消息队列基本使用

RealTouch 评估板 RT-Thread 入门文档

版本号：1.0.0

日期：2012/8/12

修订记录

日期	作者	修订历史
2012/8/12	prife	创建文档

实验目的

- 了解消息队列的基本使用
- 熟练使用消息队列实现多个线程间通信

硬件说明

本实验使用 RT-Thread 官方的 Realtouch 开发板作为实验平台。涉及到的硬件主要为：

- 串口 3，作为 rt_kprintf 输出
  需要连接 JTAG 扩展板，具体请参见《Realtouch 开发板使用手册》

实验原理及程序结构

实验设计

本实验演示消息队列在 RT-Thread 中作为线程间通信的手段是如何使用的。

实验中创建三个线程，线程 1 从消息队列中获取消息；线程 2 向消息队列中发送普通消息，即消息会被放在消息队列的尾部；线程 3 向消息队列中发送一个紧急消息，即这个消息会被放在消息队列的头部。

本实验同样使用静态消息队列来作为演示，涉及静态消息队列初始化/脱离。动态消息创建/删除类似，不再赘述。

源程序说明

本实验对应 kernel_message_queue

系统依赖

在 rtconfig.h 中需要开启

- #define RT_USING_HEAP
  此项可选，开启此项可以创建动态线程和动态互斥量，如果使用静态线程和静态互斥量，则此项不是必要的
- #define RT_USING_MESSAGEQUEUE

此项必须，开启此项后才可以使用消息队列相关 API

- #define RT_USING_CONSOLE
  此项必须，本实验使用 rt_kprintf 向串口打印按键信息，因此需要开启此项

主程序说明

在 applications/application.c 中定义静态消息队列控制块、存放消息的缓冲区。如下所示

定义全局变量代码

```
#define MSG_VIP "over"

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static char msg_pool[2048];
```

在 applications/application.c 中的 int rt_application_init()函数中，初始化消息队列代码如下所示。

初始化消息队列代码

```
    rt_err_t result;

    /* 初始化消息队列 */
    rt_mq_init(&mq, "mqt",
        &msg_pool[0], /* 内存池指向 msg_pool */
        128 - sizeof(void*), /* 每个消息的大小是 128 - void* */
        sizeof(msg_pool), /* 内存池的大小是 msg_pool 的大小 */
        RT_IPC_FLAG_FIFO);/* 如果有多个线程等待，按照先来先得到的方法分配消息 */
    if (result != RT_EOK)
    {
        rt_kprintf("init message queue failed.\n");
        return -1;
 }
```

在 int rt_application_init()初始化名为”thread1”的 thread1 的静态线程，如下所示。

初始化线程 1 代码

```
    rt_thread_init(&thread1,
                   "thread1",
                   thread1_entry,
                   RT_NULL,
                   &thread1_stack[0],
                   sizeof(thread1_stack),10,50);
  rt_thread_startup(&thread1);
```

其线程入口函数如下所示，线程 1 不停地从消息队列中取出消息，打印消息内容，并检测是否为特殊消息（宏 MSG_VIP 表示），若是，则线程 1 不再循环接收邮件，从 while 循环中调出，线程函数运行结束；若否，则继续从消息队列中获取消息。

如果消息队列中没有消息，则线程 1 会被挂起，直到消息中有消息时，线程 1 会被唤醒，从挂起变为就绪态。

线程 1 代码

```
  ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
  static char thread1_stack[1024];  //设置线程堆栈为 1024Bytes
  struct rt_thread thread1;       //定义静态线程数据结构
  /* 线程 1 入口 */
  static void thread1_entry(void* parameter)
  {
      char buf[128];

      while (1)
      {
          rt_memset(&buf[0], 0, sizeof(buf));

          /* 从消息队列中接收消息 */
          if (rt_mq_recv(&mq, &buf[0], sizeof(buf), RT_WAITING_FOREVER)
== RT_EOK)
          {
              rt_kprintf("thread1: recv msg from msg queue, the
content:%s\n", buf);

              /* 检查是否收到了紧急消息 */
              if (strcmp(buf, MSG_VIP) == 0)
```

```
            break;
        }

        /* 延时 1s */
        rt_thread_delay(RT_TICK_PER_SECOND);
    }

    rt_kprintf("thread1: got an urgent message, leave\n");
}
```

