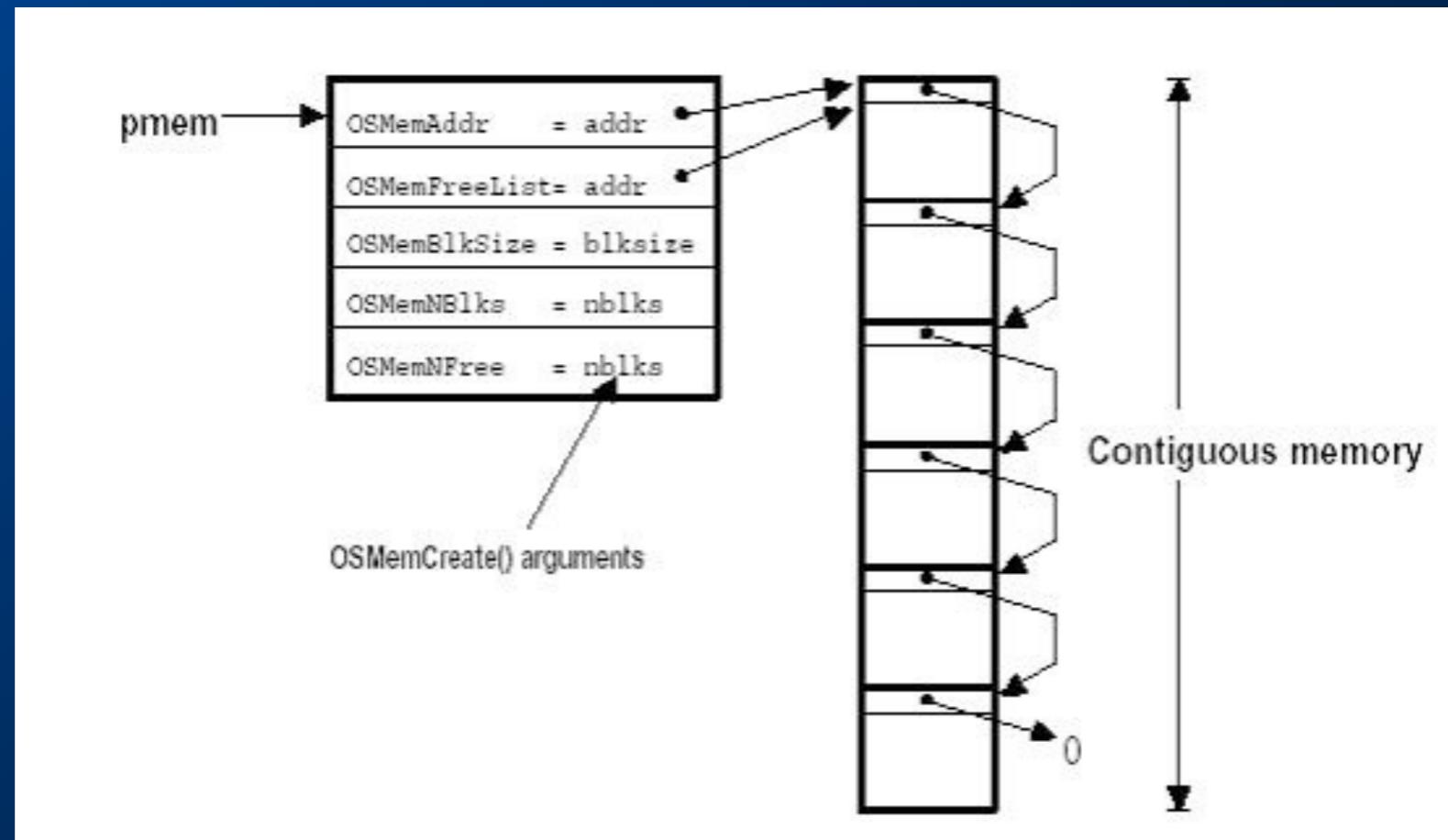


创建一个内存分区

- OSMemCreate()
 - OS_MEMORY *OSMemCreate (
void *addr, // 内存分区的起始地址
INT32U nblks, // 分区内的内存块数
INT32U blksize, // 每个内存块的字节数
INT8U *err); // 指向错误码的指针
- 例子
 - OS_MEMORY *CommTxBuf;
 - INT8U CommTxPart[100][32];
 - CommTxBuf = OSMemCreate(CommTxPart,
100, 32, &err);

