title: 【TencentOS tiny】深度源码分析（3）——队列 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:11:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

队列基本概念

队列是一种常用于任务间通信的数据结构，队列可以在任务与任务间、中断和任务间传递消息，实现了任务接收来自其他任务或中断的不固定长度的消息，任务能够从队列里面读取消息，当队列中的消息是空时，读取消息的任务将被阻塞，用户还可以指定任务等待消息的时间timeout，在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当队列中有新消息时，被阻塞的任务会被唤醒并处理新消息；当等待的时间超过了指定的阻塞时间，即使队列中尚无有效数据，任务也会自动从阻塞态转为就绪态，消息队列是一种异步的通信方式。

通过队列服务，任务或中断服务例程可以将一条或多条消息放入队列中。同样，一个或多个任务可以从队列中获得消息。当有多个消息发送到队列时，通常是将先进入队列的消息先传给任务，也就是说，任务先得到的是最先进入队列的消息，即先进先出原则（FIFO），其实TencentOS tiny暂时不支持后进先出原则LIFO操作队列，但是支持后进先出操作消息队列。

> 提示：TencentOS tiny 的队列不等同于消息队列，虽然队列的底层实现是**依赖**消息队列，但在TencentOS tiny中将它们分离开，这是两个概念，毕竟操作是不一样的。

队列的阻塞机制

举个简单的例子来理解操作系统中的阻塞机制：

> 假设你某天去餐厅吃饭，但是餐厅没菜了，那么你可能会有3个选择，你扭头就走，既然都没菜了，肯定换一家餐厅啊是吧。或者你会选择等一下，说不定老板去买菜了，一会就有菜了呢，就能吃饭。又或者，你觉得这家餐厅非常好吃，吃不到饭你就不走了，在这死等~

同样的：假设有一个任务A对某个队列进行读操作的时候（出队），发现它此时是没有消息的，那么此时任务A有3个选择：第一个选择，任务A扭头就走，既然队列没有消息，那我也不等了，干其它事情去，这样子任务A不会进入阻塞态；第二个选择，任务A还是在这里等等吧，可能过一会队列就有消息，此时任务A会进入阻塞状态，在等待着消息的到来，而任务A的等待时间就由我们自己指定，当阻塞的这段时间中任务A等到了队列的消息，那么任务A就会从阻塞态变成就绪态；假如等待超时了，队列还没消息，那任务A就不等了，从阻塞态中唤醒；第三个选择，任务A死等，不等到消息就不走了，这样子任务A就会进入阻塞态，直到完成读取队列的消息。

队列实现的数据结构

队列控制块

TencentOS tiny 通过队列控制块操作队列，其数据类型为k_queue_t，队列控制块由多个元素组成，主要有pend_obj_t 类型的pend_obj以及k_msg_queue_t 类型的msg_queue**消息列表**。其实整个队列的实现非常简

单，主要靠msg_queue中的queue_head成员变量（这其实是一个消息列表（消息链表）），所有的消息都会被记录在这个消息列表中，当读取消息的时候，会从消息列表读取消息。

> 继承自内核对象的数据结构 在 **\kernel\core\include\tos_pend.h** 的 **35** 行

```
typedef struct pend_object_st {
    pend_type_t     type;
    k_list_t        list;
} pend_obj_t;
```

> 消息列表的数据类型（消息队列控制块），在 **\kernel\core\include\tos_msg.h** 文件的第 **13** 行

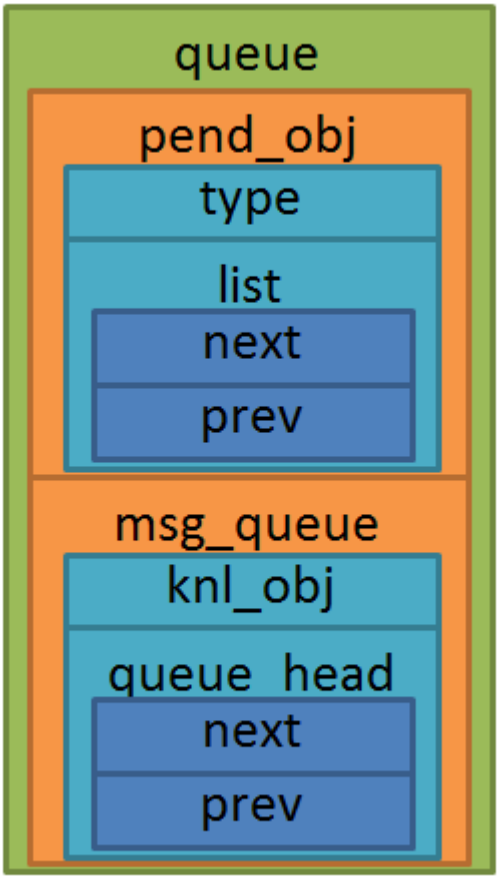
```
typedef struct k_msg_queue_st {
#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    knl_obj_t       knl_obj;
#endif

    k_list_t        queue_head;
} k_msg_queue_t;
```

