# 第 5 章

# 任务间通信

**本章目标**

- 介绍任务间通信和交互的原因。
- 说明任务之间如何相互通信。
- 描述同步和协同(非同步)任务交互。
- 说明何处、何时应优先选择任务协同而不是任务同步及选择的原因。
- 引入协同标志、事件标志和事件标志组。
- 解释单向和双向同步的概念与使用。
- 说明如何使用内存池和队列实现任务间的数据传输。
- 展示邮箱如何实现带任务同步的数据传输。

## 5.1 简介

### 5.1.1 任务间通信概述

我们之前已经了解到,从软件的角度来看任务可能是独立的,但这并不是常见的情况,任务间通常是相互作用的。而且事实证明,任务间的交互有三种不同的通信形式,如图 5.1 所示。

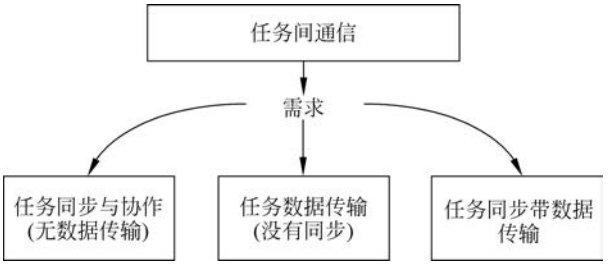


图 5.1 任务间通信分类

第一种交互方式,任务通信可以在不交换数据的情况下同步和/或协同它们的动作。同步和协同需求通常发生在任务由事件(或事件序列)而不是数据建立关联的地方。这些事件

包括与时间相关的因素，如延迟时间、运行时间和系统时间。例如，考虑这样一个需求：显示任务需要更新状态信息以响应来自键盘处理任务的命令。这里的键盘处理任务与显示任务之间没有数据传输，只有事件信号。

第二种交互方式，任务必须交换数据，但不需要同步操作。在控制系统示例中，信息由显示输出任务控制，数据从其他任务获得。它们没有必要同步工作，可以很好地异步运行，仅根据需要传输数据即可。

第三种交互方式，任务可能必须在同步的时间点交换数据。例如，测量任务的输出也是计算任务的输入，意味着有数据传输需求，但同样重要的是，计算任务只处理最新的信息，因此它与测量过程同步工作。

为了安全有效地运作，分别为这三种功能开发了单独的机制。在后面的章节中会对它们进行详细的介绍。

### 5.1.2 协同与同步

首先，让我们弄明白“协同”和“同步”这两个术语的含义。《钱伯斯英语词典》对其定义如下。

协同：整合和调整众多不同的部分或过程，以使彼此之间顺畅地联系起来。

同步：使与其他事物或彼此在精确时间内发生、移动或运行。

有时任务同步和任务协同之间的界限会有些模糊，它们之间关键的区别在于“精确时间”一词。协同忽略了时间的精确性，从根本上说，它旨在确保任务以正确的顺序和/或在满足特定条件时运行。

例如，需要实现以下规范：

“在所有互锁都清除，所有警报都解除之后才能启动压缩机”。

假设将软件设计为包含一个互锁任务、一个警报任务和一个压缩机任务。为了符合规范，压缩机任务必须延迟启动，直到其他任务完成特定的工作。

(1) 对互锁和警报任务的完成顺序并无要求。

(2) 不限制压缩机任务何时执行检查以确定是否可以开始启动序列。

(3) 互锁和警报任务提供了所需的事件信息(互锁清除，警报解除)后，它们可以继续执行其他操作。

(4) 压缩机任务在等待条件满足的同时，继续进行其他操作。

因此，在协同活动时发送方或接收方都不需要进入等待或挂起模式。

这与需要同步其操作的活动完全不同。一个自动物料处理的例子如图 5.2 所示，这里使用了托盘运输机器人和物料装载/卸载(码垛机)机器人两个机器

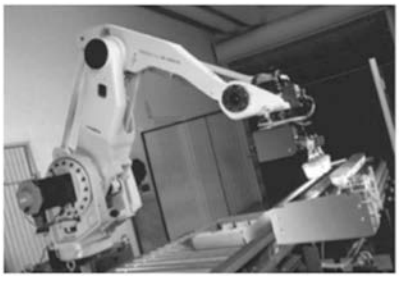

图 5.2 自动物料处理
(注：查看 YouTube 上 Lambert Material Handling 典型码垛机操作的视频；来源：Lambert Material Handling，美国纽约州雪城)

人。运输机器人的功能是在工厂中移动托盘，而码垛机器人的功能是卸货和/或将物品装载到托盘上(码垛)，所有操作都由软件控制，由运输任务和码垛任务完成。

在物料被转移到托盘或从托盘转移之前，两个机器人必须处于正确的位置。我们要做的是：

(1) 正确定位两个机器人(在同步点或集合点)。

(2) 执行所需的加载/卸载操作。

(3) 恢复单个机器人操作。

由于这两台机器人是独立的单元，我们无法预测哪个先准备好开始码垛，因此，如果运输机器人是第一个到位的，它必须等码垛机器人准备好；同样地，如果码垛机器人首先就绪，它必须等运输机器人也准备好。这对于代码设计意味着必须在每个任务的代码中确定同步发生的确切位置，在这些位置插入同步机制。

实现软件的协同和同步的结构有条件标志、事件标志和信号三种(见图 5.3)。对于两种标志而言，操作是设置、清除和检查(读取)，而对于信号而言，操作是等待、发送和检查(此处的术语选择考虑嵌入式系统的准确性、清晰性和历史用法)。请注意：在许多 RTOS 中，预定义的标志结构是事件标志。