- OSMemCreate()

- 从系统的空闲内存控制块中取得一个MCB；
- 将这个内存分区中的所有内存块链接成一个单向链表；
- 在对应的MCB中填写相应的信息。



分配一个内存块

- `void *OSMemGet(OS_MEM *pmem, INT8U *err);`
- 功能：从已经建立的内存分区中申请一个内存块。该函数的唯一参数是指向特定内存分区的指针。如果没有空闲的内存块可用，返回NULL指针。
- 应用程序必须知道内存块的大小，并且在使用时不能超过该容量。

释放一个内存块

- INT8U OSMemPut(OS_MEMORY *pmem, void *pblk);
- 功能：将一个内存块释放并放回到相应的内存分区中。
- 注意：用户应用程序必须确认将内存块放回到了正确的内存分区中，因为OSMemPut()并不知道一个内存块是属于哪个内存分区的。

等待一个内存块

- 如果没有空闲的内存块，OSMemGet()立即返回NULL。能否在没有空闲内存块的时候让任务进入等待状态？
- μC/OS-II本身在内存管理上并不支持这项功能，如果需要的话，可以通过为特定内存分区增加信号量的方法，来实现此功能。
- 基本思路：当应用程序需要申请内存块时，首先要得到一个相应的信号量，然后才能调用OSMemGet()函数。

```
OS_EVENT *SemaphorePtr;
OS_MEM  *PartitionPtr;
INT8U Partition[100][32];
OS_STK TaskStk[1000];
void main(void)
{
    INT8U err;
    OSInit();
    ...
    SemaphorePtr = OSSemCreate(100);
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);
    OSSStart();
}
```



```
void Task (void *pdata)
{
    INT8U err;
    INT8U *pblock;

    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);

        pblock = OSMemGet(PartitionPtr, &err);
        /* 使用内存块 */

        ...

        OSMemPut(PartitionPtr, pblock);

        OSSemPost(SemaphorePtr);
    }
}
```