> 队列控制块，在 **\kernel\core\include\tos_queue.h** 文件的第 **6** 行

```
typedef struct k_queue_st {
    pend_obj_t      pend_obj;
    k_msg_queue_t msg_queue;
} k_queue_t;
```

队列控制块示意图如下：

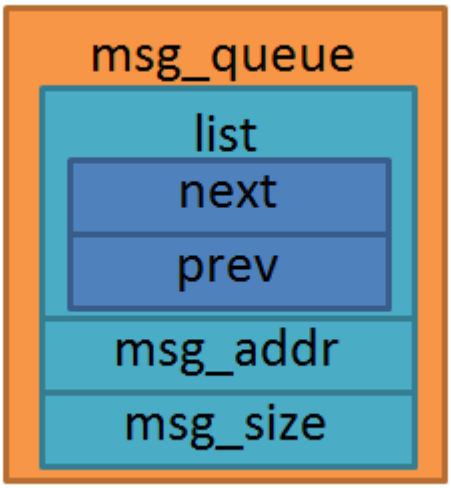


消息控制块

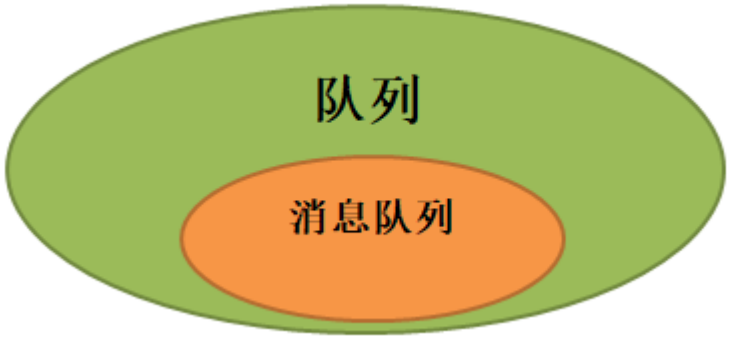
除了上述的队列控制块外，还有消息队列控制块，这是因为TencentOS tiny中实现队列是依赖消息队列的，既然队列可以传递数据（消息），则必须存在一种可以存储消息的数据结构，我称之为消息控制块，消息控制块中记录了消息的存储地址msg_addr，以及消息的大小msg_size，此外还存在一个list成员变量，可以将消息挂载到队列的消息列表中。

> **消息控制块数据结构， 在 \kernel\core\include\tos_msg.h 文件的第 7 行**

```
typedef struct k_message_st {
    k_list_t        list;
    void           *msg_addr;
    size_t          msg_size;
} k_msg_t;
```



> 其实队列的实现依赖于消息队列，他们的关系如下：



任务控制块中的消息成员变量

假设任务A在队列中等待消息，而中断或其他任务往任务A等待的队列写入（发送）一个消息，那么这个消息不会被挂载到队列的消息列表中，而是会直接被记录在任务A的任务控制块中，表示任务A从队列中等待到这个消息，因此任务控制块必须存在一些成员变量用于记录消息相关信息（如消息地址、消息大小等）：

> 任务控制块数据结构 在**\kernel\core\include\tos_task.h**文件的第 **90** 行

```
typedef struct k_task_st {
...
#if TOS_CFG_MSG_EN > 0u
    void                *msg_addr;          /**< 保存接收到的消息地址 */
    size_t               msg_size;                      /**< 保存接收到的消息大小
*/
#endif
...
} k_task_t;
```

与消息相关的宏定义

在tos_config.h文件中，使能队列组件的宏定义TOS_CFG_QUEUE_EN，使能消息队列组件宏定义TOS_CFG_MSG_EN，系统支持的消息池中消息个数宏定义TOS_CFG_MSG_POOL_SIZE。

```
#define TOS_CFG_QUEUE_EN                                1u

#define TOS_CFG_MSG_EN                                  1u

#define TOS_CFG_MSG_POOL_SIZE           3u
```