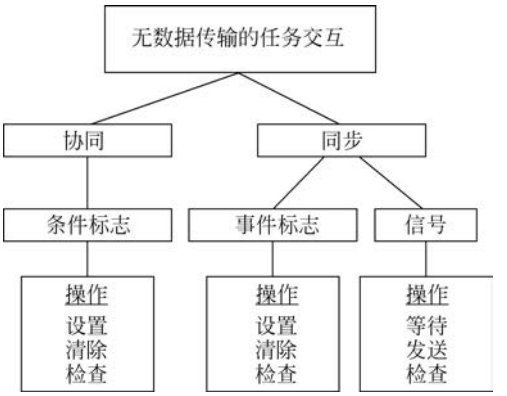


图 5.3　协同和同步结构

## 5.2　无数据传输的任务交互

### 5.2.1　任务协同机制

#### 1. 简单条件标志

第 3 章中已经介绍了标志的各种应用，基本的想法很简单，也展示了如何将它们用作忙等待互斥机制。但是在这里，它们的功能是允许任务协同它们的活动，在这个角色中，它们被称为条件标志。请注意，使用条件标志这个术语，是为了避免与事件标志混淆，在实践中条件标志简称为标志。

思考如图 5.4(a)所示的简单信令要求。

在图 5.4(a)中，HMI 任务向电机控制任务发送启动和停止命令。这些命令是响应操作员键盘(未显示)的输入而生成的。这两个任务都需要连续运行，系统才能正常运行。实现该需求的最简单方法是使用一个全局变量，同时这也是一个糟糕的方法，全局变量的问题是众所周知的。基于任务的设计的一个基本规则是所有的任务间通信都使用合适的通信组件。图 5.4(b)不言而喻，它显示了如何在此应用中使用标志。

在代码级别，条件标志是一个二进制(两值)项，最好实现为布尔值。如果该数据类型不可用，则可以使用字、字节、位或枚举类型(RTOS 定义的数据类型通常不包含普通标志，因

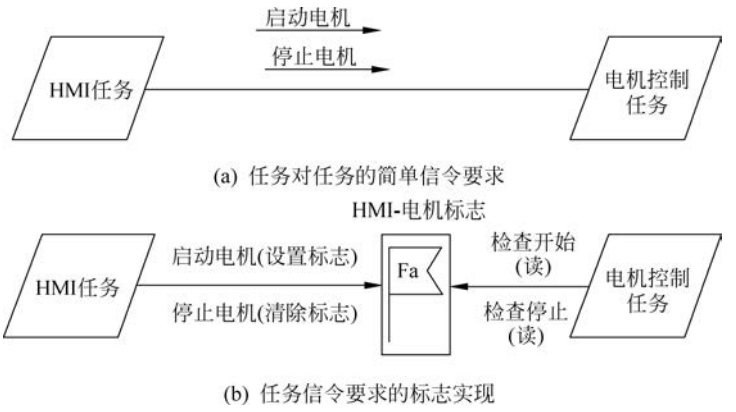


图 5.4 简单使用标志进行协同

为这是基本数据类型)。

这种设计虽然满足了要求,但健壮性并不是很好。其中一个缺点是必须在代码级别上知道“设置(set)”表示开始,“清除(clear)”表示停止。结果是:

(1) 这两者很容易混淆,容易出错。

(2) 在检查代码时更难发现错误。

最好避免使用“set”和“clear”这两个词,而是使用一种自记录代码的形式,例如代码清单 5.1。

**代码清单 5.1**

```
typedef enum {Start, Stop} StartStopFlag;
StartStopFlag HMImotorFlag = Stop;
```

还建议将所有通信组件封装在通信类或通信文件中,对于许多应用程序而言这个解决方案是完全可以接受的。然而,它的缺点是一个错误(例如使用 Start 替代了 Stop)可能会在现实世界中产生严重的问题。现在看到的是一个有效命令被转换成另一个有效命令的情况,没有信息冗余。更安全的设计如图 5.5 所示,其中每个单独的命令都有一个对应的标志。

这里使用的规则是:

(1) HMI 发送任务在改变状态前检查标志是否被重置。

(2) 电机控制任务在执行命令前检查两个标志的状态。检查成功后,命令标志被重置。

基本的代码结构如代码清单 5.2 所示(当然,这不是唯一可以使用的规则)。

**代码清单 5.2**

```
typedef enum {StartSet, StartReset} StartFlag;
typedef enum {StopSet, StopReset} StopFlag;

StartFlag MotorStartFlag = StartReset;
StopFlag MotorStopFlag = StopReset;
```

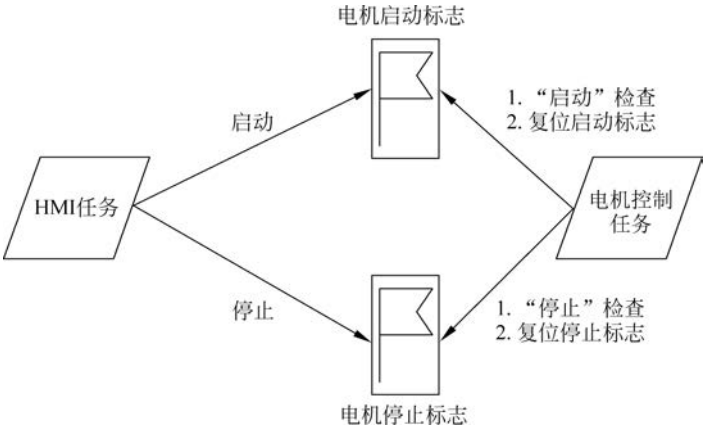


图 5.5　改进后使用标志进行协同

**2. 条件标志组**

现在看看条件标志组(见图 5.6),也就是将一组标志组合成一个单元。条件标志组通常是一个变量,每个标志位于变量中的一位。因此:

(1) 每位都可以单独改变。

(2) 整个组可以使用单个写入命令修改。

(3) 一组位可以使用位屏蔽方式改变。

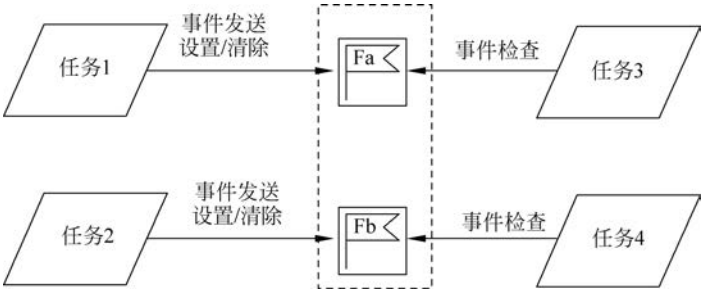


图 5.6　任务协同——条件标志组

如图 5.6 所示,单个标志组可以替换多个单独的标志。但是请注意:最好不要把普通标志建立在这样的结构上,否则一个简单的编程错误可能会对系统造成严重破坏,可以自行找出原因。

标志组在以下两种情况下特别有用。

(1) 任务正在等待一组事件(见图 5.7)。

(2) 任务向许多其他任务广播事件(见图 5.8)。

如图 5.7 所示,标志组便于实现下列组合逻辑操作。

(1) 逻辑与:只有当互锁和报警清除后,才能启动电机。

(2) 逻辑或:如果检测到超速或高温,必须停止电机。

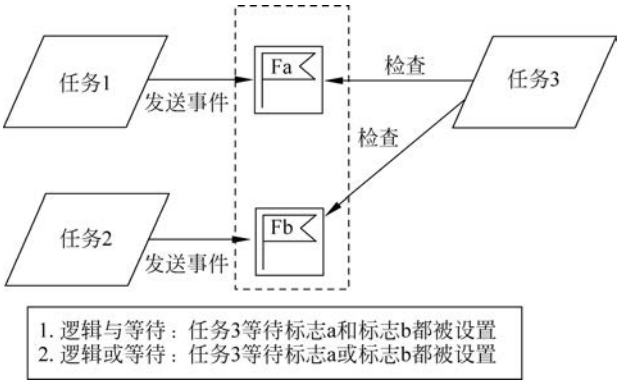


图 5.7　条件标志组——等待一组事件

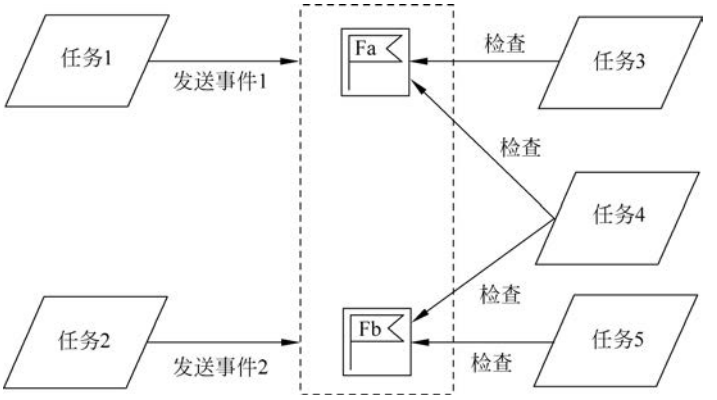


图 5.8　条件标志组——广播功能

(3) 复杂的决策：当发酵完成且碱液温度低于 30℃或操作人员手动选择“清空罐”时，将自动清空罐。

图 5.8 显示了广播功能的实际应用。这里的标志 Fa 允许任务 1 向任务 3 和任务 4 广播，而标志 Fb 允许任务 2 向任务 4 和任务 5 广播。

## 5.2.2　使用事件标志单向同步任务

单向同步是一种有限同步形式，但在某些情况下非常有效。考虑燃料箱保护系统中的一个要求，即一旦检测到火焰，就必须激发气体抑制剂瓶。为了防止爆炸，响应必须非常快，从检测到完成通常不到 10ms。我们的实现是可以让火焰探测器产生一个中断，立即调用灭火任务。因此，软件交互涉及两个任务：火焰探测中断服务程序(ISR)和灭火任务。ISR 是发送者任务，灭火任务是接收者任务。注意，这个接收者任务是非周期性的，通常它处于挂起状态，等待被发送方任务唤醒(等待会合)。相比之下，发送任务不等待与接收者同步，它仅在同步条件满足时发信号。这种单向同步如图 5.9 所示。