在 int rt_application_init()初始化名为”thread2”的 thread2 的静态线程，如下所示。

初始化线程 2 代码

```
    rt_thread_init(&thread2,
                   "thread2",
                   thread2_entry,
                   RT_NULL,
                   &thread2_stack[0],
                   sizeof(thread2_stack),10,50);
rt_thread_startup(&thread2);
```

其线程入口函数如下所示，线程 2 不停地向消息队列中发送消息，直到消息缓冲区被充满后，跳出循环，调用 rt_kprintf 打印结束信息后，线程函数运行结束。

线程 2 代码

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread2_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread2;       //定义静态线程数据结构
/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    int i, result;
    char buf[128];

    i = 0;
    while (1)
    {
```

```
        rt_snprintf(buf, sizeof(buf), "this is message No.%d", i);

        /* 发送消息到消息队列中 */
        result = rt_mq_send(&mq, &buf[0], sizeof(buf));
        if ( result == -RT_EFULL)
            break;

        rt_kprintf("thread2: send message - %s\n", buf);

        i++;
    }

    rt_kprintf("message queue full, thread2 leave\n");
}
```

在 int rt_application_init()初始化名为”thread3”的 thread3 的静态线程，如下所示。

初始化线程 3 代码

```
    rt_thread_init(&thread3,
                   "thread3",
                   thread3_entry,
                   RT_NULL,
                   &thread3_stack[0],
                   sizeof(thread3_stack),10,50);
    rt_thread_startup(&thread3);
```

其线程入口函数如下所示，线程 3 入口函数中，首先使用 rt_thread_delay 延时 5 秒钟，使得线程 1 和线程 2 充分运行，之后调用 rt_mq_urgent 函数向消息队列中发送紧急消息，这个函数会将消息插入到消息队列头部，因此接收消息的线程会优先取得。

如果此时消息队列已经满，则延时 20 个 tick 后重复发送，直到正确发送紧急消息，发送成功后线程 3 入口函数执行完毕退出。

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread3_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread3;       //定义静态线程数据结构
```

```
/* 线程 3 入口函数 */
static void thread3_entry(void* parameter)
{
    char msg[] = MSG_VIP;
    int result;

    rt_thread_delay(RT_TICK_PER_SECOND * 5);
    rt_kprintf("thread3: send an urgent message <%s> \n", msg);

    /* 发送紧急消息到消息队列中 */
    do {
        result = rt_mq_urgent(&mq, &msg[0], sizeof(msg));

        if (result != RT_EOK)
            rt_thread_delay(20);
    } while (result != RT_EOK);

}
```

3 编译调试及观察输出信息

编译请参见《RT-Thread 配置开发环境指南》完成编译烧录，参考《Realtouch 开发板使用手册》完成硬件连接，连接扩展板上的串口和 jlink。

运行后可以看到如下信息：

串口输出信息

```
 \ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Aug  9 2012
 2006 - 2012 Copyright by rt-thread team
thread2: send message - this is message No.0
thread2: send message - this is message No.1
thread2: send message - this is message No.2
thread2: send message - this is message No.3
thread2: send message - this is message No.4
thread2: send message - this is message No.5
thread2: send message - this is message No.6
```

```
  thread2: send message - this is message No.7
  thread2: send message - this is message No.8
  thread2: send message - this is message No.9
  thread2: send message - this is message No.10
  thread2: send message - this is message No.11
  thread2: send message - thread1: recv msg from msg queue, the
content:this is message No.0
  tg queue, the content:thread2: send message - this is message No.13
  thread2: send message - this is message No.14
  thread2: send message - this is message No.15
  message queue full, thread2 leave
  thread1: recv msg from msg queue, the content:this is message No.1
  thread1: recv msg from msg queue, the content:this is message No.2
  thread1: recv msg from msg queue, the content:this is message No.3
  thread1: recv msg from msg queue, the content:this is message No.4
  thread3: send an urgent message <over>
  thread1: recv msg from msg queue, the content:over
  thread1: got an urgent message, leave
```