消息池

在TencentOS tiny中定义了一个数组k_msg_pool[TOS_CFG_MSG_POOL_SIZE]作为消息池，它的数据类型是消息控制块类型k_msg_t，因为在使用消息队列的时候存取消息比较频繁，而在系统初始化的时候就将这个大数组的各个元素串初始化，并挂载到空闲消息列表中k_msg_freelist，组成我们说的消息池k_msg_pool，而池中的成员变量就是我们所说的消息。

> 为什么使用池化的方式处理消息呢，因为高效，复用率高，就像我们在池塘中去一勺水，在使用完毕再将其归还到池塘，这种操作是非常高效的，并且在有限资源的嵌入式中能将资源重复有效地利用起来。

> 消息池相关的定义 在**\kernel\core\tos_global.c**文件 第 **51** 行

```
#if TOS_CFG_MSG_EN > 0u
TOS_LIST_DEFINE(k_msg_freelist);

k_msg_t             k_msg_pool[TOS_CFG_MSG_POOL_SIZE];

#endif
```

队列创建

tos_queue_create()函数用于创建一个队列，队列就是一个数据结构，用于任务间的数据的传递。每创建一个新的队列都需要为其分配RAM，在创建的时候我们需要自己定义一个队列控制块，其内存是由编译器自动分配的。在创建的过程中实际上就是将队列控制块的内容进行初始化，将队列控制块的 pend_obj成员变量中的 type 属性标识为PEND_TYPE_QUEUE，表示这是一个队列，然后调用消息队列中的API函数tos_msg_queue_create()将队列的消息成员变量msg_queue初始化，实际上就是初始化消息列表。

> 创建队列函数，在**\kernel\core\tos_queue.c** 第 **5** 行

```
__API__ k_err_t tos_queue_create(k_queue_t *queue)
{
    TOS_PTR_SANITY_CHECK(queue);

    pend_object_init(&queue->pend_obj, PEND_TYPE_QUEUE);
    tos_msg_queue_create(&queue->msg_queue);
```

```
    return K_ERR_NONE;
}
```

销毁队列

tos_queue_destroy()函数用于销毁一个队列，当队列不在使用是可以将其销毁，销毁的本质其实是将队列控制块的内容进行清除，首先判断一下队列控制块的类型是PEND_TYPE_QUEUE，这个函数只能销毁队列类型的控制块。然后判断是否有任务在等待队列中的消息，如果有则调用pend_wakeup_all()函数将这项任务唤醒，然后调用tos_msg_queue_flush()函数将队列的消息列表的消息全部"清空"，"清空"的意思是将挂载到队列上的消息释放回消息池（如果队列的消息列表存在消息，使用msgpool_free()函数释放消息），knl_object_deinit()函数是为了确保队列已经被销毁，此时队列控制块的pend_obj成员变量中的type 属性标识为KNL_OBJ_TYPE_NONE。最后在销毁队列后进行一次任务调度，以切换任务（毕竟刚刚很可能唤醒了任务）。

> 但是有一点要注意，因为队列控制块的RAM是由编译器静态分配的，所以即使是销毁了队列，这个内存也是没办法释放的~

> 销毁队列函数，在**\kernel\core\tos_queue.c 第 14 行**

```
__API__ k_err_t tos_queue_destroy(k_queue_t *queue)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(queue);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&queue->pend_obj, PEND_TYPE_QUEUE)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();

    if (!pend_is_nopending(&queue->pend_obj)) {
        pend_wakeup_all(&queue->pend_obj, PEND_STATE_DESTROY);
    }

    pend_object_deinit(&queue->pend_obj);
    tos_msg_queue_flush(&queue->msg_queue);

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

清空队列

清空队列实际上就是将消息释放回消息池中，本质上还是调用tos_msg_queue_flush()函数。它是依赖于消息队列实现的。

> 清空队列函数，在**\kernel\core\tos_queue.c 第 41 行**

```
__API__ k_err_t tos_queue_flush(k_queue_t *queue)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(queue);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&queue->pend_obj, PEND_TYPE_QUEUE)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();
    tos_msg_queue_flush(&queue->msg_queue);
    TOS_CPU_INT_ENABLE();