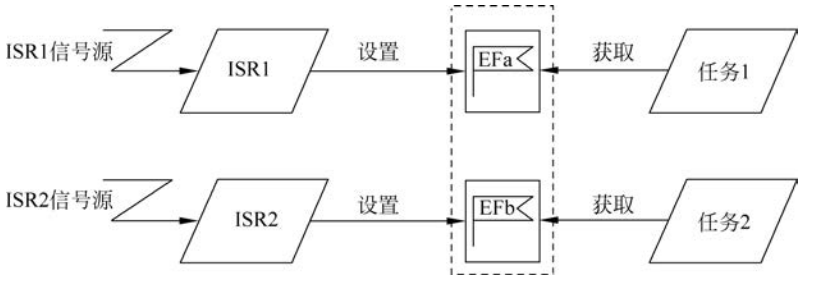


图 5.9　事件标志和单向同步

这里的事件标志用于支持任务间的交互，基本规则如下。

(1) ISR1/ISR2 是发送者任务，任务 1/任务 2 是对应的接收者。

(2) 事件标志初始化为清除状态(标志值＝0)。

(3) 当任务在清除标志上调用“获取”时，该任务会被挂起。

(4) 当任务在设置标志(标志值＝1)上调用“获取”时，它会清除标志并继续执行。

(5) 当任务在清除标志上调用“设置”时，它会设置该标志并继续执行。如果任务在标志上等待挂起，则该任务被唤醒(就绪)。

(6) 当一个任务在设置标志上调用“设置”时，它会继续执行。

例如，在图 5.9 中：

(1) 如果任务 1 在 ISR1 生成“设置”之前调用“获取”，则它被挂起。随后当“设置”设置标志(EFa)后，任务 1 就绪。

(2) 如果“设置”是在任务 1 调用“获取”之前生成的(即 ISR 任务先到达同步点)，ISR1 不会停止而是继续执行。“设置”调用使标志 EFa 处于设置状态。因此，当任务 1 调用“获取”时，它首先清除该标志，然后继续执行。但是请注意：单向同步通常用于使发送方任务唤醒接收方，是“延迟服务器”的一种。

下面来看一个基于 ThreadX 的简单示例(见图 5.10)，事件标志组是一个 32 位的变量。

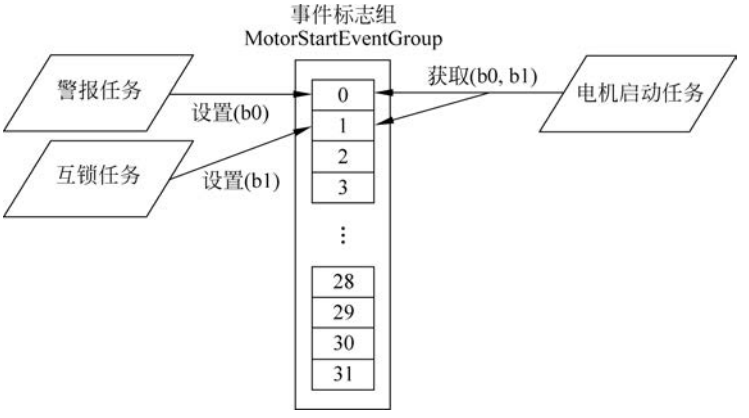


图 5.10　事件标志组——等待事件集

事件标志组的目的是通过单向同步实现“只有当互锁和警报清除时才能启动电机”的规范。当互锁清除时，互锁任务将设置事件标志组的位 1。当警报清除时，警报任务设置位 0。除非两位都被设置(值为十六进制的 00000003，简写为 03H 或 0x03)，否则接收者电机启动任务无法继续通过同步点。

代码清单 5.3 展示了基本声明和事件标志组的创建。代码清单 5.4 和代码清单 5.5 展示了如何设置标志组的各位，而代码清单 5.6 展示了接收方任务的操作。相关的注释说明了代码的作用。

**代码清单 5.3**

```
/* ============== 声明 ========================== */
TX_EVENT_FLAGS_GROUP MotorStartEventGroup;
uint GroupCreationStatus;
uint SetServiceStatus;
uint GetServiceStatus;
/* 代码中 */
/* ============ 创建事件标志组 ====================
*/
/*
所有位都自动初始化为零，即被清除
*/
GroupCreationStatus = tx_event_flags_create (&MotorStartEventGroup,
                                             "MotorStartEventGroup");
```

**代码清单 5.4**

```
/* ========== 警报任务中的设置操作 ================ */
/*
将标志值与 0x01 进行逻辑或操作以设置位 0
*/
SetServiceStatus = tx_event_flags_set (&MotorStartEventGroup, 0x01, TX_OR);
```

**代码清单 5.5**

```
/* ============ 互锁任务中的设置操作 ===============
*/
/*
将标志值与 0x02 进行逻辑或操作以设置位 1
*/
SetServiceStatus = tx_event_flags_set (&MotorStartEventGroup, 0x02, TX_OR)
```

**代码清单 5.6**

```
/* ========电机启动任务中的设置操作 =================
*/
/*
```

```
此处是同步点
任务等待被无限期地挂起,直到位 0 和 1 被设置(即标志的值为 0x03,所有其他的位忽略)
当发生这种情况时,事件标志被清除,任务继续执行
*/
GetServiceStatus = tx_event_flags_get (&MotorStartEventGroup, 0x03,
TX_AND_CLEAR, TX_WAIT_FOREVER);
```