结果分析

整个程序运行过程中各个线程的状态变化：

rt_application_init 中创建了三个线程，thread1（线程 1）、thread2（线程 2）、thread3（线程 3），它们具有相同的优先级和时间片配置。由于先使用 rt_thread_startup(&thread1)，故线程 1 优先运行，其次运行 thread2，最后运行 thread3。

在线程 1 中，首先从消息队列中接收消息，此时消息队列中并没有消息，线程 1 被挂起，内核会调度线程 2 运行。在线程 2 的线程处理函数中，不停使用 rt_mq_send 向消息队列中发送消息。注意，在 rt_mq_send 这个函数中，线程 1 就已经从挂起状态恢复为就绪状态，只是线程 2 和线程 1 优先级相同，因此线程 2 继续运行，可以从终端看到如下信息：

```
thread2: send message - this is message No.0
……
thread2: send message - this is message No.10
```

```
thread2: send message - this is message No.11
```

接下来的信息需要仔细分析：

```
thread2: send message - thread1: recv msg from msg queue, the content:this is message No.0
tg queue, the content:thread2: send message - this is message No.13
```

红色字体是线程 2 要发送的字符串，显然它没有发送完整；绿色字体为线程 1 发送字符串，因此我们可以得出结论，线程 2 的时间片在发送完红色字符串后耗尽。由于三个线程优先级相同，线程 3 依然处于挂起状态，因此线程 1 被调度执行，在线程 1 处理函数中，从

rt_mq_recv 函数返回，接下来调用

```
rt_kprintf("thread1: recv msg from msg queue, the content:%s\n", buf);
```

将接收到的消息输出到串口上。

接下来线程 1 执行 rt_thread_delay 延时 1 秒中，线程 1 被挂起，此时线程 3 依然处于挂起状态，线程 2 继续运行，故输出

```
thread2: send message - this is message No.13
```

之后线程 2 继续向消息队列中发送消息，并同时向串口输出

```
thread2: send message - this is message No.14
thread2: send message - this is message No.15
```

此时，消息队列已满，thread2 打印如下信息后，线程 2 处理函数退出。

```
message queue full, thread2 leave
```

一段时间之后（刚才线程 1 不是被挂起 1 秒钟么），线程 1 调度运行，并打印

```
thread1: recv msg from msg queue, the content:this is message No.1
```

继续休眠 1 秒钟，内核执行 IDLE 线程，1 秒钟后，线程 1 恢复运行，再次获取消息，重复这个过程。在这个循环的过程中，向串口上打印如下信息：

```
thread1: recv msg from msg queue, the content:this is message No.1
thread1: recv msg from msg queue, the content:this is message No.2
thread1: recv msg from msg queue, the content:this is message No.3
```

```
thread1: recv msg from msg queue, the content:this is message No.4
```

接下来串口信息为：

```
thread3: send an urgent message <over>
```

这表明线程 3 的 5 秒定时终于到达，线程 3 被唤醒，并在线程 1 某个休眠的时间中调度运行，它会打印上面这条信息，并使用 rt_mq_urgent 函数向消息队列中发送一条紧急消息，之所以称之为紧急消息，是因为这个函数会将消息插入到消息队列的头部。之后线程 3 的处理函数也执行完毕退出。

一段时间后，线程 1 调度运行，它从消息队列中获取消息，会得到刚才线程 3 发送的紧急消息，线程 1 跳出接收循环，打印

thread1: recv msg from msg queue, the content:over

thread1: got an urgent message, leave

线程 1 的线程处理函数也退出。

以上就是整个实验中，各个线程的状态转换过程。

5 总结

本实验演示了 RT-Thread 中消息队列作为多线程通信的用法，以静态消息控制块为例，动态消息队列的用法类似，只是创建/删除需要使用 rt_mq_create/rt_mq_delete 函数，读者可以使用动态消息队列重复本实验。