    return K_ERR_NONE;
}
```

等待队列（消息）

当任务试图从队列中的获取消息时，用户可以指定一个等待时间，当且仅当队列存在消息的时候，任务才能获取到消息。在等待的这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列消息有效。当其他任务或中断服务程序往其等待的队列中写入了数据，该任务将自动由阻塞态转为就绪态。当任务等待发生超时，即使队列中尚无有效消息，任务也会自动从阻塞态转为就绪态。 等待队列的过程也是非常简单的，先来看看参数吧（其中msg_addr与msg_size参数是用于保存函数返回的内容，即输出）：

参数	说明
queue	队列控制块指针
msg_addr	用于保存获取到的消息（这是输出的）
msg_size	用于保存获取到消息的大小（这是输出的）
timeout	等待时间（以k_tick_t为单位）

等待队列消息的过程如下：

1. 首先检测传入的参数是否正确
2. 尝试调用tos_msg_queue_get()函数获取消息，如果队列存在消息则会获取成功（返回K_ERR_NONE），否则获取失败。（关于该函数在下一章讲解）

3. 当获取成功则可以直接退出函数，而当获取消息失败的时候，则可以根据指定的等待时间timeout进行阻塞，如果不等待（timeout =TOS_TIME_NOWAIT），则直接返回错误代码K_ERR_PEND_NOWAIT。

4. 如果调度器被锁了knl_is_sched_locked()，则无法进行等待操作，返回错误代码K_ERR_PEND_SCHED_LOCKED，毕竟需要切换任务，调度器被锁则无法切换任务。

5. 调用pend_task_block()函数将任务阻塞，该函数实际上就是将任务从就绪列表中移除k_rdyq.task_list_head[task_prio]，并且插入到等待列表中object->list，如果等待的时间不是永久等待TOS_TIME_FOREVER，还会将任务插入时间列表中k_tick_list，阻塞时间为timeout，然后进行一次任务调度knl_sched()。

6. 当程序能执行到pend_state2errno()时，则表示任务等待到消息，又或者发生超时，那么就调用pend_state2errno()函数获取一下任务的等待状态，看一下是哪种情况导致任务恢复运行。

7. 如果是正常情况（等待获取到消息），则将消息从任务控制块的k_curr_task->msg_addr读取出来，并且写入msg_addr 中用于返回。同样的消息的大小也是会通过msg_size 返回。

> 获取（等待）队列消息函数，在**\kernel\core\tos_queue.c 第 60 行**

```
__API__ k_err_t tos_queue_pend(k_queue_t *queue, void **msg_addr, size_t
*msg_size, k_tick_t timeout)
{
    TOS_CPU_CPSR_ALLOC();
    k_err_t err;

    TOS_PTR_SANITY_CHECK(queue);
    TOS_PTR_SANITY_CHECK(msg_addr);
    TOS_PTR_SANITY_CHECK(msg_size);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&queue->pend_obj, PEND_TYPE_QUEUE)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();

    if (tos_msg_queue_get(&queue->msg_queue, msg_addr, msg_size) == K_ERR_NONE) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

    if (timeout == TOS_TIME_NOWAIT) {
        *msg_addr = K_NULL;
        *msg_size = 0;
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_NOWAIT;
    }

    if (knl_is_sched_locked()) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_SCHED_LOCKED;
```

```
    }

    pend_task_block(k_curr_task, &queue->pend_obj, timeout);

    TOS_CPU_INT_ENABLE();
    knl_sched();

    err = pend_state2errno(k_curr_task->pend_state);

    if (err == K_ERR_NONE) {
        *msg_addr = k_curr_task->msg_addr;
        *msg_size = k_curr_task->msg_size;
        k_curr_task->msg_addr = K_NULL;
        k_curr_task->msg_size = 0;
    }

    return err;
}
```

将等待消息的任务添加到对应等待列表函数，在**\kernel\core\tos_pend.c**文件的 第 **106** 行

```
__KERNEL__ void pend_task_block(k_task_t *task, pend_obj_t *object, k_tick_t
timeout)
{
    readyqueue_remove(task);
    pend_list_add(task, object);

    if (timeout != TOS_TIME_FOREVER) {
        tick_list_add(task, timeout);
    }
}
```

获取任务等待状态的函数，在**\kernel\core\tos_pend.c**文件的 第 **72** 行