### 5.2.3　使用信号双向同步任务

图 5.2 所示的物料搬运机器人动作的同步需要使用双向同步。双向同步是通过使用信号实现的,这些信号操作包括“等待”“发送”和“检查”(见图 5.11)。

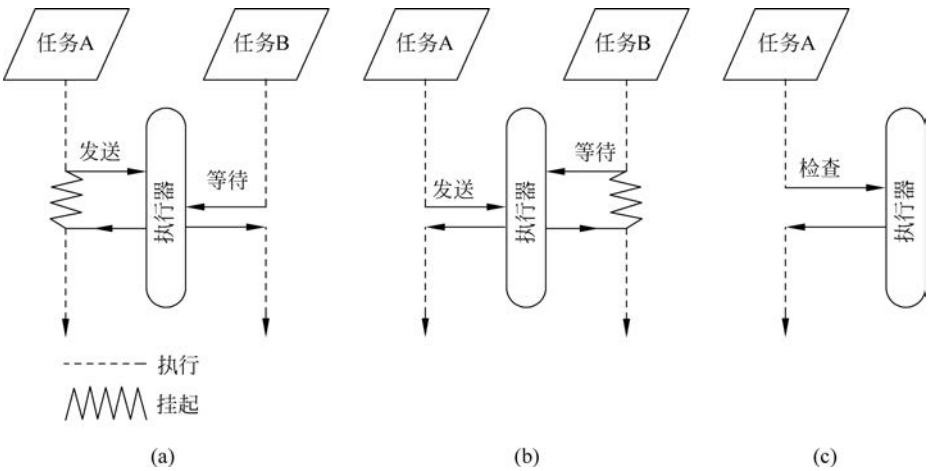


图 5.11　使用信号进行任务同步

发出信号的动作是执行者的责任,对用户来说这类操作是透明的。

首先来看“发送”动作,如图 5.11(a)所示。任务 A 执行它的程序到达发送信号的位置,并成功地将信号发给执行器。在此时刻没有任务等待接收这个信号,因此任务 A 被挂起。一段时间后,任务 B 对任务 A 发出的信号产生等待请求。任务 B 获得信号后继续执行,等待请求还会重新启动任务 A。

如果任务 B 在任务 A 发送信号之前产生等待会发生什么(见图 5.11(b))? 结果会是任务 B 被挂起,直到任务 A 发送信号。此时任务 A 唤醒任务 B,然后继续执行。

允许任务决定是否参与同步可以为信号的构造增加灵活性。在图 5.11(c)中,检查操作会检查信号的状态,但本身不会停止任务执行。决策留给检查任务,可以非常有效地用于轮询操作。

在实践中,信号通常使用函数实现,如代码清单 5.7 所示。

**代码清单 5.7**

```
void Send (SyncSignal SignalName);
/* 向同步信号量 SignalName 发送一个信号,如果没有任务等待信号则挂起 */
void Wait (SyncSignal SignalName);

/* 等待信号,如果在生成请求时没有信号,任务就会挂起,否则将重新激活发送方并重新调度系
统 */
typedef enum { false, true } bool;

bool Check(SyncSignal SignalName);
/* 检查是否有任务等待发送信号,如果有信号则返回 true */
```

下面是一些需要说明的重点。

(1) 任务之间没有一对一的联系,在这些构造中没有指定任务配对。

(2) 任务被认为既是发送者又是接收者。

(3) 信号与特定的任务无关。一个任务需要“等待”,而另一个任务需要相应地“发送”。

(4) 它们展示出信号量的不安全性。

(5) 信号看起来非常像二进制信号量,它们的实现非常相似。它们之间的根本区别在于使用方式,而不是构造。信号量通常用作互斥机制,信号用于同步。

(6) 很少有 RTOS 提供双向同步结构。

信号量可以用于创建信号,但不是简单的一对一关系。一种设计方法是使用两个信号量(每个信号方向一个,见图 5.12)来构建单个信号。用于实现该功能的伪代码见代码清单 5.8。代码清单 5.9 给出了一个基于 Pthreads 的更完整的示例。这里与发送信号等价的是 Pthreads 构造的“发布”(post)。

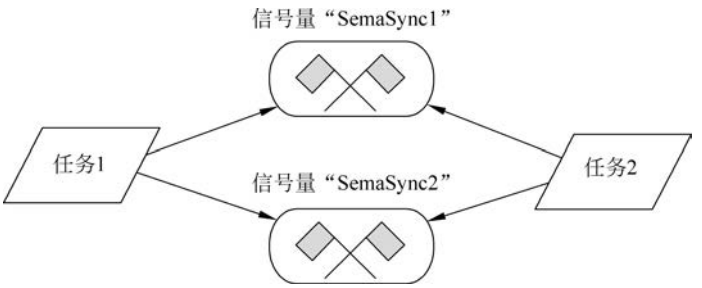


图 5.12 使用信号量来模拟信号

**代码清单 5.8**

```
/* 任务 1 代码: */
    while (1)
    {
        代码语句;
        /* 同步点 */
```

```
        Signal (SemaSync2);
        Wait (SemaSync1)
        代码语句;
    } /* 循环结束 */

/* 任务 2 代码: */
    while (1)
    {
        代码语句;
        /* 同步点 */
        Signal (SemaSync1);
        Wait (SemaSync2);
        代码语句;
} /* 循环结束 */
```

**代码清单 5.9**

```
/* 简单的 pthread 示例 - 使用信号量同步 */
/* ------------------------------------------------- */
main 代码单元
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

void * Task1(void *);
void * Task2(void *);

#define NUM_THREADS 2
pthread_t tid[NUM_THREADS];              /* 线程 ID 数组 */
sem_t SemaSync1, SemaSync2;              /* 信号量 */
main( int argc, char *argv[] )
{
    int i;
    sem_init(&SemaSync1, 0, 0);
    sem_init(&SemaSync2, 0, 0);
    pthread_create(&tid[0], NULL, Task1, NULL);
    pthread_create(&tid[1], NULL, Task2, NULL);
    for ( i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
    /* 其他代码语句 */
} /* main 结束 */
/* -------任务 1 代码-------------------------------- */
void * Task1(void * parm)
{
    /* 同步之前的代码 */
    /* 同步点 */
    sem_post(&SemaSync2);
    sem_wait(&SemaSync1);
```

```
    /* 同步之后的代码 */
}
/* -------任务 1 代码结束---------------------------- */
/* -------任务 2 代码------------------------------- */
void * Task2(void * parm)
{
    /* 同步之前的代码 */
    /* 同步点 */
    sem_post(&SemaSync1);
    sem_wait(&SemaSync2);
    /* 同步之后的代码 */
}
/* -------任务 2 代码结束---------------------------- */
```