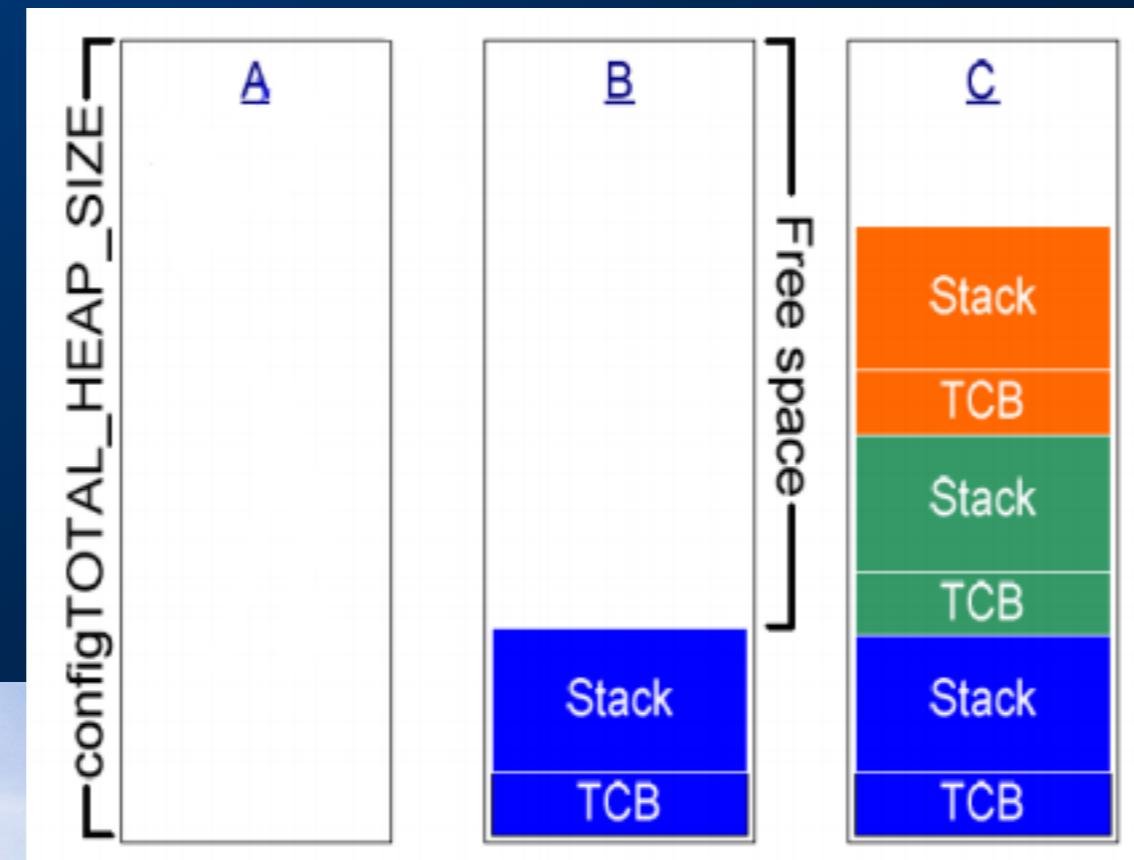
freertos内存管理

- 三种pvPortMalloc()和vPortFree()的实现范例



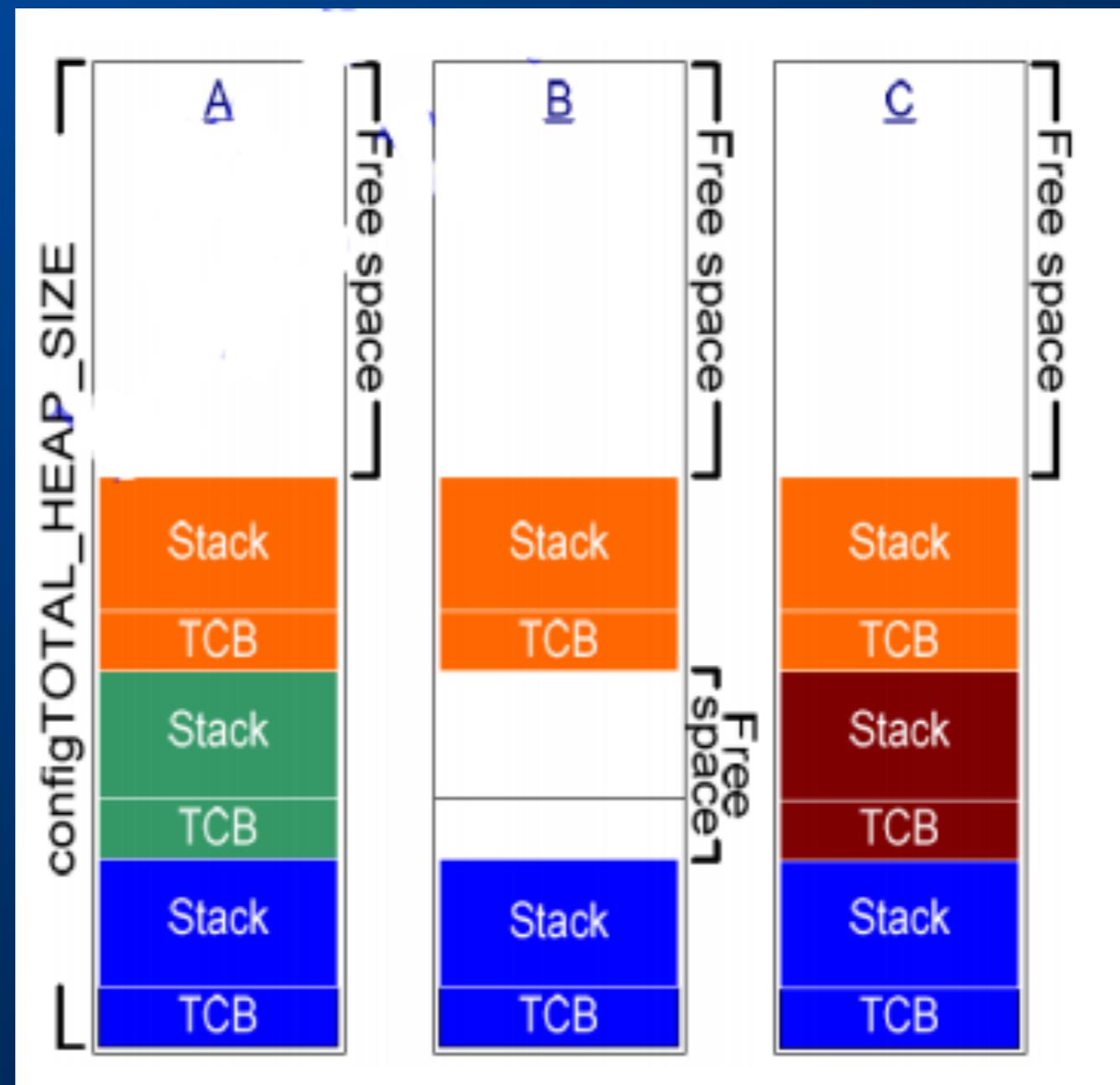
Heap_1.c

- 其实现了一个非常基本的pvPortMalloc()版本，而没有实现vPortFree()。如果应用程序不需要删除任务，队列或者信号量，则其具有使用heap_1的潜质。其具有确定性。
- 这种分配方案将FreeRTOS的内存堆空间堪称一个简单的数组。当调用pvPortMalloc()时，则将数组又简单的细分成为更小的内存块。数组大小在FreeRTOSConfig.h中由configTOTAL_HEAP_SIZE定义。



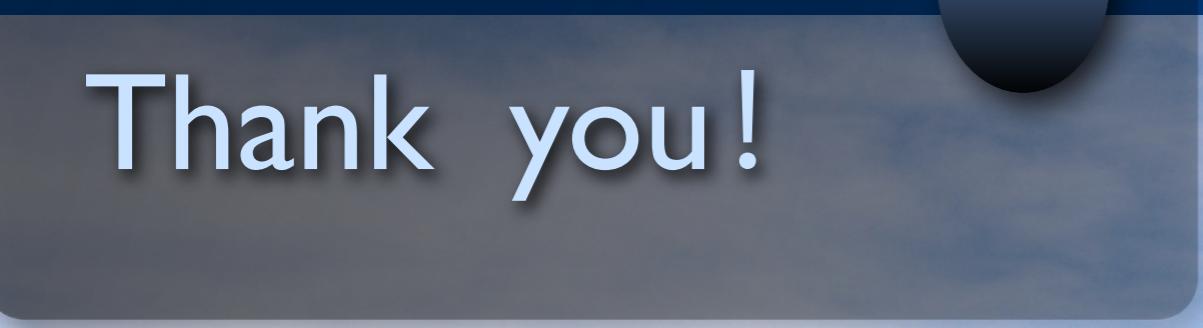
Heap_2.c

- 其采用了一个最佳匹配算法来分配内存，并支持内存释放。由于声明了一个静态数组，所以会让整个应用程序看起来耗费了很多内存，即使是在数组没有进行任何实际分配之前。
- 最佳匹配算法保证pvPortMalloc()会使用最接近请求大小的空间块。
例如：
 - 对空间包含了三个空闲内存块，分别为5字节，25字节和100字节。
 - pvPortMalloc()被调用用以请求分配20字节大小的内存空间。
- Heap_2.c不会把相邻的空闲块合并成一个更大的内存块，所以会产生内存碎片如果分配和释放的总是相同大小的内存块，则内存碎片不会称为一个问题。所以Heap_2.c适合于那些重复创建与删除具有相同空间任务的应用程序。



Heap_3.c

- 简单的调用了标准库malloc()和free(), 但是通过暂时挂起调度器使得函数调用具备了线程安全特性。



Thank you!