```
__KERNEL__ k_err_t pend_state2errno(pend_state_t state)
{
    if (state == PEND_STATE_POST) {
        return K_ERR_NONE;
    } else if (state == PEND_STATE_TIMEOUT) {
        return K_ERR_PEND_TIMEOUT;
    } else if (state == PEND_STATE_DESTROY) {
        return K_ERR_PEND_DESTROY;
    } else if (state == PEND_STATE_OWNER_DIE) {
        return K_ERR_PEND_OWNER_DIE;
    } else {
        return K_ERR_PEND_ABNORMAL;
    }
}
```

（消息）写入队列

任务或者中断服务程序都可以给消息队列发送消息，当发送消息时，TencentOS tiny会从消息池中取出一个消息，挂载到队列的消息列表末尾（FIFO发送方式）。tos_queue_post()是唤醒一个等待队列消息任务，tos_queue_post_all()则会唤醒所有等待队列消息的任务，无论何种情况，都是调用queue_do_post将消息写入队列中。 消息的写入队列过程：

1. 首先检测传入的参数是否正确
2. 判断一下是否有任务在等待消息，如果有则根据opt参数决定唤醒一个任务或者所有等待任务，否则直接将消息写入队列中。
3. 当没有任务在等待消息时，调用tos_msg_queue_put()函数将消息写入队列，写入队列的方式遵循FIFO原则（TOS_OPT_MSG_PUT_FIFO），写入成功返回K_ERR_NONE。而如果消息池已经没有消息了（消息最大个数由TOS_CFG_MSG_POOL_SIZE宏定义决定），则写入失败，返回K_ERR_QUEUE_FULL错误代码。（关于该函数将在下一章讲解）
4. 如果有任务在等待消息，则调用queue_task_msg_recv()函数将消息内容与大小写入任务控制块的msg_addr与msg_size成员变量中，此外还需要唤醒任务，就通过调用pend_task_wakeup()函数将对应的等待任务唤醒，核心处理思想就是通过TOS_LIST_FIRST_ENTRY获取到等待在队列上的任务，然后唤醒它。
5. 对于唤醒所有等待任务的处理其实也是一样的，只不过是多了个循环处理，把等待列表中的所有任务依次唤醒，仅此而已~

> 写入队列消息函数，在**\kernel\core\tos_queue.c** 第 **159** 、**164** 行

```
__API__ k_err_t tos_queue_post(k_queue_t *queue, void *msg_addr, size_t msg_size)
{
    TOS_PTR_SANITY_CHECK(queue);
    TOS_PTR_SANITY_CHECK(msg_addr);

    return queue_do_post(queue, msg_addr, msg_size, OPT_POST_ONE);
}

__API__ k_err_t tos_queue_post_all(k_queue_t *queue, void *msg_addr, size_t
msg_size)
{
    TOS_PTR_SANITY_CHECK(queue);
    TOS_PTR_SANITY_CHECK(msg_addr);

    return queue_do_post(queue, msg_addr, msg_size, OPT_POST_ALL);
}
```

> 写入队列消息函数实际调用的函数，通过opt参数进行不一样的处理，在**\kernel\core\tos_queue.c** 第 **118** 行

```
__STATIC__ k_err_t queue_do_post(k_queue_t *queue, void *msg_addr, size_t
msg_size, opt_post_t opt)
{
    TOS_CPU_CPSR_ALLOC();
    k_list_t *curr, *next;

    TOS_PTR_SANITY_CHECK(queue);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&queue->pend_obj, PEND_TYPE_QUEUE)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();

    if (pend_is_nopending(&queue->pend_obj)) {
        if (tos_msg_queue_put(&queue->msg_queue, msg_addr, msg_size,
TOS_OPT_MSG_PUT_FIFO) != K_ERR_NONE) {
            TOS_CPU_INT_ENABLE();
            return K_ERR_QUEUE_FULL;
        }
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

    if (opt == OPT_POST_ONE) {
        queue_task_msg_recv(TOS_LIST_FIRST_ENTRY(&queue->pend_obj.list, k_task_t,
pend_list),
                            msg_addr, msg_size);
    } else { // OPT_QUEUE_POST_ALL
        TOS_LIST_FOR_EACH_SAFE(curr, next, &queue->pend_obj.list) {
            queue_task_msg_recv(TOS_LIST_ENTRY(curr, k_task_t, pend_list),
                            msg_addr, msg_size);
        }
    }

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

> 唤醒等待的任务函数，在**\kernel\core\tos_pend.c**文件 的第 **87** 行

唤醒等待任务的思想就是将任务从对应的等待列表移除，然后添加到就绪列表中。

```
__KERNEL__ void pend_task_wakeup(k_task_t *task, pend_state_t state)
{
    if (task_state_is_pending(task)) {
        // mark why we wakeup
```

```
            task->pend_state = state;
            pend_list_remove(task);
        }

        if (task_state_is_sleeping(task)) {
            tick_list_remove(task);
        }

        if (task_state_is_suspended(task)) {
            return;
        }

        readyqueue_add(task);
    }
```

总结

代码精悍短小，思想清晰，非常建议深入学习~

喜欢就关注我吧！