最后一个例子展示了分散在任务中的信号量操作。如果在实践中选择使用这种方法，会以发生事故而告终。更好的做法是使用监视器类型的技术，图 5.13 从概念上给出了该技术的结构。

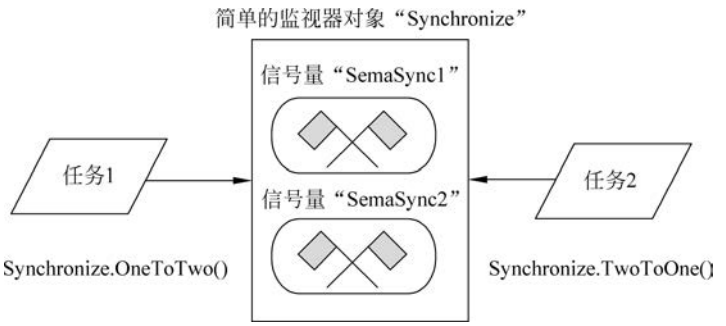


图 5.13 基于信号量的更安全的信号结构

## 5.3 无任务同步或协同的数据传输

### 5.3.1 概述

在许多情况下，任务交换信息时不需要任何同步或协同。可以通过包含互斥特性的直接数据存储来实现此类需求。在实践中使用了两种数据存储机制——内存池(pool)和队列，如图 5.14(a)所示。队列也称为通道、缓冲区或管道。

### 5.3.2 内存池

内存池是一个可读可写随机访问的数据存储，见图 5.14(b)。通常用它来保存进程的共有的项目，例如系数值、系统表、报警设置等。图 5.14(b)中显示任务 A 和任务 C 将数据存入池中，而任务 B 将信息从池中读取出来。读取操作不是破坏性的，即池中的信息不会

因读取操作而改变。内存池由可读写的内存组成，通常是 RAM(以读操作为主时可以使用闪存)。可以使用记录或结构(C 或 C++中的结构类型)轻松地创建内存池，因此它们通常不作为 RTOS 特定的类型提供。构建一个内存池时，它应该像所有任务通信组件一样健壮和安全。存储自身应该封装为私有项，并包含互斥保护(见图 5.15)，只能通过访问接口中提供的公共函数访问内存池的数据。

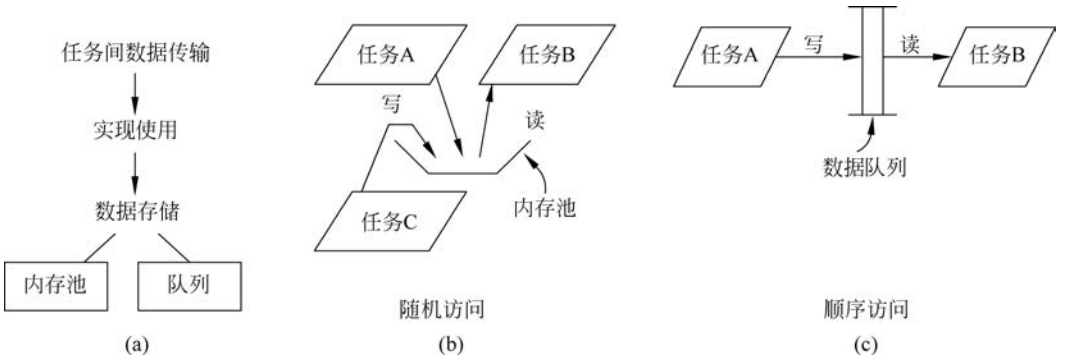


图 5.14　任务间数据传输

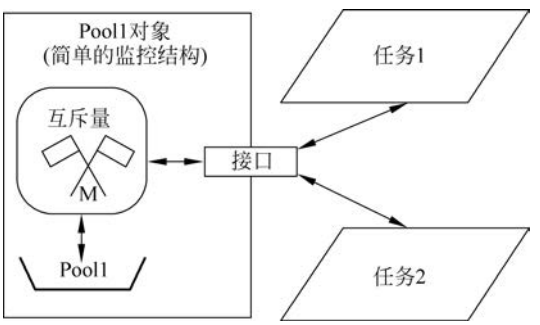


图 5.15　受保护的内存池

在实际系统中，可根据需要使用多个内存池，这限制了对信息的访问，从而避免了全局数据的问题。

### 5.3.3　队列

队列被用作进程之间的通信管道，通常是一对一的，见图 5.14(c)。任务 A 将信息存入队列，任务 B 以先进先出的方式提取信息。队列通常应足够大，可以承载许多数据，而不仅仅承载单个数据项。因此，它可以充当缓冲或暂存器，为管道提供灵活性。它的优点是插入和提取功能可以异步进行(只要管道没有填满)。它在 RAM 中实现。进程之间传递的信息可能是数据本身，也可能是指向数据的指针。指针通常用于在 RAM 存储受限时处理大量数据。实现队列的技术有两种：链表类型结构和循环缓冲区。

链表类型结构已经在调度部分讨论过，无须进一步详细描述。链表的一个非常有用的特性是它的大小不一定是固定的，而是可以根据需要扩大或缩小。此外，可以构建非常大的

队列，仅受可用内存空间的限制。但是对于嵌入式系统来说，这些并不是特别的优势。首先，如果 RAM 有限，则根本不可能构造很大的队列。其次，处理多个消息的大型 FIFO 队列从数据输入到数据输出可能会有较长的传输延迟，就性能而言，对于许多实时应用可能太慢。因此，用于嵌入式的首选队列（通道）结构是循环缓冲区，如图 5.16 所示。

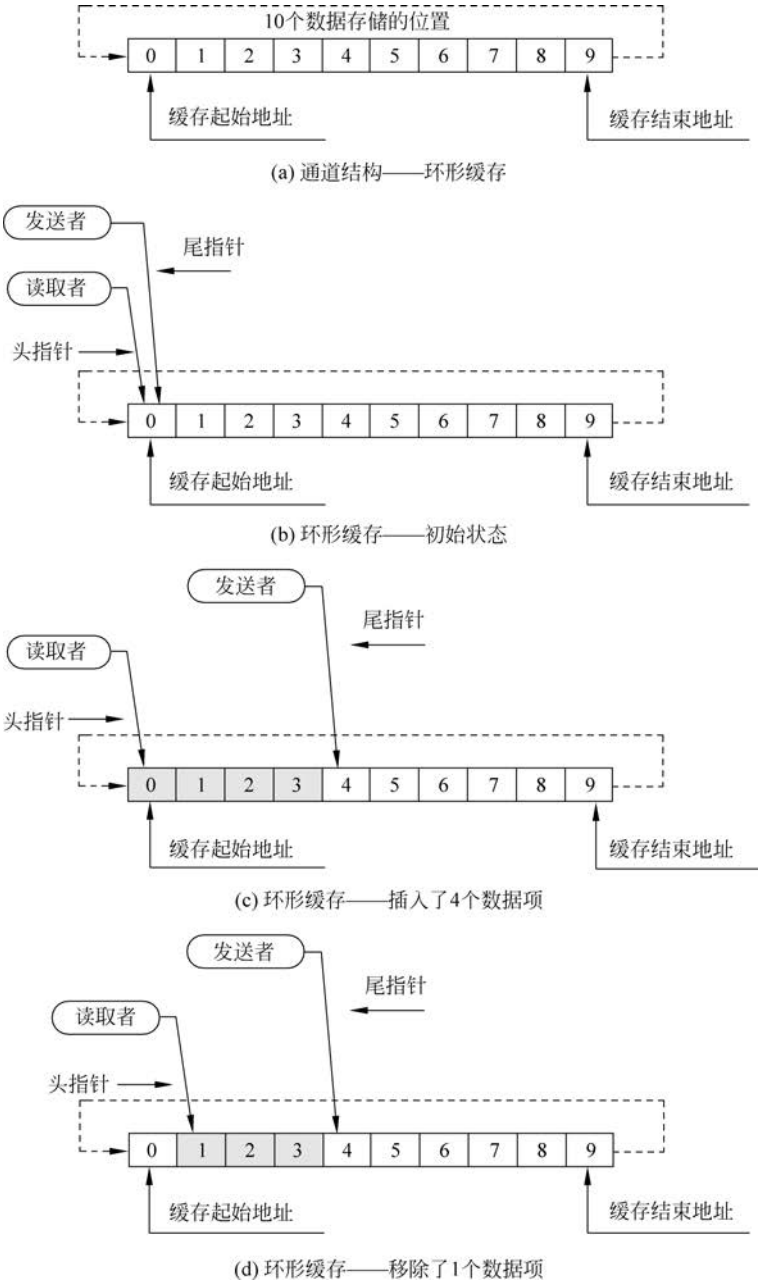


图 5.16　循环缓冲区——用于队列的循环存储

循环缓冲区通常设计为使用固定数量的内存空间，用来保存一定数量的数据，如图 5.16(a)所示。缓冲区大小是在创建时定义的(例如这里是 10 个数据单位)，但在之后是固定的。使其循环的原因是数据单元 0 是数据单元 9 的后继，寻址是使用模 9 计数器完成的(就像 12 小时时钟使用模 12 计算一样)。

在读写操作期间，数据可以在通道中移动。但是，一般来说这会带来不可接受的时间开销，这里使用另一种方法，图 5.16(b)展示了如何使用指针来标识存储数据的起始和结束位置(“读取者”和“发送者”)。通过指针，不必在缓冲区中移动数据。插入的数据单元始终位于相同的内存位置，仅需改变指针的值，如图 5.16(c)和图 5.16(d)所示。这些指针也可以用来定义队列满和队列空的条件(当它们相等时)。

在正常情况下，任务 A 和任务 B 异步进行，根据需要从队列中插入和删除数据。任务挂起只在两种情况下发生：队列满和队列空。如果队列满时，任务 A 尝试加载一个数据单元，那么任务 A 将被挂起。同样，如果队列空时，任务 B 试图读取一个数据单元，则任务 B 被挂起。在很多实现中挂起不会发生，而是会触发错误异常。

内存池和队列之间有一个重要的区别——内存池读取数据不会影响内容，但是从队列读取时会“消耗”数据，即破坏性操作(实际上这只是概念性看法，读指针只是移到了下一个位置)。

代码清单 5.10～代码清单 5.13 给出了队列使用的概要。

**代码清单 5.10**

```
/* 基础 API */
/* 1. 创建队列 */
FOS_CreateQueue(QLength, QItemSize);
/* 2. 从队列获取消息 */
FOS_GetFromQueue(QName, AddOfQData, QwaitingTime);
/* 3. 向队列发送消息 */
FOS_SendToQueue(QName, AddOfQData, QwaitingTime);
```

**代码清单 5.11**

```
/* 创建一个全局的队列联结发送任务 A 和接收任务 B */
/* 使用 RTOS 提供的数据类型 */
FOS_QName GlobalQA2B;
FOS_QLength QA2Blength = 1;
FOS_ItemSize QA2BItemSize = 4;
GlobalQA2B = FOS_CreateQueue (QA2Blength, QA2BItemSize);
```

**代码清单 5.12**

```
/* 发送到队列 - 任务 A */
/* 使用 RTOS 提供的数据类型 */
long DataForQueueA2B;
const FOS_QwaitTime NoWaiting = 0;
```

```
FOS_ QloadStatus QLoadState;
QLoadState = FOS_SendToQueue (GlobalQA2B, &DataForQueueA2B, 0);
```

**代码清单 5.13**

```
/* 从队列获取 - 任务 B */
/* 使用 RTOS 提供的数据类型 */
long DataFromQueueA2B;
const FOS_QwaitTime NoWaiting = 0;
FOS_QreadStatus QreadState;
QreadState = FOS_GetFromQueue (GlobalQA2B, & DataFromQueueA2B, NoWaiting);
```

## 5.4 有数据传输的任务同步

如前所述，在某些情况下，任务不仅等待事件，而且还使用与这些事件相关的数据。为了实现这个目的，需要同步机制和数据存储区域。此时可以使用的结构是邮箱，邮箱包含同步信号和数据存储，如图 5.17 所示。

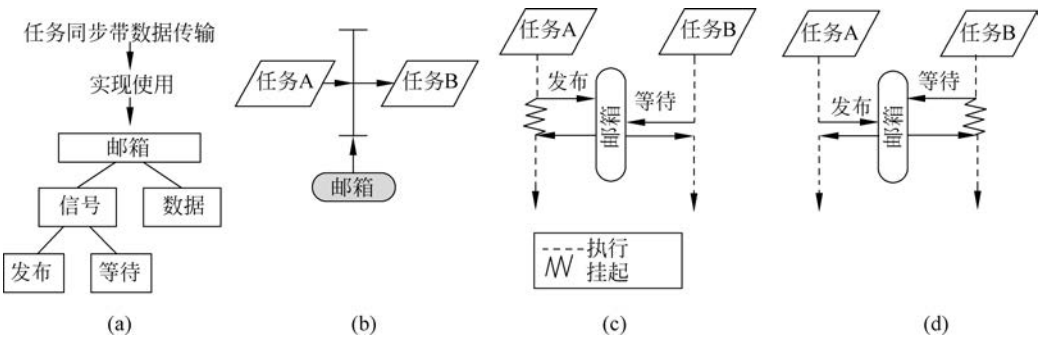


图 5.17 任务同步的数据传输

当一个任务希望向另一个任务发送信息时，它将数据发布(post)到邮箱。相应地，当一个任务在邮箱中查找数据时，它会等待(pend)。实际上，发布和等待是信号。此外，数据本身通常不通过邮箱传递，而是使用数据指针。无论数据内容有多大，数据都被视为一个单元。

因此，从概念上讲只有一个存储项。任务同步是通过挂起任务直到满足所需条件来实现的，如图 5.17(c)和图 5.17(d)所示。任何发布到没有任务等待的邮箱的任务都会被挂起，当接收者等待信息时它才会恢复。相反，如果等待操作先发生，则任务将挂起直到发布操作发生。

邮箱经常用作多对多通信管道。这比一对一结构的安全性要低得多，在关键应用程序中是不可取的。

代码清单 5.14～代码清单 5.17 给出了基于 MicroC/OS-Ⅱ API 的邮箱使用的概要。

**代码清单 5.14**

```
/* 基础 API */
/* 1. 创建一个空的邮箱 */
OSMboxCreate ((void *) 0);
/* 2. 往邮箱存一条消息 */
OSMboxPost (MBoxName, AddOfMessage);
/* 3. 从邮箱收取一条消息 */
OSMboxPend (MBoxName, TimeOutValue, ErrorCode);
```

**代码清单 5.15**

```
/* 创建联结发送任务 A 和接收任务 B 的全局邮箱 */
OS_EVENT *GlobalMBoxA2B;
GlobalMBoxA2B = OSMboxCreate ((void *) 0);
```

**代码清单 5.16**

```
/* 发送到邮箱 - 任务 A */
INT8U DataForMBoxA2B[50];
INT8U err;
err = OSMboxPost (GlobalMBoxA2B, (void *) &DataForMBoxA2B[0]);
```

**代码清单 5.17**

```
/* 从邮箱获取 - 任务 B */
void * DataFromMBoxA2B;
INT8U Wait200 = 200;
INT8U err;
DataFromMBoxA2B = OSMboxPend (GlobalMBoxA2B, Wait200, &err);
```

## 5.5 回顾

通过本章的学习,应该能够达到以下目标。

- 了解为什么任务通常会相互通信和交互。
- 清楚了交互关系可以分为三类：没有数据传输的交互、任务间异步数据传输和同步数据传输。
- 知道协同标志和事件标志的用途以及它们与标志组的关系。
- 清楚使用标志组的原因。
- 了解事件标志如何支持单向同步。
- 理解信号是什么以及它如何实现双向同步。
- 知道如何用信号量或互斥量构造信号。
- 理解使用两种数据传输机制(内存池和队列)的原因。

- 清楚内存池和队列是如何构建的。
- 了解内存池和队列这两种方法的差异、优点和缺点。
- 理解循环缓冲区的工作原理并了解为什么使用它来实现数据传输通道。
- 了解邮箱的功能。
- 清楚邮箱的工作原理和使用邮箱的原因。

此时,应该完成《嵌入式实时操作系统——基于 STM32Cube、FreeRTOS 和 Tracealyzer 的应用开发》中剩下的实验。